Converting Java Programs to Use Generic Libraries

Alan Donovan Adam Kieżun Matthew S. Tschantz Michael D. Ernst

MIT Computer Science & Artificial Intelligence Lab
32 Vassar St, Cambridge, MA 02139 USA
{adonovan,akiezun,tschantz,mernst}@csail.mit.edu

ABSTRACT

Java 1.5 will include a type system (called JSR-14) that supports *parametric polymorphism*, or *generic* classes. This will bring many benefits to Java programmers, not least because current Java practice makes heavy use of logically-generic classes, including container classes.

Translation of Java source code into semantically equivalent JSR-14 source code requires two steps: parameterization (adding type parameters to class definitions) and instantiation (adding the type arguments at each use of a parameterized class). Parameterization need be done only once for a class, whereas instantiation must be performed for each client, of which there are potentially many more. Therefore, this work focuses on the instantiation problem. We present a technique to determine sound and precise JSR-14 types at each use of a class for which a generic type specification is available. Our approach uses a precise and context-sensitive pointer analysis to determine possible types at allocation sites, and a set-constraint-based analysis (that incorporates guarded, or conditional, constraints) to choose consistent types for both allocation and declaration sites. The technique handles all features of the JSR-14 type system, notably the raw types that provide backward compatibility. We have implemented our analysis in a tool that automatically inserts type parameters into Java code, and we report its performance when applied to a number of real-world Java programs.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*

General Terms

languages, theory, experimentation

Keywords

generic types, parameterized types, parametric polymorphism, type inference, instantiation types, JSR-14, Java 1.5, Java 5, raw types

1. INTRODUCTION

The next release of the Java programming language [21] will include support for generic types. Generic types (or parametric polymorphism [7]) make it possible to write a class or procedure abstracted over the types of its method arguments.

In the absence of generic types, Java programmers have been writing and using pseudo-generic classes, which are usually expressed in terms of Object. Clients of such classes widen (up-cast) all the actual parameters to methods and narrow (down-cast) all the return values to the type at which the result is used — which can be thought of as the type at which the pseudo-generic class is 'instantiated' in a fragment of client code. This leads to at least two 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 types 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-generic 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 behavior of the program and the intention of the programmer. Generic types provide better documentation, and the type checker guarantees their accuracy.

Non-generic solutions to the problems (e.g., wrapper classes such as `StringVector`) are unsatisfying. They introduce nonstandard and sometimes inconsistent abstractions that require extra effort for programmers to understand. Furthermore, code duplication is error-prone.

Java with generic types (which we call JSR-14 after the Java Specification Request [4] that is being worked into Java 1.5) solves these problems while maintaining full interoperability with existing Java code.

Currently, programmers who wish to take advantage of the benefits of genericity in 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 JSR-14. There are two parts to this task: adding type parameters to class definitions ('parameterization'), and modifying uses of the classes to supply the type parameters ('instantiation').

Parameterization must be performed just once for each library class. The process might be done (perhaps with automated assistance) by an expert familiar with the library and how it is intended to be used. Even for a non-expert, this task may be relatively easy. For example, the *javac* compiler, the *htmlparser* program and the *antlr* parser generator define their own container classes in addition to, or in lieu of, the JDK collections. One of us (who had never seen the code before) fully annotated the *javac* libraries with generic types (135 annotations in a 859-line codebase) in 15 minutes, the *antlr* libraries in 20 minutes (72 annotations in a 532-line codebase) and the *htmlparser* libraries in 8 minutes (27 annotations in a 430-line codebase).

This paper focuses on the instantiation problem. Instantiation must be performed for every library client; there are typically many more clients than libraries, and many more programmers are involved. When a library is updated to use generic types, it is desirable to perform instantiation for legacy code that uses the library, though no one may be intimately familiar with the legacy code. Generic libraries are likely to appear before many programs that are written in a generic style (for example, Java 1.5 will be distributed with generic versions of the JDK libraries), and are likely to be a motivator for converting those programs to use generic types. Generically typed libraries permit programmers to incrementally add generics to their programs and gain benefits in a pay-as-you-go fashion.

Figures 2, 3, and 4 give an example of a (generic) library, non-generic client code, and the client code after being transformed by our tool. The library defines the class `Cell`, which is a container holding one element, and its subclass `Pair`, which holds two elements, possibly of different types. The client code defines a number of methods that create and manipulate `Cell`s and `Pair`s. The paper uses this code as a running example.

In brief, the generic type instantiation problem is as follows. The input is a set of generic (i.e., JSR-14 annotated) classes (which we call *library code*) and a set of non-generic (i.e., pre-JSR-14) classes (*client code*) that use the library code. The goal is to annotate the client code with generic type information in such a way that (a) the program's behavior remains unchanged, and (b) as many casts as possible can be removed. Later sections expand on this goal.

The remainder of this paper is organized as follows. Section 2 introduces JSR-14, a generic version of Java that is expected to be adopted for Java 1.5. Section 3 lays out our design goals and assumptions, and Section 4 overviews our algorithm. The next two sections describe the two parts of the algorithm, namely allocation type inference (Section 5) and declaration type inference (Section 6). Section 7 discusses the implementation of our prototype tool, and Section 8 presents preliminary experimental results. Section 9 discusses related work. Finally, Section 10 proposes future work, and Section 11 concludes.

2. JSR-14: JAVA WITH GENERIC TYPES

This section briefly introduces the syntax and semantics of JSR-14.

Generic types are an example of bounded parametric polymorphism [7]. Parametric polymorphism is an abstraction mechanism that permits a single piece of code to work uniformly over many distinct types in a type-safe manner. Type parameters stand for the types over which the code is (conceptually) instantiated.

2.1 Syntax

Figure 2 shows the definition of two generic classes in Java. The name of the generic class is followed by a list of type variables (V for class `Cell`, and F and S for class `Pair`). Each type variable has an optional upper bound or bounds. The default bound is `extends Object`, which may be omitted for brevity. The type variables may be used within the class just as ordinary types are, except that instances of the type variables may not be constructed by an instance creation expression, as in `new V()`. The `Pair` class shows that one generic type can extend (subclass) another. The scope of a class's type variable is essentially the same as the scope of `this`: all instance methods and declarations of instance fields, and any inner classes, but not static members or static nested classes. Also, a type parameter can be referred to in the class's declaration.

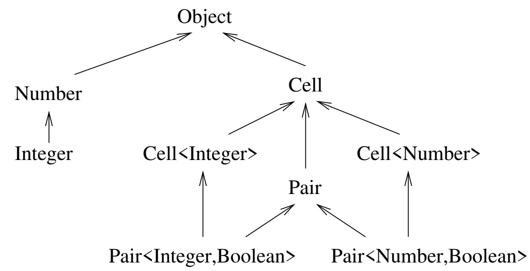

Figure 1: A portion of the type hierarchy for JSR-14 (Java with generic types), which uses invariant parametric subtyping. Arrows point from subtypes to supertypes. Classes Cell in Pair are defined in Figure 2.

A generic class may be instantiated (used) by supplying type arguments that are consistent with the bounds on the type variables (Figure 4). Type-checking ensures that the code is type-correct, no matter what type arguments that satisfy the bounds are used. (See below for a caveat regarding raw types.)

Methods may also be generic, adding their own additional type variables. In Figure 2, `replaceValue` is a generic method, which is preceded by a list of (bounded) type variables. (The type variable U has a non-trivial bound.) Type arguments at uses of generic methods need not be specified by the programmer; they are automatically inferred by the compiler. (Line 11 in Figure 4 contains a use of a generic method.) The scope of a method type variable is just the method itself.

A *raw type* is a generic type used without any type parameters. (On line 30 of Figure 4, parameter `c6` in `displayValue` is raw.) Raw types are a concession to backward compatibility, and they behave exactly like types in non-generic Java.

2.2 Type system

This section informally overviews salient points of the type system of JSR-14 [4]. Figure 1 shows part of the type hierarchy.

Invariant parametric subtyping. Different instantiations of a parameterized type are unrelated by the type hierarchy.[1] Cell⟨Integer⟩ is not a subtype of Cell⟨Number⟩, even though Integer is a subtype of Number; this is the right choice because Cell⟨Integer⟩ does not support the `set(Number)` operation. Cell⟨Number⟩ is not a subtype of Cell⟨Integer⟩; this is the right choice because Cell⟨Number⟩ does not support the `Integer get()` operation.

Raw types. Great effort was expended in the design of JSR-14 to ensure maximum compatibility with existing non-generic code, in both directions. As a result, the type system of JSR-14 subsumes that of non-generic Java, with the un-parameterized types of generic classes such as `Cell` being known as *raw* types, and raw types being supertypes of parameterized versions. (A raw type can

[1] By contrast, Java arrays use covariant subtyping: Integer[] is a subtype of Number[] because Integer is a subtype of Number. This violates the substitutability principle of subtyping [28]. In order to preserve type safety, the JVM implementation must perform a run-time check at array stores. Since JSR-14 uses homogeneous translation by type erasure (see below), run-time checking for generics is impossible, and soundness must be ensured statically through invariant subtyping.

```
// A Cell is a container that contains exactly one item, of type V.
class Cell<V extends Object> {
    V value;
    Cell(V value) { set(value); }
    void set(V value) { this.value = value; }
    V get() { return value; }
    <U extends V> void replaceValue(Cell<U> that) {
        this.value = that.value;
    }
}

// A Pair has a first and a second element, possibly of different types.
class Pair<F, S> extends Cell<F> {
    S second;
    Pair(F first, S second) { super(first); this.second = second; }
}
```

Figure 2: Example generic library code: definitions of `Cell` and `Pair`. Access modifiers are omitted for brevity throughout.

```
static void example() {
  Cell c1 = new Cell(new Float(0.0));
  Cell c2 = new Cell(c1);
  Cell c3 = (Cell) c2.get();
  Float f = (Float) c3.get();
  Object o = Boolean.TRUE;
  Pair p =
      new Pair(f, o);

  Cell c4 = new Cell(new Integer(0));
  c4.replaceValue(c1);

  displayValue(c1);
  displayValue(c2);

  setPairFirst(p);

  displayNumberValue(p);
  displayNumberValue(c4);

  Boolean b = (Boolean) p.second;
}
static void setPairFirst(Pair p2) {
  p2.value = new Integer(1);
}
static void displayNumberValue(Cell c5) {
  Number n = (Number) c5.get();
  System.out.println(n.intValue());
}
static void displayValue(Cell c6) {
  System.out.println(c6.get());
}
```

Figure 3: Example non-generic client code that uses the library code of Figure 2. The code illustrates a number of features of JSR-14, but it does not compute a meaningful result.

```
static void example() {
  Cell<Float> c1 = new Cell<Float>(new Float(0.0));
  Cell<Cell<Float>> c2 = new Cell<Cell<Float>>(c1);
  Cell<Float> c3 = (Cell) c2.get();
  Float f = (Float) c3.get();
  Boolean o = Boolean.TRUE;
  Pair<Number,Boolean> p =
      new Pair<Number,Boolean>(f, o);

  Cell<Number> c4 = new Cell<Number>(new Integer(0));
  c4.replaceValue(c1);

  displayValue(c1);
  displayValue(c2);

  setPairFirst(p);

  displayNumberValue(p);
  displayNumberValue(c4);

  Boolean b = (Boolean) p.second;
}
static void setPairFirst(Pair<Number,Boolean> p2) {
  p2.value = new Integer(1);
}
static void displayNumberValue(Cell<Number> c5) {
  Number n = (Number) c5.get();
  System.out.println(n.intValue());
}
static void displayValue(Cell c6) {
  System.out.println(c6.get());
}
```

Figure 4: Example client code of Figure 3, after being automatically updated to use generic types. Changed declarations are underlined. Eliminated casts are struck through.

be considered equivalent to a type instantiated with a bounded existential type, e.g., $\mathsf{Cell}\langle \exists \mathsf{x} \,.\, \mathsf{x} \leq \mathsf{Object}\rangle$, because clients using a raw type expect *some* instantiation of the corresponding generic class, but have no information as to what it is [25].)

Unchecked operations. In standard Java, the type system ensures that the type of an expression is a conservative approximation to the kind of objects that may flow to that expression at run time. However, in JSR-14, it is possible to construct programs in which this is not the case, since raw types create a loophole in the soundness of the type system.

Calls to methods and accesses of fields whose type refers to the type variable T of a raw type are *unchecked*, meaning that they may violate the invariants maintained by the non-raw types, resulting in a class cast exception being thrown at run time. The compiler issues a warning when it compiles such operations. (The operations are legal in non-generic Java, and all the same hazards apply, except that the compiler issues no warnings. Use of raw types is no less safe than the original code, though it is less safe than use of non-raw parameterized types.) For example, coercion is permitted from a raw type to any instantiation of it, and vice versa. As another example, one may safely call the method `Vector.size()` on an expression of raw type, since it simply returns an int. On the other hand, a call to `Vector.add(x)` on an expression of type raw Vector would be unchecked, because there may exist an alias to the same object whose declared type is Vector⟨Y⟩, where the type of `x` is not a subtype of Y. Subsequent operations on the object through the alias may then fail due to a type error. It is the programmer's responsibility to ensure that all unchecked operations are in fact safe.

Type erasure. The type rules of JSR-14 suggest the implementation strategy of *type erasure*, in which after the parameterized types have been checked, they are *erased* by the compiler (which inserts casts as necessary), yielding the type that would have been specified in the original non-generic code. For example, the erasure of method `Cell.set(V)` is `Cell.set(Object)`.

Homogeneous translation. Implementation by type erasure implies a *homogeneous* translation. A single class file contains the implementation for every instantiation of the generic class it defines, and the execution behavior is identical to that of the same program written without the use of generic types. Parametric type information is not available at run time, so one cannot query the parameterized type of an object using `instanceof` or reflection, nor can the Java Virtual Machine (JVM) check for type violations at run time as it does with accesses to the built-in array classes. Homogeneous translation is in contrast with approaches that change the JVM [31], and with C++ [40, 41] and other languages [9] in which different code is generated for each instantiation.

2.2.1 Versions of JSR-14

JSR-14 [27] was inspired by GJ (Generic Java) [6, 5]. Different versions of JSR-14 have introduced and eliminated a variety of features related to parametric polymorphism. Our work uses the version of JSR-14 implemented by version 1.3 of the early-access JSR-14 compiler[2]. This particular version proved longer-lived and more stable than other versions, and it is quite similar to the latest proposal (as of July 2004), implemented by Java 1.5 Beta 2.

Java 1.5 Beta 2 has one substantive difference from JSR-14-1.3: Java 1.5 Beta 2's type system is enriched by *wildcard* types [45] such as Vector⟨? extends Number⟩, which represents the set of Vector types whose elements are instances of Number, and Vector⟨? super Integer⟩, which represents the set of Vector types into which an Integer may be stored. Like raw types, wildcard types are effectively parameterized types whose arguments are bounded existential types, but wildcard types generalize this idea, allowing the bounds to express either subtype or supertype constraints [25, 26]. Wildcard types obviate some (though not all) uses of raw types. Wildcard types will improve the precision of our analysis by permitting closer least upper bounds to be computed for some sets of types; see Section 6.5.1 for further detail. This will benefit both the union elimination (Section 6.5) and constraint resolution (Section 6.4) components of our algorithm.

A second, minor difference is that Java 1.5 Beta 2 forbids array creation expressions for arrays of parameterized types, such as `new Cell<String>[..]`, or type variables, such as `new Cell<T>[..]`, or `new T[..]` where `T` is a type variable. Other constructs, such as `List.toArray()`, permit working around this restriction.

[2] `http://java.sun.com/developer/earlyAccess/adding_generics/`

We do not foresee any major obstacles to the adaptation of our type rules, algorithms, and implementation to Java 1.5; such adaptation is important to the usability of our tools once Java 1.5 is finalized.

3. DESIGN PRINCIPLES

We designed our analysis in order to be sound, behavior preserving, compatible, complete, and practical. This section describes each of these inter-related principles, then gives the metric (cast elimination) that we use to choose among multiple solutions that fit the constraints. Finally, we explicitly note the assumptions upon which our approach relies.

3.1 Soundness

The translation must be sound: the result of the analysis must be a type-correct JSR-14 program. Crucially, however, in the presence of unchecked operations, simply satisfying the compiler's type-checker does not ensure type safety.

For instance, there exist type-correct programs in which a variable of type Cell⟨Float⟩ may refer to a Cell containing an Integer. Such typings arise from the unsafe use of unchecked operations.

We require that all unchecked operations in the translated program are *safe*, and are guaranteed not to violate the invariants of any other type declaration. This guarantee cannot be made using only local reasoning, and requires analysis of the whole program.

3.2 Behavior preservation

The translation must preserve the dynamic behavior of the code in all contexts. In particular, it must not throw different exceptions or differ in other observable respects. It must interoperate with existing Java code, and with JVMs, in exactly the same way that the original code did. The translation should also preserve the static structure and the design of the code, and it should not require manual rewriting before or after the analysis.

To help achieve these goals, we require that our analysis changes only type declarations and types at allocation sites; no other modifications are permitted. Changing other program elements could change behavior, cause the code to diverge from its documentation (and from humans' understanding), and degrade its design, leading to difficulties in understanding and maintenance. This implies that inconvenient idioms may not be rewritten, nor may dead code be eliminated. (The type-checker checks dead code, and so should an analysis.) We leave such refactoring to humans or other tools.

Furthermore, we do not permit the erasure of any method signature or field type to change. For instance, a field type or method parameter or return type could change from Cell to Cell⟨String⟩, but not from Object to String. Changing field or method signatures would have far-ranging effects; for instance, method overriding relationships would change, affecting the semantics of clients or subclasses that might not be in the scope of the analysis. If the tool is working under a closed-world assumption, then it may offer the option to change field and method signatures as long as the behavior is preserved. We permit changing the declared types of local variables, so long as the new type is a subtype of the old, because such changes have no externally visible effect.[3]

Finally, we do not permit any changes to the source code of the library or the generic information contained in the compiled

[3] This statement is not strictly true, even though the type of local variables is not captured in the byte codes. The types of locals can affect method overloading resolution and which version of a field (that is re-declared to shadow one in a superclass) is accessed. Therefore, an implementation should ensure that such behavioral changes do not occur.

bytecode of the library. The analysis should not even need library source code, which is often unavailable.

It is straightforward to see that these constraints ensure behavior preservation. The new code differs only in its type erasure and in the types of local variables; neither of these is captured in the bytecodes that run on the JVM, so the bytecodes are identical. (The signature attribute, which records the type parameters, is ignored by the virtual machine.)

3.3 Compatibility

We constrain ourselves to the confines of the JSR-14 language rather than selecting or inventing a new language that permits easier inference or makes different tradeoffs. (For example, some other designs are arguably simpler, more powerful, or more expressive, but they lack JSR-14's integration with existing Java programs and virtual machines.) Invariant parametric subtyping, raw types, and other features of the JSR-14 type system may be inconvenient for an analysis, but ignoring them sheds no light on JSR-14 and is of no direct practical interest to Java programmers. Therefore, we must address the (entire) JSR-14 language, accepting the engineering tradeoffs made by its designers.

3.4 Completeness

We strive to provide a nontrivial translation for all Java code, rather than a special-case solution or a set of heuristics. Java code is written in many styles and paradigms, and relies on many different libraries. The absolute amount of code not covered by a partial solution is likely to be very large.

A few important libraries, such as those distributed with the JDK, are very widely used. Special-case solutions for them may be valuable [42], and such an approach is complementary to ours. However, such an approach is limited by the fact that many substantial programs (two examples are *javac* and *antlr*) define their own container classes rather than using the JDK versions.

Our approach works equally well with non-containers. Many generic classes implement container abstractions, but not all do. For example, class `java.lang.Class`, or the `java.lang.ref` package which uses generics to provide support for typed 'weak' references. Our implementation also uses them for I/O adapters that convert an object of one type to another (say, type T to String), and the C++ Standard Template Library [37] provides additional examples.

3.5 Practicality

Our goal is not just an algorithm for computing type arguments, but also a practical, automated tool that will be of use to Java programmers. For any legal Java program, the tool should output legal JSR-14 code. Furthermore, if it is to be widely useful, it should not rely on any specific compiler, JVM, or programming environment. (On the other hand, integrating it with a programming environment, without relying on that environment, might make it easier to use.)

A practical tool should not require any special manual work for each program or library, and touch-ups of the inputs or results should not be necessary. Equally importantly, as a follow-on to a point made above, special preparation of each library is not acceptable, because library code is often unavailable (for example, it was not provided with the JSR-14 compiler that we are using), because library writers are unlikely to cater to such tools, and because human tweaking is error-prone and tedious.

3.6 Success metric: cast elimination

There are multiple type-correct, behavior-preserving JSR-14 translations of a given Java codebase. Two trivial solutions are as follows. (1) The null translation, using no type arguments. JSR-14 is a superset of Java, so any valid Java program is a valid JSR-14 program in which each type is a JSR-14 raw type. (2) Instantiate every use of a generic type at its upper bounds, and retain all casts that appear in the Java program. For example, each use of Cell would become Cell⟨Object⟩. These trivial solutions reap none of the benefits of parametric polymorphism.

```
interface I {}
class A {}
class B1 extends A implements I {}
class B2 extends A implements I {}
                                    // Three possible typings:
void foo(boolean b) {               //    #1    |    #2    |    #3
  Cell cb1 = new Cell(new B1()); //  Cell<A> | Cell<I> | Cell<B1>
  Cell cb2 = new Cell(new B2()); //  Cell<A> | Cell<I> | Cell<B2>
  Cell c = b ? cb1 : cb2;         //  Cell<A> | Cell<I> | Cell
                                    // Casts eliminated:
  A a = (A)c.get();                //    yes   |   no    |   no
  I i = (I)c.get();                //    no    |   yes   |   no
  B1 b1 = (B1)cb1.get();           //    no    |   no    |   yes
  B2 b2 = (B2)cb2.get();           //    no    |   no    |   yes
}
```

Figure 5: Java code with multiple non-trivial JSR-14 translations.

Figure 5 shows an example fragment of code for which multiple translations are possible. As shown in the figure, three possible typings are

1. `cb1`, `cb2`, and `c` are all typed as Cell⟨A⟩
2. `cb1`, `cb2`, and `c` are all typed as Cell⟨I⟩
3. `cb1` is typed as Cell⟨B1⟩; `cb2` is typed as Cell⟨B2⟩; and `c` is typed as (raw) Cell. In this case `c` cannot be given a non-raw type due to invariant subtyping.

Because the intent of the library and client programmers is unknowable, and because different choices capture different properties about the code and are better for different purposes, there is no one best translation into JSR-14.

As a measure of success, we propose counting the number of casts that can be eliminated by a particular typing. Informally, a cast can be eliminated when removing it does not affect the program's type-correctness. Cast elimination is an important reason programmers might choose to use generic libraries, and the metric measures both reduction in code clutter and the amount of information captured in the generic types. (Casts are used for other purposes than for generic data types — as just two examples, to express invariants known to the application, or to resolve method overloading — so the final JSR-14 program is likely to still contain casts.) If two possible typings eliminate the same number of casts, then we prefer the one that makes less use of raw types. Tools could prioritize removing raw types over removing casts if desired. However, some use of raw types is often required in practice.

In practice, when we have examined analysis results for real-world code, this metric has provided a good match to what we believed a programmer would consider the best result. As an example of the metric, Figure 5 shows that the first two typings remove one cast each; the third removes two casts, leaving `c` as a raw type.

It is not always desirable to choose the most precise possible type for a given declaration, because it may lead to a worse solution globally: precision can often be traded off between declaration sites. In Figure 5, as a result of invariant parametric subtyping, the types of `c`, `cb1`, and `cb2` may all be equal, or `cb1` and `cb2` can have more specific types if `c` has a less specific type. Another situation in which the use of raw types is preferred over the use of non-raw types is illustrated by the method `displayValue` on lines 30–32 of Figure 4. If its parameter were to be made non-raw, the type argument must be Object, due to constraints imposed by the calls at lines 13 and 14. This has many negative ramifications. For example, `c1` and `c2` would have type Cell⟨Object⟩ and `c3` would have raw Cell, and the casts at lines 4 and 5 could not be eliminated.

3.7 Assumptions

In this section we note some assumptions of our approach.

We assume that the original library and client programs conform to the type-checking rules of JSR-14 and Java, respectively. (This is easy to check by running the compilers.)

The client code is Java code containing no type variables or parameters; that is, we do not refine existing JSR-14 types in client code.

We do not introduce new type parameters; for instance, we do not parameterize either classes or methods in client code. (The type parameterization problem is beyond the scope of this paper and appears to be of less practical importance.)

Our analysis is whole-program rather than modular; this is necessary in order to optimize the number of casts removed and to ensure the use of raw types is sound (Section 3.1). Furthermore, we make the closed-world assumption, because we use constraints generated from uses in order to choose declaration types.

4. ALGORITHM SYNOPSIS

Our goal is to select sound type arguments for each use of a generic type anywhere in the program. We divide this task into two parts: allocation type inference and declaration type inference.

Allocation type inference (Section 5) proposes types for each allocation site (use of `new`) in the client code. It does so in three steps. First, it performs a context-sensitive pointer analysis that determines the set of allocation sites to which each expression may refer. Second, for each *use* (method call or field access) of an object of generic type, it unifies the pointer analysis information with the declared type of the use, thereby constraining the possible instantiation types of the relevant allocation sites. Third, it resolves the context-sensitive types used in the analysis into JSR-14 parameterised types. The output of the allocation type inference is a precise but conservative parameterised type for each object allocation site.

For example, in Figure 4, the results of allocation type inference for the three allocations of Cell on lines 2, 3, and 10 are Cell⟨Float⟩, Cell⟨Cell⟨Float⟩⟩, and Cell⟨Number⟩, respectively.

Declaration type inference (Section 6) starts with the allocation type inference's output, and selects types for all uses of parameterized types, including declarations (fields, locals, and method parameters and returns), casts, and allocation sites. At allocation sites, it need not necessarily choose the type proposed by the allocation-site inference (though our current implementation does; see Section 6.4). It operates in two steps. The first step creates a type constraint graph that expresses the requirements of the JSR-14 type system; this graph includes variables (*type unknowns*) that stand for the type arguments at generic instantiations. The second step solves the type constraints, yielding a JSR-14 typing of the entire program. Finally, our tools insert type parameters into the original program's source code.

The allocation type inference is a whole-program analysis; this is required for safety, as explained in Section 3.1, as local analysis cannot provide a guarantee in the presence of unchecked operations. It is context-sensitive, and is potentially more precise than the JSR-14 type system.

The declaration type inference is context-insensitive, and its output is sound with respect to the JSR-14 type system. It can be supplied a whole program, but can also be run on any subpart of a program, in which case it 'frames' the boundaries — constrains the types at the interface so that they will not change — giving possibly inferior results.

τ	::=	C	raw type
	|	$\mathsf{C}\langle\tau_1,\ldots,\tau_n\rangle$	class type
	|	T	type variable
	|	$\mathsf{obj}(\mathsf{C_i})$	type identifier for allocation site C_i
	|	$\{\tau_1,\ldots,\tau_n\}$	union type
	|	Null	the null type

Figure 6: Type grammar for allocation type inference.

5. ALLOCATION TYPE INFERENCE

Allocation type inference determines possible instantiations of type parameters for each allocation site — that is, each use of `new` in the client code. The goal is to soundly infer the most precise type (that is, the least type in the subtype relation) for each allocation site.

Soundness requires that the allocation-site type be consistent with all uses of objects allocated there, no matter where in the program those uses occur. As an example, suppose that the allocation type inference examined only part of the code and decided to convert an instance of `c = new Cell()` into `c = new Cell<Integer>()`. If some unexamined code executed `c.set(new Float(0.0))`, then that code would not type-check against the converted part (or, if it was already compiled or it used a raw type reference, it would simply succeed and cause havoc at run time). Alternatively, the pointer analysis can avoid examining the whole program by making conservative approximations for the unanalyzed code, at the cost of reduced precision. Thus, our allocation type inference could be made modular, by running over a scope smaller than the whole program, but at the cost of unsoundness, reduced precision, or both.

5.1 Defi nitionsand terminology

Figure 6 gives the type grammar used by the allocation type inference. It is based on the grammar of reference types from the JSR-14 type system. For brevity, we omit array types, including primitive array types, although our formalism can be easily extended to accommodate them.

By convention we use C for class names and T for type variables; the distinction between these is clear from the class declarations in a program. In addition to JSR-14 types, the grammar includes three other types used only during the analysis.

Allocation site types: Every allocation site of each generic class C is given a unique label, C_i, and for each such label a unique type identifier $\mathsf{obj}(\mathsf{C_i})$ is created [48]. This type identifier represents the type of all objects created at that allocation site. Some allocation sites within generic library code may be analyzed many times, due to context-sensitivity (see Section 5.4), and for such sites, a new label and type identifier are created each time. All allocations of a non-generic class share the same label.

Union types: A union type represents the least common supertype ('join') of a set of types without computing it immediately. Union types defer the computation of a join until the complete set of types is known, minimizing loss of precision from arbitrary choices when a set of Java types does not have a unique join due to multiple inheritance. The use of union types is not strictly necessary for correctness; we could eliminate them earlier (at each point where they would otherwise be introduced), but at the cost of reduced precision.

The Null type: The Null type denotes the type of the null pointer, and is a subtype of every other type.

5.2 Allocation type inference overview

The allocation type inference consists of three steps: pointer analysis, s-unification, and resolution of parametric types. The output of the allocation-type inference is a parameterized type for each allocation site that conservatively approximates all subsequent uses of the allocated object.

1. Pointer analysis (Section 5.4) abstracts every expression e in the program by a set of allocation-site labels, POINTS-TO(e). The presence of a label C_i in this set indicates that objects created at C_i may flow to e, or, equivalently, that e may point to objects created at C_i. POINTS-TO sets generated by a sound pointer analysis are a conservative over-approximation of all possible executions of the program: the results can indicate that e may point to C_i when this cannot actually occur. A more precise pointer analysis produces a smaller POINTS-TO set.

Many different pointer analysis algorithms exist, differing in precision, complexity, and cost [23]. We use a context-sensitive pointer analysis based on the Cartesian Product Algorithm [1].

2. S-unification (Section 5.5) combines the results of pointer analysis with the declarations of generic library classes in order to generate subtype constraints. Its name comes from its similarity to conventional unification: both generate constraints by structural induction over types. 'S-unification' stands for 'unification with subtyping'. Whereas conventional unification identifies two terms by equating variables with their corresponding subterms, s-unification generates subtype constraints between variables and terms.

At each invocation of a generic library method, one s-unification is performed for the result, if any, and one is performed for each method parameter. Furthermore, for each allocation site of a generic library class, one s-unification is performed for each field of the class.

S-unification is a worklist algorithm. Generic classes can refer to other generic classes (for instance, when inferring nested generic types such as $\mathsf{Cell}\langle\mathsf{Cell}\langle\mathsf{Integer}\rangle\rangle$), so if more information becomes available, previous s-unifications may need to be re-done.

The result of s-unification is a set of constraints on the values of the type variables at each generic class allocation site. For example, $\mathsf{Cell}\langle\mathsf{V}\rangle$ has method `set(V)`. If we determine that for the code `c.set(x)`, POINTS-TO(x) = {obj(String)} and POINTS-TO(c) = {obj(Cell_2)}, then we know that the instantiation of `V` in obj(Cell_2) must allow a String to be assigned to it. In other words, we know that String is a subtype of the instantiation of `V` in obj(Cell_2). We write this as $\mathsf{String} \leq \mathtt{V}_{\mathsf{obj(Cell_2)}}$; see Section 5.5.

The s-unification step is necessary because while pointer analysis can distinguish different instances of a given class (for example, two distinct allocations of Cell), it does not directly tell us the type arguments of the parameterized types: it doesn't know that one is a $\mathsf{Cell}\langle\mathsf{Number}\rangle$ while another is a $\mathsf{Cell}\langle\mathsf{Cell}\langle\mathsf{Number}\rangle\rangle$. The s-unification step examines the uses of those Cells, such as calls to `set`, to determine the instantiation of their type variables.

3. Resolution of parametric types (Section 5.6). For each parameter of every allocation site of a generic class, the s-unification algorithm infers a set of subtype constraints. Taken together, each set can be considered a specification of the instantiation type of one type-parameter as a union type. For example, in Figure 9, obj(Cell_{10}) has two constraints, $\mathsf{Integer} \leq \mathtt{V}_{\mathsf{obj(Cell_{10})}}$ and $\mathsf{Float} \leq \mathtt{V}_{\mathsf{obj(Cell_{10})}}$; equivalently, we say that obj(Cell_{10}) has the union type {Integer, Float}.

If the program being analyzed uses generic types in a nested fashion, such as $\mathsf{Cell}\langle\mathsf{Cell}\langle\mathsf{Float}\rangle\rangle$,[4] then the union types may refer to other allocation types rather than classes. In this case, the types must be resolved to refer to classes. See .

Local	POINTS-TO set
c1	{obj(Cell_2)}
c2	{obj(Cell_3)}
c3	{obj(Cell_2)}
c4	{obj(Cell_{10})}
f	{obj(Float)}

Figure 7: POINTS-TO sets for local variables in the example of Figure 8.

5.3 Example

We illustrate the algorithm with a code fragment from Figure 3:

```
2  Cell c1 = new Cell2(new Float(0.0));
3  Cell c2 = new Cell3(c1);
4  Cell c3 = (Cell) c2.get();
5  Float f = (Float) c3.get();

10 Cell c4 = new Cell10(new Integer(0));
11 c4.replaceValue(c1);
```

The allocation sites at lines 2, 3, and 10 are labeled $\mathtt{Cell}_2$, $\mathtt{Cell}_3$, and $\mathtt{Cell}_{10}$, and their types are obj(Cell_2), obj(Cell_3), and obj(Cell_{10}). obj(Float) and obj(Integer) are not numbered: Float and Integer are not generic classes, so all of their instances are considered identical.

Figures 7–9 demonstrate the operation of the allocation type inference algorithm.

The first step is pointer analysis. Figure 7 shows the POINTS-TO sets (the output of the pointer analysis) for local variables, and Figure 8 shows the POINTS-TO sets of other expressions of interest. For each expression, the result of the pointer analysis is the set of allocation sites that it may point to at run-time. In this example, only $\mathtt{Cell}_{10}$`.value` points to more than a single site.

The second step is s-unification, which is performed for each generic class field, method call result, and method call parameter. The S-unifications column of Figure 8 shows the s-unifications (calls to the S-UNIFY procedure), and the resulting inferences about the instantiations of type variables. Informally, S-UNIFY(*context*, *lhs*, *rhs*) means 'within the context of allocation site *context*, constrain the free type variables in *lhs* so that *rhs* ≤ *lhs*'. Section 5.5 discusses s-unification for this example in more detail.

The third step is resolution of the s-unification type constraints. Figure 9 illustrates this process. S-unification produced two different constraints for $\mathtt{V}_{\mathsf{obj(Cell_{10})}}$ — $\mathsf{obj(Integer)} \leq \mathtt{V}$ and $\mathsf{obj(Float)} \leq \mathtt{V}$ — so we represent the type of $\mathtt{V}_{\mathsf{obj(Cell_{10})}}$ by the union type {obj(Float), obj(Integer)}. Union types may be eliminated (Section 6.5) by selecting a most precise JSR-14 type that is a supertype of this union — in this case, it would be $\mathsf{V} \equiv \mathsf{Number}$, resulting in the type $\mathsf{Cell}\langle\mathsf{Number}\rangle$ for obj(Cell_{10}) — but this step is not required as union types may be passed on to the next phase of the algorithm.

5.4 Pointer analysis

Pointer analysis is the problem of soundly approximating what possible allocation sites may have created the object to which an expression refers; thus, it also approximates the possible classes of the expression. This information has many uses in program analysis, for example in static dispatch of virtual methods [10, 11, 3, 44], construction of precise call graphs [14, 22], and static elimination of casts [48].

[4]This use of 'nested' refers to lexical nesting of generic type arguments. It is unrelated to the Java notion of a nested class (class whose declaration occurs within the body of another class or interface) [21].

Line	Expression	POINTS-TO set	S-unifications
2	`new` $\texttt{Cell}_2(\bullet)$	{obj(Float)}	S-UNIFY(obj(Cell_2), V, {obj(Float)}) $\Longrightarrow_{16}$ S-UNIFY(obj(Cell_2), V, obj(Float)) $\Longrightarrow_{29}$ obj(Float) $\leq V_{\text{obj}(\text{Cell}_2)}$
3	`new` $\texttt{Cell}_3(\bullet)$	{obj(Cell_2)}	S-UNIFY(obj(Cell_3), V, {obj(Cell_2)}) $\Longrightarrow_{16}$ S-UNIFY(obj(Cell_3), V, obj(Cell_2)) $\Longrightarrow_{29}$ obj(Cell_2) $\leq V_{\text{obj}(\text{Cell}_3)}$
4	$\texttt{Cell}_3$`.get()`	{obj(Cell_2)}	(same as for `new` $\texttt{Cell}_3(\bullet)$)
5	$\texttt{Cell}_2$`.get()`	{obj(Float)}	(same as for `new` $\texttt{Cell}_2(\bullet)$)
10	`new` $\texttt{Cell}_{10}(\bullet)$	{obj(Integer)}	S-UNIFY(obj(Cell_{10}), V, {obj(Integer)}) $\Longrightarrow_{16}$ S-UNIFY(obj(Cell_{10}), V, obj(Integer)) $\Longrightarrow_{29}$ obj(Integer) $\leq V_{\text{obj}(\text{Cell}_{10})}$
11	$\texttt{Cell}_{10}$`.replaceValue(`$\bullet$`)`	{obj(Cell_2)}	S-UNIFY(obj(Cell_{10}), Cell⟨U⟩, {obj(Cell_2)}) $\Longrightarrow_{16}$ S-UNIFY(obj(Cell_{10}), Cell⟨U⟩, obj(Cell_2)) (*) $\Longrightarrow_{42}$ S-UNIFY(obj(Cell_{10}), Cell⟨U⟩, Cell⟨{obj(Float)}⟩) $\Longrightarrow_{38}$ S-UNIFY(obj(Cell_{10}), U, {obj(Float)}) $\Longrightarrow_{22}$ S-UNIFY(obj(Cell_{10}), V, {obj(Float)}) $\Longrightarrow_{16}$ S-UNIFY(obj(Cell_{10}), V, obj(Float)) $\Longrightarrow_{29}$ obj(Float) $\leq V_{\text{obj}(\text{Cell}_{10})}$
	$\texttt{Cell}_2$`.value`	{obj(Float)}	(same as for `new` $\texttt{Cell}_2(\bullet)$)
	$\texttt{Cell}_3$`.value`	{obj(Cell_2)}	(same as for `new` $\texttt{Cell}_3(\bullet)$)
	$\texttt{Cell}_{10}$`.value`	{obj(Integer), obj(Float)}	S-UNIFY(obj(Cell_{10}), V, {obj(Integer), obj(Float)}) $\Longrightarrow_{16}$ S-UNIFY(obj(Cell_{10}), V, obj(Integer)) $\Longrightarrow_{29}$ obj(Integer) $\leq V_{\text{obj}(\text{Cell}_{10})}$ $\Longrightarrow_{16}$ S-UNIFY(obj(Cell_{10}), V, obj(Float)) $\Longrightarrow_{29}$ obj(Float) $\leq V_{\text{obj}(\text{Cell}_{10})}$

Figure 8: Example of s-unification, for lines 2–5 and 10–11 of Figure 3. The table shows, for each field and method call of a generic class, its POINTS-TO set, and the calls to S-UNIFY issued for it. A bullet • indicates that the POINTS-TO set is for the value of an actual parameter to a method call. A 'snapshot' (see Section 5.5) of obj(Cell_2) is taken where indicated by the asterisk (*). Subscripts on arrows indicate the line number in Figure 10 at which the recursive call appears or the constraint is added.

S-unification constraints	LBOUNDS values	Resolved types	JSR-14 types
obj(Float) $\leq V_{\text{obj}(\text{Cell}_2)}$ obj(Cell_2) $\leq V_{\text{obj}(\text{Cell}_3)}$ obj(Integer) $\leq V_{\text{obj}(\text{Cell}_{10})}$ obj(Float) $\leq V_{\text{obj}(\text{Cell}_{10})}$	$V_{\text{obj}(\text{Cell}_2)}$ = {obj(Float)} $V_{\text{obj}(\text{Cell}_3)}$ = {obj(Cell_2)} $V_{\text{obj}(\text{Cell}_{10})}$ = {obj(Integer), obj(Float)}	obj(Cell_2) = Cell⟨Float⟩ obj(Cell_3) = Cell⟨Cell⟨Float⟩⟩ obj(Cell_{10}) = Cell⟨{Integer, Float}⟩	obj(Cell_2) = Cell⟨Float⟩ obj(Cell_3) = Cell⟨Cell⟨Float⟩⟩ obj(Cell_{10}) = Cell⟨Number⟩

Figure 9: Resolution of s-unification constraints. The first column shows the constraints arising from the S-UNIFY calls of Figure 8. The second column shows the equivalent union types; note that obj(Cell_3) depends on the type of obj(Cell_2). The third column shows the final allocation-site types after type resolution. The fourth column shows what the result would be, if union types were eliminated at this stage.

To achieve greater precision, a context-sensitive analysis may repeatedly examine the effect of a statement, or the value of a variable, in differing contexts. Our pointer analysis employs both kinds of context sensitivity, *call* and *data*. This permits distinguishing among different instances of a single generic class: one `new Cell()` expression may create Cell⟨Integer⟩, while another creates Cell⟨Float⟩. By '$\texttt{Cell}_i$ creates Cell⟨Integer⟩', we mean that instances of class Cell allocated at $\texttt{Cell}_i$ are used only to contain Integers. Our method applies equally well to generic classes that are not containers.

A *call context-sensitive* pointer analysis may analyze a method more than once depending on where it was called from or what values were passed to it. Each specialized analysis of the same method is called a *contour*, and a *contour selection function* maps from information statically available at the call-site to a contour. The contour selection function may either return an existing contour or create a new one. If the contour is new, the method must be analyzed from scratch. For an existing contour, re-analysis of the method is necessary only if the new use of the contour causes new classes to flow to it; if the re-analysis causes the results to change, then additional contours that depend on the result must be re-analyzed until a fixed point is reached.

Data context-sensitivity concerns the number of separate abstractions of a single variable in the source code. An insensitive algorithm maintains a single abstraction of each field, and is unable to distinguish between the values of corresponding fields in different instances of the same class. In contrast, a data context-sensitive scheme models fields of class C separately for each distinctly-labeled allocation-site of class C. Data context-sensitivity is sometimes called 'field cloning' or the 'creation type scheme' [48]. Limiting either call or data context-sensitivity reduces execution time but may also reduce the precision of the analysis results.

Our technique uses a variant of Agesen's Cartesian Product Algorithm (CPA) [1]. We briefly explain that algorithm, then explain our variation on it.

CPA is a widely-used call-context-sensitive pointer analysis algorithm. CPA uses an n-tuple of allocation-site labels $\langle c_1, \ldots, c_n \rangle$ as the contour key for an n-ary method $f(x_1, \ldots, x_n)$. The key is an element of $C_1 \times \ldots \times C_n$, where each C_i is the set of classes that flow to argument x_i of method f at the call-site being analyzed.

The execution time of CPA is potentially exponential, due to the number of keys — the size of the cross-product of classes flowing to the arguments at a call-site. To enable CPA to scale, it is necessary to limit its context-sensitivity. Typically, this is achieved by imposing a threshold L on the size of each argument set. When more than L classes flow to a particular argument, the contour selection function effectively ignores the contribution of that argument to the cross-product by replacing it with the singleton set $\{\star\}$, where $\star$ is a special marker. Call-sites treated in this way are said to be *megamorphic*. The reduction in precision in this approach is applied to only those call sites at which the threshold is exceeded; at another call-site of the same method, analysis of the same parameter may be fully context-sensitive.

CPA is primarily used for determining which classes flow to each use, so in the explanation of CPA above, the abstract values described were classes. The abstraction in our variant of CPA is allocation site type identifiers, which is more precise since it distinguishes allocations of the same class.

Our variant of CPA limits both call and data context-sensitivity so that they apply only to the generic parts of the program. This policy fits well with our intended application, for it reduces analysis costs while limiting negative impacts on precision.

First, to reduce call sensitivity, our contour selection function makes *all* non-generic method parameter positions megamorphic. More precisely, only those parameter positions (and `this`) whose declared type contains a type variable are analyzed polymorphically. Thus, only generic methods, and methods of generic classes, may be analyzed polymorphically. We do not employ a limit-based megamorphic threshold.

For example, `Cell.set(V)` may be analyzed arbitrarily many times, but a single contour is used for all calls to `PrintStream.println(Object x)`, because neither its `this` nor `x` parameters contains a type variable. Calls to a method `f(Set<T> x, Object y)` would be analyzed context-sensitively with respect to parameter `x`, but not `y`.

A few heavily-used non-generic methods, such as `Object.clone` and `System.arraycopy`, need to be treated context-sensitively. We provide annotations to the analysis to ensure this treatment and prevent a loss of precision. Additional methods can be annotated using the same mechanism to ensure precise treatment as required.

Second, to reduce data sensitivity, we use the generic type information in libraries to limit the application of data context-sensitivity to fields. Only fields of generic classes, whose declared type includes a type variable, are analyzed sensitively. For example, a separate abstraction of field `Cell.value` (declared type: V) is created for each allocation site of a `Cell`, but only a single abstraction of field `PrintStream.textOut` (of type BufferedWriter) is created for the entire program.

Our implementation of the pointer analysis is similar to the framework for context-sensitive constraint-based type inference for objects presented by Wang and Smith [48]. Their framework permits use of different contour-selection functions and data context-sensitivity functions (such as their DCPA [48]); our choices for these functions were explained immediately above. Our implementation adopts their type constraint system and closure rules. The analysis generates a set of initial type constraints from the program, and iteratively applies a set of closure rules to obtain a fixed point solution to them. Once the closure is computed, the - sets can be read off the resulting type-constraint graph.

In summary, pointer analysis discovers the types that flow to the fields and methods of a class, for each allocation site of that class. However, this information alone does not directly give a parameterized type for that allocation site: we must examine the uses of the objects (allocated at the site) in order to determine the type arguments. It is necessary to unify the pointer analysis results for fields and methods with their declared types in order to discover constraints on the instantiation type for the allocation site. The unification process is the topic of the next section.

```
// -        unifies lhs with rhs, in the process constraining, in
//              , the type variables of context so that rhs ≤ lhs.
// context is an allocation site of a generic class C.
// lhs is the type of a JSR-14 declaration appearing within class
//    C, typically containing free type variables of C.
// rhs is a type, typically a union of obj(Ci) types denoting a
//             -  -set; it never contains free type variables.
procedure  -       (context, lhs, rhs)
  if lhs has no free type variables then
    return
  // First, switch based on rhs
  if rhs = Null then
    return
  else if rhs = {τ1,...,τn} then // Union type
    for all τi ∈ rhs do
         -       (context, lhs, τi)
    return
  // Second, switch based on lhs
  if lhs = T then // Type variable
    if T is declared by a generic method then
      for all b ∈          (T) do
           -       (context, b, rhs)
      return
    let tclass := the class that declares T
    if tclass ≠         (context) then
      let lhs′ := instantiation expression of T in         (context)
         -       (context, lhs′, rhs)
      return
              (context, lhs) :=             (context, lhs) ∪ {rhs}
    if              changed then
      for all (c, l, r) ∈             | r = lhs do
           -       (c, l, r)
    return
  else if lhs = C⟨τ1, ..., τn⟩ then // Class type
    if rhs = D⟨τ′1, ..., τ′m⟩ then
      let rhs′ :=         (rhs, C) // rhs′ = C⟨τ″1, ..., τ″n⟩
      for 1 ≤ i ≤ n do
           -       (context, τi, τ″i)
      return
    else if rhs = obj(Ci) then
                 :=          ∪ {(context, lhs, rhs)}
         -       (context, lhs,             (rhs))
      return
    else
      error: This cannot happen
  else // There are no other possibilities for lhs
    error: This cannot happen
```

Figure 10: S-unification algorithm.

5.5 S-unification

S-unification combines the results of pointer analysis with the declarations of generic library classes in order to generate subtype constraints. S-unification has some similarity to the unification used in type inference of ML and other languages. Both are defined by structural induction over types. Conventional unification

- (*expr*) is the pointer-analysis result for *expr*: a union type whose elements are the allocation site type identifiers $\mathsf{obj}(\mathsf{C_i})$ that *expr* may point to.

(*context*, *typevar*) is the (mutable) union type whose elements are the discovered lower-bounds on type variable *typevar* within allocation site type *context*.

$(\mathsf{obj}(\mathsf{C_i})) = \mathsf{C}\langle \mathsf{S_1}, \ldots, \mathsf{S_n}\rangle$
where $\mathsf{S_j} =$ $(\mathsf{obj}(\mathsf{C_i}), T_j)$
and T_j is C's j^{th} type variable.

is a global set of triples $(\mathsf{obj}(\mathsf{C_i}), \tau, \mathsf{obj}(\mathsf{D_j}))$. The presence of a triple $(\mathit{context}, \mathit{lhs}, \mathit{rhs}) \in$ indicates that a call to - with those arguments depended upon the current value of (*rhs*), and that if that value should change, the call should be re-issued.

(T) returns the set of upper bounds of a type variable T.

$(\mathsf{D}\langle\tau_1, \ldots, \tau_n\rangle, \mathsf{C})$ returns the (least) supertype of $\mathsf{D}\langle\tau_1, \ldots, \tau_n\rangle$ whose erasure is C.

$(\mathsf{obj}(\mathsf{C_i})) = \mathsf{C}$ is the class that is constructed at allocation site C_i.

Figure 11: S-unification helper definitions.

identifies two terms by finding a consistent substitution of the variables in each term with the corresponding subterm; the substitution, or unifier, is a set of equalities between variables and subterms. In s-unification, the unifier is a set of *inequalities*, or subtype constraints. S-unification also differs in that it is a worklist algorithm: as new information becomes available, it may be necessary to repeat some s-unifications.

S-unification is performed by the - procedure of Figure 10. It can be thought of as inducing the subtype constraint $\mathit{rhs} \leq \mathit{lhs}$ resulting from the Java assignment '*lhs* = *rhs*;'. The three parameters of the - procedure are as follows. The *context* argument is the type identifier of an allocation site of generic class *C*, whose variables are to be constrained. The *lhs* argument is the declared type of a JSR-14 field or method parameter declaration appearing within class *C*. The *rhs* argument is typically the corresponding - set — that is, a union of allocation site types — for declaration *lhs* inferred by the pointer analysis of Section 5.4. Figure 11 lists several helper definitions used by the s-unification algorithm.

S-unification infers, for each type variable T of each distinct allocation site type $\mathsf{obj}(\mathsf{C_i})$, a set of types, each of which is a lower bound on the instantiation of the type variable; in other words, it infers a union type. When s-unification is complete, this union type captures all the necessary constraints on the instantiation of the type variable.

These lower bounds are denoted (*context*, *typevar*), where *context* is an allocation site type, and *typevar* is a type variable belonging to the class of the allocation. ($\mathsf{V}_{\mathsf{obj}(\mathsf{Cell}_{10})}$ is shorthand for $(\mathsf{obj}(\mathsf{Cell}_{10}), \mathsf{V})$.) All are initialized to the empty union type, and types are added to them as s-unification proceeds.

After the pointer analysis of Section 5.4 is complete, - is called for each field and method defined in the generic classes in the program. Specifically, it is called for each context-sensitive abstraction of a field or method parameter or result.

Our example has three different `Cell` allocation sites, each with a distinct abstraction of field `value`, so - is called once for each. The information in Figure 8 is therefore data context-sensitive. In these calls to - , the *context* argument is the allocation site type, *lhs* is the declared type of the field, and *rhs* is the - set of the field. (See the last three rows of Figure 8.) In contrast, a single abstraction is used for all instances of Float, since it is non-generic (- is not called for non-generic types).

Similarly, there may be many context-sensitive method-call abstractions for a single source-level call site (although in our small example, they are one-to-one). - is called once for each formal parameter and return parameter at each such call. The information in Figure 8 is therefore call context-sensitive. In these calls to - , the *context* argument is the allocation site type of the receiver expression (in our example it is the sole element of - (`this`)), *lhs* is the declared type of the method parameter, and *rhs* is the - set of the argument or result. In contrast, a single abstraction would be maintained for all calls to a non-generic method such as `PrintStream.println` (not shown). See Section 5.4 for more details.

To build some initial intuitions of the workings of the algorithm before showing all details of its operation, we present the steps performed for some expressions of Figure 8.

The Cell constructor's formal parameter type is `V`, and at `new` $\mathtt{Cell}_3$`(•)` on line 3, the actual parameter points to $\mathsf{obj}(\mathsf{Cell}_2)$. Therefore, whatever type is ascribed to $\mathsf{obj}(\mathsf{Cell}_2)$, it must be assignable to (i.e., a subtype of) the type of `V` in $\mathsf{obj}(\mathsf{Cell}_3)$. This requirement is expressed by issuing a call to - $(\mathsf{obj}(\mathsf{Cell}_3), \mathtt{V}, \{\mathsf{obj}(\mathsf{Cell}_2)\})$. When processing $\mathtt{Cell}_{10}$`.value`, unification against a non-trivial union type results in multiple recursive calls to - .

In the second line of the `replaceValue` s-unification call, indicated by the asterisk (*) in Figure 8, - must unify $\mathsf{Cell}\langle\mathsf{U}\rangle$ with $\mathsf{obj}(\mathsf{Cell}_2)$. However, the type of $\mathsf{obj}(\mathsf{Cell}_2)$ is not yet known — the goal of allocation type inference is to determine constraints on the obj types. To permit unification to proceed, - uses, in place of $\mathsf{obj}(\mathsf{Cell}_2)$, a *snapshot*: the type implied by its current constraints. In this case, because the only constraint on Cell_2 is $\mathsf{obj}(\mathsf{Float}) \leq \mathsf{V}_{\mathsf{obj}(\mathsf{Cell}_2)}$, the snapshot is $\mathsf{Cell}\langle\{\mathsf{obj}(\mathsf{Float})\}\rangle$. If subsequent unifications add any new constraints on $\mathsf{obj}(\mathsf{Cell}_2)$, then the snapshot changes and the unification must be re-performed. Reunification is not necessary in our example.

As can be seen from the duplicated entries in the S-unifications column of Figure 8, there is significant redundancy in the `Cell` example. The formal parameter to method `set` (which is not used by this part of the client code), the result of method `get`, and the field `value` are all of declared type `V`. Since the - sets for all three of these will typically be identical, many of the unifications issued will be identical. In this particular case, it would suffice for the algorithm to examine just the `value` field. However, in more complex generic classes (e.g., Vector or HashSet), there may be no single declaration in the class whose - set can be examined to determine the instantiation, and in such cases, the analysis must use information from fields, method parameters, and method results. (Also, this ensures correct results even in the presence of unchecked operations, such as a cast to a type variable T. An approach that assumes that any such cast succeeds may choose incorrect type parameters.)

5.5.1 S-unification algorithm details

This section discusses the - algorithm presented in Figure 10. Readers who are not interested in a justification of the details of the algorithm may skip this section. Line numbers refer to the pseudocode of Figure 10.

The first few cases in the algorithm are straightforward. If there are no free type variables to constrain (lines 9–10), or only the `null` value flows to a type variable (lines 12–13), then no constraints can be inferred. When the *rhs* of a unification is a union type (as for `Cell`$_{10}$`.value` in Figure 8), S-UNIFY descends into the set and unifies against each element in turn (lines 14–17).

Otherwise, *lhs* contains free type variables, so it is either a type variable or a (parameterized) class type. First, consider the case when it is a type variable (lines 19–33).

JSR-14 source code need not explicitly instantiate type variables declared by generic methods, so our algorithm need not track constraints on such variables. Without loss of precision, unifications against type variables declared by a method are replaced by unifications against the method variable's type bound (lines 20–23), which may refer to a class variable. Care must be exercised to prevent infinite recursion in the presence of F-bounded variables such as `T extends Comparable<T>`; for clarity, this is not shown in the algorithm of Figure 10.

The call to `replaceValue` gives a concrete example. In the fourth call to S-UNIFY (see Figure 8), *lhs* is the type variable U. This variable is declared by the generic method

```
<U extends V> replaceValue(Cell<U>)
```

and not by the generic class of *context*, which is `Cell`. Since we cannot meaningfully constrain U in this context, we replace this type variable by its bound, which is V, and S-UNIFY again, eventually obtaining a Float constraint on V.

The type variable may be declared in a different class than *context* — for example, when processing inherited methods and fields, which may refer to type variables declared by a superclass of the receiver. Lines 24–28 handle this case. For example, Pair⟨F, S⟩ inherits field `V value` from class Cell⟨V⟩. It would be meaningless to constrain V in the context of a Pair allocation, since Pair has no type variable V. The instantiation expression of Cell's V in Pair is F. So, a unification in Pair context, whose *lhs* is V, becomes a unification against F. This produces the correct results for arbitrarily complex instantiation expressions in `extends`-clauses.

The last possibility for a type variable is that it is declared by the class being constrained — that is, the class of *context*. In this case, LBOUNDS(obj(Cell$_2$), V) is updated by adding *rhs* to it (line 29). This is the only line in the algorithm that adds a type constraint.

Now, consider the case when *lhs* is a class type (lines 34–43); *rhs* is either a class type or an allocation site type.

If *rhs* is a class type, then corresponding type parameters of *lhs* and *rhs* can be unified (lines 37–38). This is only sensible if the classes of *lhs* and *rhs* are the same, so that their type parameters correspond. The class of *rhs* is widened to satisfy this requirement. In our example, while processing `Cell`$_{10}$`.replaceValue(•)`, the widening is the identity operation since the classes of *lhs* and *rhs* already match: they are both Cell.

The algorithm's final case handles the possibility that *rhs* is an allocation type (lines 40–43). In this case, the allocation type is replaced by a *snapshot*: the type implied by the current set of type constraints on the allocation type.

A snapshot uses the current state of information about a type variable, but this information is subject to change if the variable's LBOUNDS-set grows. If this happens, unifications that depended upon SNAPSHOT information must be recomputed (lines 30–32). Each time an allocation-site type o appears as the *rhs* of a call to S-UNIFY, a SNAPSHOT of it is used, and a triple (*context*, *lhs*, *rhs*) is added to the set REUNIFY $\subseteq (A \times \tau \times A)$, where A is the set of allocation-site types. This set is global (its value is preserved across calls to S-UNIFY), and initially empty. Each triple in REUNIFY is the set of arguments to the call to S-UNIFY in which a SNAPSHOT was used. Whenever the value of LBOUNDS(o) grows, SNAPSHOT(o) becomes stale, so we must again call S-UNIFY(c, l, r), for each triple $(c, l, r) \in$ REUNIFY such that $r = o$. Since the process of s-unification is idempotent with respect to the same argument values, and monotonic with respect to larger *rhs* argument values, this is sound.

5.6 Resolution of parametric types

The result of s-unification is an LBOUNDS union type for each type variable of each generic allocation-site type, where the union elements are allocation-site types. For our example, these are illustrated in the LBOUNDS values column of Figure 9. The step of resolution uses these unions to determine a type for each allocation site; we call this type the resolved type.

For a non-generic allocation site type such as Float, the resolved type is just the type of the class itself. However, one allocation site type can depend on another allocation site type. In particular, the resolved type of a generic allocation depends on other resolved types: obj(Cell$_2$) depends upon obj(Float), and obj(Cell$_3$) depends upon obj(Cell$_2$). Intuitively, if obj(Cell$_3$) 'is a Cell of obj(Cell$_2$)', then we need to know the resolved type of obj(Cell$_2$) before we can give a resolved type to obj(Cell$_3$). To perform resolution, we resolve allocation site types in reverse topological order of resolution dependencies.

For our running example, the resolution dependency graph is:

$$\begin{tikzcd} & \text{obj(Cell 10)} \arrow[r] \arrow[dr] & \text{obj(Integer)} \\ \text{obj(Cell 3)} \arrow[r] & \text{obj(Cell 2)} \arrow[r] & \text{obj(Float)} \end{tikzcd}$$

Additional code (included in our implementation) is required for correct handling of type variable bounds constraints, out-of-bounds types, and to prevent infinite recursion for F-bounded variables such as `T extends Comparable<T>`.

The graph of resolution dependencies is not necessarily acyclic: an expression such as `cell.set(cell)` gives rise to a cycle. A type system with support for recursive types could assign a type such as **fix** $\lambda x.$Cell⟨x⟩. However, JSR-14 has no means of expressing recursive types, so we instantiate all types within a strongly-connected component of the dependency graph as raw types (e.g., raw Cell). We have not yet observed cycles in any real-world programs. The semantic contract of some generic interfaces makes cycles unlikely: for example, the specification of the `Set` interface expressly prohibits a set from containing itself.

6. DECLARATION TYPE INFERENCE

The allocation type inference produces a precise parameterized type for each allocation site of a generic class. The next step, called declaration type inference, uses this information to derive a new type for every variable declaration in the client code, including fields and method parameters.

We note two requirements and one goal for the new types. (1) They must be mutually consistent, so that the resulting program obeys the type rules of the language. (2) They must be sound, so that they embody true statements about the execution of the program; we cannot give a declaration the type Cell⟨Float⟩ if its element may be an Integer. (3) They should to be precise, ascribing the the most specific type possible to each declaration.

The consistency requirement is enforced by type constraints [35], which expresses relationships between the types of program variables and expressions in the form of a collection of monotonic inequalities on the types of those expressions. A solution to such a system of constraints corresponds to a well-typed program.

The soundness requirement is satisfied by using the results of allocation type inference for the type of each allocation site. Since

τ	::=	C	raw type
	|	$\mathsf{C}\langle\tau_1,\ldots,\tau_n\rangle$	class type
	|	T	type variable
	|	X_i	type unknown
	|	$\{\tau_1,\ldots,\tau_n\}$	union type
	|	Null	the null type

Figure 12: Type grammar for declaration type inference.

the behavior of the whole program was examined in order to derive these types, they represent all possible uses. Since the types of allocation sites are sound, all other type declarations are also sound in any consistent solution.

To achieve the goal of precision, we would like to obtain a minimal solution to the system of type constraints, if possible. As we have seen, there may be no unique minimal solution, so we have to content ourselves with solutions composed of local minima.

Sections 6.1 and 6.2 discuss the form of the type constraints. Section 6.3 describes how they are generated from the input program, with an explanation of the need for conditional constraints to properly handle raw types. Finally, Sections 6.4 and 6.5 how the system of type constraints can be solved to obtain a translated program.

Due to space limitations, we illustrate the generation of type constraints for a core subset of the features of the JSR-14 language. The ideas can be extended naturally to support all features of the real language, as in our implementation.

6.1 Type Constraints

A constraint $\alpha_1 \xrightarrow{R} \alpha_2$ is a manifestation of a relationship R between two terms α_1 and α_2. A constraint is *satisfied* if and only if the pair (α_1, α_2) is a member of relation R. If the terms are partially unknown — in other words, they contain variables — then the satisfaction of the constraint depends upon the values of those variables. The problem of constraint solving is therefore to find a set of assignments to the variables that satisfies the complete system of constraints.

Type constraints [35] express relationships between the types of program elements, such as fields and method formal parameters. The relation R is the subtype relation $\leq$, and the grammar of terms is the grammar of types. Type constraint solving assigns to each type constraint variable, a value from the type domain.

For this problem, we use the type grammar τ, shown in Figure 12. This grammar modifies the type grammar of Figure 6 by removing allocation site types and augmenting it with variables, which we call *type unknowns* or *constraint variables*, to distinguish them from the normal usage of 'type variable' in JSR-14 as a synonym for 'type parameter'.

The subtype relation can be viewed as a directed graph whose nodes are types and edges are constraints. The subtype relation is transitive, reflexive, and antisymmetric, so we use the equality notation $\alpha_1 = \alpha_2$ as an abbreviation for a pair of subtype constraints $\alpha_1 \leq \alpha_2$ and $\alpha_2 \leq \alpha_1$.

Our algorithm contains three different constraint systems (described in Sections 5.4, 5.5, and 6), because different parts of the algorithm have different purposes and require different technical machinery. The pointer analysis (Section 5.4) is context-sensitive for precision in computing value flow; we adopt the constraints directly from previous work [48]. By contrast, the results of declaration type inference (Section 6) must satisfy the type-checker, which is context-insensitive, so that constraint system is most naturally context-insensitive. The s-unification constraints (Section 5.5)

Suppose we have declarations:
  `class` $B\langle U_1,\ldots,U_i\rangle$
  `class` $C\langle T_1 \triangleleft S_1,\ldots,T_n \triangleleft S_n\rangle \triangleleft B\langle\tau_1,\ldots,\tau_i\rangle$
where '$\triangleleft$' is an abbreviation for `extends`/`implements`

Then:

$\textsc{widening}(C, C) = \emptyset$
$\textsc{widening}(C, A) = [U_1 := \tau_1,\ldots,U_i := \tau_i] \circ \textsc{widening}(B, A)$
  where $A \neq C$

$\textsc{elaborate}(C) = C\langle X_1,\ldots,X_n\rangle$ (X_i are fresh)
Generates constraints: $X_i \leq S_i\,\theta$
  where $\theta = [T_1 := X_1,\ldots,T_n := X_n]$

$\textsc{receiver}(E.x) = \textsc{declared}([\![E]\!]) \circ \textsc{widening}(C, A)$
  where class A declares member `x`

$\textsc{declared}(C\langle\tau_1,\ldots,\tau_i\rangle) = [T_1 := \tau_1,\ldots,T_i := \tau_i]$

Figure 13: Auxiliary definitions for declaration type inference: WIDENING, which models the widening conversion of parameterized types; ELABORATE, which creates a fresh elaboration of a parametric class with type unknowns; and RECEIVER, which defines the substitution applied to the type of class instance members due to the parameterized type of the receiver.

bridge these two different abstractions, essentially collapsing the context-sensitivity. It might be possible to unify some of these constraint systems, but to do so would complicate them and intertwine conceptually distinct phases of our algorithm.

6.2 Definitions

This section defines terminology used in the description of the declaration type inference.

The term *instantiation* denotes a ground type resulting from the application of a generic type to a set of type arguments. A *type argument* is an actual type parameter used for a generic instantiation. A generic instantiation is either a *parameterized type*, if the generic type is applied to one or more type arguments, or a raw type, if it is applied without explicit type arguments. For example, the parameterized type Cell⟨String⟩ is the generic instantiation resulting from the application of generic type Cell⟨V⟩ to the type argument String.

In our notation, the metavariable C ranges over class names, E ranges over expressions, F ranges over field names, and M ranges over method names. F and M denote the declaration of a specific field or method, including its type and the name of the class in which it is declared. Metavariable X ranges over type unknowns.

We say that a method M in class C *overrides* a method M' in class C' if M and M' have identical names and formal parameter types, and C is a subclass of C'.

$[A := T]$ denotes the substitution of the type variable A with type (or type unknown) T. Substitutions are denoted by the metavariable θ. We denote the empty substitution with $\emptyset$, the composition of two substitutions with $\theta \circ \theta'$, and the application of substitution θ to type T with $T\,\theta$. The result of substitution application is a type (or a type unknown). For example, given class `Pair` of Figure 2, $\mathsf{Pair}\langle\mathsf{F},\mathsf{S}\rangle[\mathsf{F} := X_1, \mathsf{S} := X_2] = \mathsf{Pair}\langle X_1, X_2\rangle$.

Figure 13 defines auxiliary functions.

The WIDENING function defines the widening conversion [21] of (generic) types: it indicates which instantiation of a superclass is a supertype of a given instantiation of a subclass. For example, in the context of types shown in Figures 1 and 2, WIDENING(Pair, Cell) = [V := F], which informally means that Pair is a subtype of Cell when the type variable V of Cell is substituted by F of Pair, so Pair⟨String, Boolean⟩ is a subtype of Cell⟨String⟩.

The ELABORATE function takes a class type C and returns the elaboration of the type — the type obtained by applying C's generic type to a set of fresh type unknowns, one for each type parameter of the class. In addition, this function generates type constraints that ensure the fresh type unknowns are within their bound. For example, ELABORATE(Pair) might return Pair$\langle X_1, X_2\rangle$ and generate constraints $X_1 \leq$ Object and $X_2 \leq$ Object, since both variables F and S are bounded at Object. (We occasionally refer to the type unknowns created during the elaboration of a particular declaration as 'belonging' to that declaration.)

The RECEIVER function returns the receiver-type substitution for an instance member (field or method) expression. This substitution, when applied to the declared type of the member, yields the apparent type of the member through that reference. For example, in Figure 4, variable `p` has type Pair⟨Number, Boolean⟩, so the receiver substitution RECEIVER(`p.value`) is [V := Number]. There are two components to the receiver substitution; the first corresponds to the parameterization of the declaration of `p`, and is [F := Number, S := Boolean]. The second corresponds to the extends clauses between the declared class of `p` (Pair) and the class that declared the member `value` (Cell); in this case, it is [V := F]. The result of RECEIVER is the composition of these substitutions, [V := Number].

The ERASURE function (not shown) returns the erased [6, 27] version of a generic type. For example, ERASURE(Pair⟨String, Boolean⟩) = Pair and ERASURE(*Return*(`Cell.get`)) = ERASURE(V) = Object.

6.3 Creating the type constraint system

Generation of type constraints consists of two steps. First, declarations are elaborated to include type unknowns for all type arguments. Each use of a generic type in the client program, whether in a declaration (e.g., of a field or method parameter), or in an operator (e.g., a cast or `new`), is elaborated with fresh type unknowns standing for type arguments. For example, consider the types in Figure 2 and the statement `Pair p = new Pair(f, o)` on lines 7–8 of Figure 3. The declaration type inference creates four fresh type unknowns $X_1, \ldots, X_4$, so the elaborated code is `Pair<`X_1`,`X_2`> p = new Pair<`X_3`,`X_4`>()`.

Second, the declaration type inference algorithm creates type constraints for various program elements. Some type constraints are unconditionally required by the JSR-14 (and Java) type system, or to ensure behavior preservation; see Section 6.3.1. Other type constraints may be in effect or not, depending on the values given to type unknowns. In particular, declaring a generic instantiation to be raw induces different constraints on the rest of the program than does selecting specific type arguments for the generic instantiation; see Section 6.3.2.

6.3.1 Ordinary type constraints

Figure 14 shows type constraints induced by the key features of JSR-14. To cover the entire language, additional constraints are required for exceptions, arrays, `instanceof` expressions, etc. We omit their presentation here because they are similar to those presented. For a more detailed list of various program features and type constraints for them, see [43].

Also, to ensure certain properties of the translation (i.e., principles presented in Section 3), an additional set of constraints is generated. Informally, we must ensure that the erasure of the program remains unchanged (which places constraints on declared types of method parameters and return types, fields, etc.) and that, in the translated program, the declared types of all program elements are no less specific than in the original program (and in the case of library code the types must remain exactly the same). This approach is similar to that taken in [43] and [13]. These type constraints are straightforward and are omitted here.

Constraint generation is achieved by descent over the syntax of all the method bodies within the client code. Figure 14 defines the metasyntactic function $[\![\cdot]\!]$, pronounced 'type of', which maps from expression syntax to types, generating constraints as a side effect. The figure also defines three other metafunctions, *Field*, *Param*, and *Return*, for the types of fields, method parameters, and results. Terms of the form $[\![E]\!] \triangleq (\ldots)$ are not constraints, but form the definition of $[\![\cdot]\!]$.

As an example, consider line 11 of Figure 3: `c4.replaceValue(c1)`, where `c1`'s declaration, elaborated by the introduction of a type unknown, is Cell$\langle X_1\rangle$ and the declaration of `c4` is elaborated to Cell$\langle X_4\rangle$. Thus, we have that:

- $[\![c1]\!] \triangleq \mathsf{Cell}\langle\mathsf{V}\rangle[\mathsf{V} := X_1]$
- $[\![c4]\!] \triangleq \mathsf{Cell}\langle\mathsf{V}\rangle[\mathsf{V} := X_4]$
- $\theta = [\mathsf{V} := X_4]$
- $\theta_{fresh} = [\mathsf{U} := X_\mathsf{U}]$ (X_U is fresh)

Rules **(10)**–**(11)** give us, respectively:

- $[\![\mathtt{c1}]\!] \leq \mathsf{Cell}\langle\mathsf{U}\rangle[\mathsf{V} := X_4][\mathsf{U} := X_U]$
  i.e., $\mathsf{Cell}\langle X_1\rangle \leq \mathsf{Cell}\langle X_U\rangle$
- $X_\mathsf{U} \leq X_4$

6.3.2 Guarded type constraints

Generic instantiations are of two kinds: parameterized types and raw types. For parameterized types, the generated type constraints represent type arguments by a type unknown.

For raw types, there is no X for which raw Cell is a Cell$\langle X\rangle$; constraints that try to refer to this X are meaningless. Type constraints are invalid if they refer to type unknowns arising from an elaboration of a generic declaration that is later assigned a raw type. In that case, a different set of constraints is required, in which the types that previously referred to the 'killed' type unknown are now replaced by their ERASURE.

For example, consider the following code:

```
void foo(Cell c) {
   x = c.get();
   c.set("foo");
}
```

If the declaration of `c` is parameterized (say, Cell$\langle X_1\rangle$), then the constraint $X_1 \leq [\![\mathtt{x}]\!]$ must be satisfied (rules **(1)** and **(9)** in Figure 14). On the other hand, if the declaration is raw, then the constraint ERASURE(*Return*(`Cell.get`)) = Object $\leq [\![\mathtt{x}]\!]$ must be satisfied. Similar constraints arise from the call to `set`: if the declaration is parameterized, then $[\![\mathtt{"foo"}]\!] \leq X_1$; otherwise, $[\![\mathtt{"foo"}]\!] \leq$ ERASURE(*Param*(`set`, 1)) = Object.

Each method invocation (or field reference) on an object whose declaration is a generic instantiation gives rise to two alternative sets of conditional constraints. Any constraint that references a type unknown must be predicated upon the 'parameterizedness' of the type of the receiver expression; we call such expressions *guard expressions*. (Actually, our implementation uses a representation in which all temporaries are explicit, so we call them *guard variables*.) When the type of a guard variable is raw, the alternative constraint after ERASURE is used instead, so the killed type unknown is no longer mentioned. For example, the guarded type constraints created for the second line in the example above are $[\![X_1]\!] \leq_{\mathtt{c}} [\![\mathtt{x}]\!]$ (`c` is the guarding variable), which is interpreted only if $[\![\mathtt{c}]\!]$ is non-raw, and Object $\leq_{\overline{\mathtt{c}}} [\![\mathtt{x}]\!]$ (the left-hand side is erased), which is interpreted only if $[\![\mathtt{c}]\!]$ is raw. Depending on `c`, exactly one of these two constraints is interpreted, and the other is ignored.

program construct	implied type constraint(s)	
statement $E_1 := E_2$;	$[\![E_2]\!] \le [\![E_1]\!]$	**(1)**
statement `return` E; (in method M)	$[\![E]\!] \le \mathit{Return}(M)$	**(2)**
expression `this` (in class B)	$[\![\texttt{this}]\!] \triangleq B$	**(3)**
expression `null`	$[\![\texttt{null}]\!] \triangleq \mathsf{Null}$	**(4)**
expression x_i (in method M)	$[\![x_i]\!] \triangleq \mathit{Param}(M,\ i)$	**(5)**
expression `new` $B\langle\tau_1, \ldots, \tau_k\rangle$ B has type variables $\langle T_1 \triangleleft S_1, \ldots, T_k \triangleleft S_k\rangle$ $\theta = [T_1 := \tau_1, \ldots, T_k := \tau_k]$	$[\![\texttt{new}\ B\langle\tau_1, \ldots, \tau_k\rangle]\!] \triangleq B\langle\tau_1, \ldots, \tau_k\rangle$ $\tau_j \le S_j\ \theta$	**(6)** **(7)**
expression $E.f$ (field F of class B) $\theta = \textsc{receiver}(E.f)$	$[\![E.f]\!] \triangleq \mathit{Field}(F)\ \theta$	**(8)**
expression $E.m(E_1, \ldots, E_n)$ (method M of class B) $\theta = \textsc{receiver}(E.m)$ M has type variables $\langle T_1 \triangleleft S_1, \ldots, T_k \triangleleft S_k\rangle$ $\theta_{fresh} = [T_1 := X_1, \ldots, T_k := X_k]$ (fresh X_i)	$[\![E.m(E_1, \ldots, E_n)]\!] \triangleq \mathit{Return}(M)\ \theta\ \theta_{fresh}$ $[\![E_i]\!] \le \mathit{Param}(M,\ i)\ \theta\ \theta_{fresh}$ $T_j\ \theta_{fresh} \le S_j\ \theta\ \theta_{fresh}$	**(9)** **(10)** **(11)**
method M overrides method M'	$\mathit{Param}(M',\ i) = \mathit{Param}(M,\ i)$ $\mathit{Return}(M) \le \mathit{Return}(M')$	**(12)** **(13)**
method M is defined in library code as: $\langle T_1 \triangleleft S_1, \ldots, T_n \triangleleft S_n\rangle\ \tau\ M(\tau_1\ x_1, \ldots, \tau_n\ x_n)$	$\mathit{Return}(M) \triangleq \tau$ $\mathit{Param}(M,\ i) \triangleq \tau_i$	**(14)** **(15)**
method M is defined in client code as: $\tau\ M(\tau_1\ x_1, \ldots, \tau_n\ x_n)$	$\mathit{Return}(M) \triangleq \textsc{elaborate}(\tau)$ $\mathit{Param}(M,\ i) \triangleq \textsc{elaborate}(\tau_i)$	**(16)** **(17)**
field F is defined in library code as: $\tau\ F$	$\mathit{Field}(F) \triangleq \tau$	**(18)**
field F is defined in client code as: $\tau\ F$	$\mathit{Field}(F) \triangleq \textsc{elaborate}(\tau)$	**(19)**

Figure 14: Type constraints for key features of JSR-14. The type for an expression E or a metasyntactic expression such as $\mathit{Field}(F)$ is defined using the notation $[\![E]\!] \triangleq (\ldots)$. The generation of constraints is explained in Section 6.3.1. The three sections of the table show the constraints generated for statements, expressions, and declarations, respectively.

6.3.3 Allocation Types

For soundness, the types of allocation sites must be consistent with the types inferred by the allocation type inference of Section 5. The most straightforward way to incorporate the results of allocation type inference into the constraint system is simply to define the types of each generic allocation site (as used in rule **(6)**) to be exactly the type inferred for it.

This is simple and easy to implement (and is what our implementation does). It is, though, somewhat overconstrained beyond what is necessary for correctness. A slightly more flexible approach would be to instantiate the `new` expression with a set of fresh type unknowns, and to constrain each of these unknowns to be a subtype of the corresponding parameter type from the inferred type. This approach permits choosing a less specific assignment for a type unknown, which may be desirable, as illustrated by Figure 5.

For example, allocation type inference reports the type $\mathsf{Pair}\langle\mathsf{Number}, \mathsf{Boolean}\rangle$ for the allocation on line 8 of Figure 4. The first approach would simply make this the type of the `new` expression. The second approach would instead make the type of the expression $\mathsf{Pair}\langle X_1, X_2\rangle$, where X_1 and X_2 are fresh type unknowns constrained in the following way: $\mathsf{Number} \le X_1$, $\mathsf{Boolean} \le X_2$.

6.4 Solving the type constraints

The algorithm of Section 6.3 creates type unknowns for each type argument, and creates (ordinary and guarded) type constraints that relate the type unknowns to one another and to types of other program elements, such as fields, method parameters, etc. The final type constraint graph expresses the type rules of JSR-14, plus our additional constraints created for behavior preservation. Any solution to the constraint graph (i.e., assignment of types to constraint variables) therefore represents a well-typed and semantically equivalent translation of the program.

Conceptually, solving the constraints is simple: for each constraint variable in turn, assign a type that satisfies its current constraints. If this choice leads to a contradiction (i.e., there is no satisfying assignment to the remaining constraint variables), then choose a different type for the constraint variable. If all choices for this constraint variable lead to a contradiction, then backtrack and make a different choice for some previously-assigned constraint variable. Because valid typings always exist (Section 3.7), the process is guaranteed to terminate.[5] In principle, the space of type assignments could be exhaustively searched to find the best typing (that eliminates the largest number of casts, per Section 3.6).

This section outlines one practical algorithm for finding a solution to the type constraints; it is based upon a backtracking search, but attempts to reduce the degree of backtracking to a practical level. We have implemented this technique, and it performs well in practice. See Section 8 for the results.

The algorithm constructs a graph, initially containing edges only for the unconditional constraints. The algorithm iterates over all the guard variables in order, trying, for each guard variable g, first to find a solution in which g's type is parameterized (non-raw), and if that fails, to find a solution in which g has a raw type. If no solution can be found due to a contradiction, such as a graph edge whose head is a proper subtype of its tail, or an attempt to assign two unequal values to the same type unknown, then a previously-made decision must be to blame, and the algorithm backtracks.

Each time it begins a search rooted at a (tentative) decision on the type for a particular guard, the algorithm adds to the graph all of the conditional edges predicated upon that guard decision, whether parameterized or raw. Backtracking removes these edges.

As the edges are added, several closure rules are applied. For

[5]Strictly speaking, the set of possible types is infinite, so it cannot be enumerated. However, it is rare to find completely unconstrained type unknowns, and in any case, a k-limited subset of the Herbrand universe of types is enumerable.

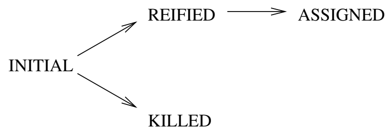

Figure 15: States in the declaration type inference algorithm for each type unknown. At each step, every type unknown is associated with a state: initially the INITIAL state, and at completion, either the ASSIGNED or KILLED state. Edges indicate the permitted transitions between states; only during backtracking is a previous state restored.

example, if the graph contains a path from Cell⟨X⟩ to Cell⟨Y⟩, then the interpretation of this path is Cell⟨X⟩ $\leq$ Cell⟨Y⟩, and by the rules of invariant parametric subtyping, this implies $X = Y$. This causes the addition of two new constraints, $X \leq Y$ and $Y \leq X$. This process is iterated until no further closure rules are applicable. The other closure rules are omitted for brevity.

Once the conditional edges have been added, if the search is trying to infer a parameterized type for guard variable g, then for each unknown X_u belonging to g, the algorithm computes the union of the types that reach X through paths in the graph. This is the set of lower bounds on X_u, and the least upper bound of this union is the type that will be assigned to X_u.

6.4.1 Dependency graph

This section describes how to order the guard variables so as to minimize the backtracking required. This strategy nearly or completely eliminates backtracking in every case we have observed.

We create a dependency graph that indicates all nodes whose assignment *might* affect a node, under any type assignment. The set of nodes in this graph is the same as of the type constraint graph. The set of edges consists of every ordinary edge (from the type constraint graph of Section 6.3.1), every guarded edge (from Section 6.3.2), and, for every guarded edge, an edge from the type of the guard to the head.

In the absence of cycles in the dependency graph, no backtracking is required: the nodes can be visited and their types assigned in the topological order. If the dependency graph has cycles, then backtracking (undoing decisions and their consequences) may be required, but only within a strongly connected component. As a heuristic, within a strongly connected component, we decide any nodes that are guards for some constraints first, because such choices are likely to have the largest impact.

6.4.2 Deciding guards, assigning types

In the declaration type inference algorithm, each type unknown is in one of four states, illustrated in Figure 15. Each type unknown starts in the INITIAL state (or is reset to it via backtracking), which means that it has not yet been considered by the algorithm. A type unknown is in the KILLED state if the guard variable to which it belongs has has been assigned a raw type. The REIFIED state indicates that the algorithm decided to give a parameterized type to the guard variable to which the type unknown belongs, but that the choice of which type to assign it has not yet been made. As soon as a type unknown becomes KILLED or REIFIED, the algorithm adds the relevant conditional edges.

Finally, the ASSIGNED state means that the type is parameterized, and the type arguments have been decided upon. (The type arguments themselves are indicated by a separate table of assignments.) When the algorithm finishes, every type unknown is in the ASSIGNED or KILLED state.

We use the term *decide* for the process of moving a type unknown from the INITIAL state to one of the other states. All the type unknowns belonging to the same guard variable are decided simultaneously.

We distinguish between REIFIED and ASSIGNED to permit deferring the choice of assigned type. Unconstrained type unknowns remain in the REIFIED state until a constraint is added. This prevents premature assignment from causing unnecessary contradictions and backtracking, and yields more precise results.

Section 6.5 presents a join algorithm that determines the least upper bound of a set of JSR-14 types. The solving algorithm uses that procedure extended to handle REIFIED type unknowns. The algorithm treats REIFIED type unknowns as a free choice, so long as that choice is used consistently. This is best illustrated with an example:

REIFIED-JOIN({Pair⟨Number, X_1⟩, Pair⟨X_2, Boolean⟩})
= Pair⟨Number, Boolean⟩
REIFIED-JOIN({Pair⟨Number, X_3⟩, Pair⟨X_3, Boolean⟩}) = Pair

The function REIFIED-JOIN can unify the reified type unknowns with other types to achieve a more precise result. In the first example, it successfully assigns types to X_1 and X_2.

Allocation type inference returns Null as the element type of an empty container; leaving the type unknown standing for the type argument fully unconstrained (Null $\leq X$ is a vacuous constraint). The declaration type inference algorithm can select a non-null type for the element based upon other constraints. For example, if an allocation of an empty cell only flows to a variable of type Cell⟨String⟩, then we can assign Cell⟨String⟩ to the empty cell also.

Any REIFIED unknowns remaining when all the guards have been decided can be assigned a type arbitrarily; our implementation chooses the upper bound on the unknown, typically Object.

6.5 Join algorithm

Union types are converted into JSR-14 types that represent their least common supertype (or *join*) by the following procedure.

Consider a union type u as a set of types. For each non-Null element $t \in u$, compute the set of all its supertypes, including itself. The set of common supertypes is the intersection of these sets.

$$\textit{common supertypes}(u) = \bigcap_{t \in u} \{ s \mid t \leq s \}$$

This set always contains at least Object. At this point, we discard marker interfaces from the set. Marker interfaces — such as Serializable, Cloneable, and RandomAccess — declare no methods, but are used to associate semantic information with classes that can be queried with an `instanceof` test. Such types are not useful for declarations because they permit no additional operations beyond what is allowed on Object. Furthermore, they are misleadingly frequent superclasses, which would lead to use of (say) Serializable in many places that Object is preferable.

We also discard the *raw* Comparable type. Even though it is not strictly a marker, this widely-used interface has no useful methods in the case where its instantiation type is not known: calling `compareTo` without specific knowledge of the expected type usually causes an exception to be thrown. Parameterized instantiations of this interface, such as Comparable⟨Integer⟩, are retained.

From the resulting set, we now discard any elements that are a strict supertype of other elements of the set, yielding the set of least common supertypes of u:

$$\textit{least common supertypes}(u) = \{ t \in cs \mid \neg\exists t' \in cs.\ t' < t \}$$
$$\text{where } cs = \text{filtered } \textit{common supertypes}(u)$$

Again, this set is non-empty, and usually, there is just a single item remaining. (Though the `java.util` package makes extensive use of multiple inheritance, least common supertypes are always uniquely defined for these classes. Also, the boxed types such as Integer, Float, etc., have common supertype Number once the rules for marker interfaces are applied.) However, if after application of these rules the set has not been reduced to a single value, the union elimination procedure chooses arbitrarily. This occurred only once in all of our experiments.

The procedure just described is derived directly from the subtyping rules of the JSR-14 specification, and thus implements invariant parametric subtyping. So, for example:

$$\mathsf{Cell}\langle\{\mathsf{Integer}, \mathsf{Float}\}\rangle \xrightarrow{\text{union elim}} \mathsf{Cell}\langle\mathsf{Number}\rangle$$

$$\left\{\, \mathsf{Cell}\langle\mathsf{Integer}\rangle, \mathsf{Cell}\langle\mathsf{Float}\rangle \,\right\} \xrightarrow{\text{union elim}} \text{raw } \mathsf{Cell}$$

6.5.1 Wildcard types

Java 1.5 has not yet been finalized, but it appears that it will include *wildcard types*, which generalize the use of bounded existentials as type arguments. Every parameterized type such as Cell⟨Number⟩ has two corresponding wildcard supertypes, which are written `Cell<? extends Number>` and `Cell<? super Number>` in the proposed syntax.

The syntax `Cell<? extends Number>` denotes the type $\mathsf{Cell}\langle\exists \mathsf{T}.\, \mathsf{T} \leq \mathsf{Number}\rangle$, which is the type of all Cells whose elements are some (unspecified) subtype of Number. It is therefore a supertype of Cell⟨Integer⟩ and Cell⟨Float⟩, but a more specific one than raw Cell: it allows one to `get` elements at type Number, and forbids potentially dangerous calls to `set`, since the required argument type T is unknown.

`Cell<? super Number>` denotes the type $\mathsf{Cell}\langle\exists \mathsf{T}.\, \mathsf{Number} \leq \mathsf{T}\rangle$, whose elements are of some unspecified supertype of Number. It is a supertype of Cell⟨Number⟩ and Cell⟨Object⟩. This type permits one to `set` elements that are instances of Number, but the result type T of `get` is unknown, i.e., Object.

Use of wildcard types may increase the precision our results, as they represent a closer and more appropriate least upper bound than a raw type in many situations. However, the methods of Cell that reference a type variable from both their parameter and result types belong to neither wildcard type, because $\mathsf{Cell}\langle\exists \mathsf{T}.\, \mathsf{T} \leq \mathsf{Number}\rangle$ has only the `get`-like methods while $\mathsf{Cell}\langle\exists \mathsf{T}.\, \mathsf{Number} \leq \mathsf{T}\rangle$ has all the `set`-like ones.

In order to ascribe a wildcard type to a variable declaration, an analysis must solve an additional set of constraints that restrict which members may be accessed through that variable. Investigating this problem would be an interesting direction for future work.

7. IMPLEMENTATION

We have implemented the algorithms described in this paper as a fully-automated translation tool called Jiggetai. Jiggetai's output is a type-correct, behaviorally equivalent JSR-14 version of the original Java program. Figure 16 shows the tool's architecture. This section notes a few salient points of the implementation.

7.1 Program representation

Since the allocation-type inference is a whole-program analysis, and we cannot demand that source be available for pre-compiled libraries, the analysis must be performed on the bytecode (classfile) representation of the program. However, the declaration-type inference is logically a source-level analysis. For uniformity, we implement both analyses at the bytecode level.

The first component of our system is called the *lossless compiler*, which is a modified version of the standard JSR-14 compiler that preserves source-level information by inserting additional tables of

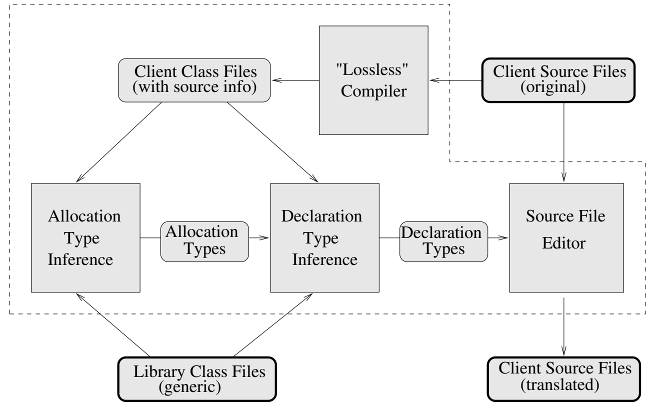

Figure 16: Architecture of the Jiggetai tool for automatic translation from Java to JSR-14. Inputs and outputs are shown with heavy outlines. The dashed border contains Jiggetai itself.

data as *attributes* (comments) in the class file. This information includes: (i) the mappings between source variables and virtual machine registers; (ii) the type of each source variable; and (iii) the type and lexical extent of every declaration or other use of a type-name in the program (locals, fields, methods, casts, allocation sites, extends-clauses, etc.).

In addition, the compiler disables certain optimizations such as dead-code elimination. Dead program statements are still subject to type checking and must be visible to the analysis.

We have extended the Soot [46] class-file analysis package to perform analysis at the source level of abstraction by mapping untyped JVM registers to typed source variables. Our lossless compiler and Soot extensions may be useful to other researchers and tool builders who desire the relative simplicity of the bytecode format while retaining tight integration with source code.

7.2 Allocation type inference

JSR-14 requires that bytecode classfiles that define generic types include a `Signature` attribute that gives information about the type parameters that they define. This attribute is ignored by the JVM, but is required for type-checking client code: Due to JSR-14's type erasure strategy for compilation, it is the only way to know that a classfile represents a generic type, or how many type parameters that class takes.

Generic type information (`Signature` attributes) is missing from all private and anonymous classes in JSR-14-1.3's `java.util` package.

Our analysis interprets classes without a `Signature` attribute as non-generic, so the context sensitivity policy of Section 5.4 analyzes them only once, effectively merging all instances of them together. For example, this effect occurs with (the second type parameter of) `Hashtable` and `HashMap`.

We have implemented two solutions to this problem.

1. Perform more comprehensive retrofitting, which effectively adds `Signature` attributes to all classes, not just named public ones. This approach is sound, regardless of the accuracy of the retrofitting, because the retrofitted types on private library classes are used only as a context-sensitivity hint by the pointer analysis. The retrofitting can be done by hand or via heuristics. For instance, the following heuristic captures the missing information in the JDK libraries almost perfectly: 'If a private or anonymous class extends a generic container class, inherit all generic annotations from the superclass.'

Program	Lines	NCNB	Casts	Gen. casts
antlr	47621	26349	161	50
htmlparser	27640	13062	488	33
JavaCUP	11048	4433	595	472
JLex	7841	4737	71	56
junit	10174	5727	54	26
TelnetD	11190	3976	46	38
v_poker	6316	4703	40	31

Figure 17: Subject programs. *Lines* is the total number of lines of Java code, and *NCNB* is the number of non-comment, non-blank lines. *Casts* is the number of casts in the original Java program, and *Gen. casts* is the number of those that are due to use of raw types.

Program	Gen. casts	Elim	% Elim	Time (sec)
antlr	50	49	98 %	396
htmlparser	33	26	78 %	462
JavaCUP	472	466	99 %	235
JLex	57	56	98 %	35
junit	26	16	62 %	181
TelnetD	38	37	97 %	32
v_poker	31	24	77 %	47

Figure 18: Experimental results. *Gen. casts* is the number of generic casts (resulting from use of raw types) in the original Java program; *Elim* is the number of casts eliminated by our translation to JSR-14, and *% Elim* expresses that number as a percentage.

2. Create a type-correct stub version of the library, and use it in place of the real library when compiling. (This approach is taken by Tip et al. [42].) This approach is labor-intensive and unsound, because the stub method bodies do not necessarily induce the same generic type constraints as the original library would. We implemented it to compare its performance and results with the retrofitting approach.
   Aggregated over all the benchmarks in Figure 18, the use of stubs enabled an additional 0.7% of casts to be eliminated, and execution took 1% longer. The use of stubs roughly halved the running time of the pointer analysis, although the contribution of this phase to the overall runtime was relatively small.

As with all whole-program static analyses, our pointer analysis requires hand-written annotations to summarize the effects of calling native methods and reflective code; without them, soundness cannot be ensured. Currently, we use very naive and conservative annotations for such methods; none of our benchmarks makes significant use of them.

8. EXPERIMENTS

In order to evaluate our analyses and tools, we ran our implementation over the programs listed in Figure 17. The programs are as follows: *antlr* is a scanner/parser generator toolkit[6]; *htmlparser* is a library of parsing routines for HTML[7]. *JavaCUP* is an LALR parser generator[8]; *JLex* is a lexical analyzer generator[9]; *junit* is a unit-testing framework[10]; *TelnetD* is a Telnet daemon[11]; *v_poker* is a video poker game[12]. Figure 17 gives their sizes. The notable number of generic casts in the *JavaCUP* parser generator is due to its parser, which is implemented by a machine-generated source file that makes heavy use of a `Stack` of grammar symbols. Figure 18 shows the results of our experiments.

As our library, we used all generic library classes from package `java.util`, as shipped with the JSR-14-1.3 compiler. This package contains 166 classes, of which 37 are non-generic, 30 are generic top-level classes, and 99 are generic inner classes.

As noted in Section 3.6, casts are used for other purposes than for downcasting elements retrieved from generic classes, so even a perfect translation would not eliminate all casts from the program. We counted the number of generic casts by hand, determining for each cast whether or not it was statically safe, based on human inspection of the values stored into each generic container. (For four of the benchmarks, we performed a complete manual generic translation and counted the number of casts eliminated.)

We executed our tools within Sun's 1.4.1 HotSpot Client JVM with a maximum heap size of 200 MB, running under Linux kernel 2.4 on a 3GHz Pentium 4. Our unoptimized implementation took no more than 8 minutes to translate any program.

The execution time of the larger benchmarks was overwhelmingly dominated by the naive implementation of the resolution algorithm of Section 6.4. We believe that the running time of this phase could be brought down to a small number of seconds, enabling applications based upon interactive refinement of the solution.

8.1 Evaluation

For most of the benchmarks Jiggetai eliminated over 95% of the generic casts. For the other programs, a few specific causes can be identified.

Conservative `extends` parameterization. Whenever the analysis encounters a client class that extends a generic library class, the `extends` clause is parameterized very conservatively, with each type variable instantiated at its erasure. For example, the declaration `class PersonList extends List` is translated to `extends List<Object>`, even if the elements of PersonList are always of class Person.

Without this conservative assumption, extends-clause information would be only partial during analysis, but our algorithm requires it to be complete. This assumption was responsible for the 7 generic casts remaining in *v_poker*.

Missing `clone` covariance. The declared result type of the `clone` method in existing Java code is Object, even though `clone` always returns an instance of the same class as its receiver. JSR-14 allows one to specify *covariant result types* that capture this fact, so for example, the `clone` method of HashSet⟨T⟩ could be declared `HashSet<T> clone()`. Nonetheless, the `Set` interface, via which instances of `HashSet` may be frequently manipulated, does not covariantly specialize `clone`, since it does not require that its instances be cloneable.[13] Therefore, type information is lost during calls to `Set.clone`.

This is the reason for the low score obtained for *junit*. We repeated the experiment after replacing `(C) c.clone()` with just c, and the score went up to 100%. This suggests that type constraint generation for the `clone` method should be handled with a covariant special case.

'Filter' idiom. One particular pathological case, which we have named the *filter idiom*, is typified by the following code:

[6] `http://www.antlr.org/`
[7] `http://htmlparser.sourceforge.net/`
[8] `http://www.cs.princeton.edu/~appel/modern/java/CUP/`
[9] `http://www.cs.princeton.edu/~appel/modern/java/JLex/`
[10] `http://www.junit.org/`
[11] `http://telnetd.sourceforge.net/`
[12] `http://vpoker.sourceforge.net/`
[13] Or, for compatibility, `clone` may not have been covariantly specialized, as is the case for `HashSet`.

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

Here, `strings` contains only instances of String, but the call to `add(o)` generates a constraint that the element type is Object. If the programmer had explicitly cast `o` to String before the call to `add`, the desired type List⟨String⟩ would have been inferred. But in non-generic Java, there is no need for such a cast, because List will accept values of any type, so it was omitted[14].

The filter idiom is heavily used by the *htmlparser* benchmark. This problem could be addressed by exploiting path-dependent dataflow information arising from the `instanceof` test.

Declaration splitting. Occasionally, a single variable declaration was used sequentially for two different *webs* (du-ud chains), such as using `Iterator i` to traverse first one list, then another of a different type. Even though the webs are disjoint, the single declaration of `i` means the analysis infers a single type for `i`. Similarly, multiple variables declared in the same statement, such as `Iterator i, j;`, are constrained to have the same type.

In *JavaCUP*, we found one example of each. After we manually split the declarations, the analysis eliminated 6 more casts (100%).

9. RELATED WORK

Our primary contribution in this paper is a practical refactoring tool for automated migration of existing Java programs into JSR-14. We first discuss work related to our goal; namely, existing work on introducing generic types into programs to broaden the applicability of a pre-existing components. Then, we briefly discuss work related to our techniques: type constraint systems and type inference.

9.1 Generalization for re-use

Two notable previous papers [39, 17] use automated inference of polymorphism with the goal of source-code generalization for re-use — for example, to permit the code to be used in more situations or to provide compile-time type correctness guarantees. Since the result is source code for human consumption, rather than deductions for later analysis or optimization, a primary goal 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 [39] aim to translate C to C++. They use type inference to detect latent polymorphism in C functions designed for use with parameters of primitive type, and the result of generalization is a collection of C++ function templates that operate on a larger set of types. A major issue addressed by Siff and Reps is that C++ classes can overload arithmetic operators for class types. Their algorithm determines — and documents — the set of constraints imposed by the generalized 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. The problem domain is quite different than ours, because unlike JSR-14, C++ templates need not type-check and are never separately compiled; the template is instantiated by simple textual substitution, and only the resulting code need type-check. This permits the template to impose arbitrary (implicit) constraints on its type variables, in contrast to JSR-14's erasure approach.

Duggan [17] presents a type analysis for inferring genericity in a Java-like language. Duggan gives a modular (intra-class) constraint-based parameterization analysis that translates a monomorphic object-oriented kernel language called MiniJava into a polymorphic variant, PolyJava, that permits abstracting classes over type variables. The translation creates generic classes and parameterized instantiations of those classes, and it makes some casts provably redundant. PolyJava differs from JSR-14 in a number of important respects. In particular, it supports a very restricted model of parametric subtyping: abstract classes and interfaces are not supported, and each class must declare exactly as many type variables as its superclass. The type hierarchy is thus a forest of trees, each of which has exactly the same number of type variables on all classes within it. (Each tree inherits from Object⟨⟩ via a special-case rule.) Because the analysis does not use client information to reduce genericity, we suspect the discovered generic types are unusably over-generic; however, the system is not implemented, so we are unable to confirm this.

Von Dincklage and Diwan [47] address both the parameterization and instantiation problems. They use a constraint-based algorithm employing a number of heuristics to find likely type parameters. Their Ilwith tool determined the correct generalization of several classes from the standard libraries, after hand editing to rewrite constructs their analysis does not handle. The technique requires related classes to be analyzed as a unit. However, it does not perform a whole-program analysis and so can make no guarantees about the correctness of its choices of type arguments. In contrast to our sound approach, they try to capture common patterns of generic classes using an unsound collection of heuristics. For example, their implementation assumes that public fields are not accessed from outside the class and that the argument of `equals` has the same type as the receiver. Their approach can change method signatures without preserving the overriding relation, or change the erasure of parameterized classes, making them possibly incompatible with their existing clients. Also in contrast to our work, their approach fails for certain legal Java programs, they do not handle raw types, their implementation does not perform source translation, and they do not yet have any experience with real-world applications. (We previously explored a similar approach to the parameterization and instantiation problems [16]. We restricted ourselves to a sound approach, and abandoned the approach after discovering that heuristics useful in specific circumstances caused unacceptable loss of generality in others.)

Tip et al. [42] present a technique for migrating non-generic Java code to use generic container classes. Tip et al.'s algorithm employs a variant of CPA to create contexts for methods and then uses these contexts in type constraint generation and solving. In our approach, CPA is used for pointer analysis, the results of which are then used to compute allocation site type arguments and, lastly, context-less type constraints are used to compute type arguments for all declarations in the client code. Their tool is implemented as a source-code analysis and a refactoring in the Eclipse [18] integrated development environment (IDE). Because it focuses only on the standard collections library and it is source-code-based, their approach uses hand-made models of the collection classes. While they do not handle raw types, their method is capable of discovering type parameters for methods, thus changing them into generic methods. This may help reduce the number of (possibly dangerous) unchecked warnings and raw references without sacrificing the number of eliminated casts. For example, the method `display-`

[14] Interestingly, this is an example of a JSR-14 program that requires *more* casts than its non-generic counterpart.

`Value` in Figure 3 could be changed into a generic method, rather than leaving the reference raw. The authors do not discuss soundness or behavior preservation.

Tip, Kieżun, and Baümer [43] present the use of type constraints for refactoring (i.e., modifying the program's source code without affecting its behavior). While their work focused on refactoring for generalization, ours can be seen as refactoring for *specialization*, changing types from raw to non-raw.

The CodeGuide [12] IDE offers a 'Generify' refactoring with broadly the same goal as our work. It can operate over a whole program or a single class; we have verified that the latter mode is unsound, but because no details are provided regarding its implementation, we cannot compare it to our own. The IDEA [24] IDE also supports a Generify refactoring; again, no details about the analysis techniques are available, and we have not experimented with this tool.

9.2 Type constraint systems

Both our allocation type inference and declaration type inference are type-constraint-based algorithms in the style of Aiken and Wimmers [2], who give a general algorithm for solving systems of inclusion constraints over type expressions. Our type constraints are different in that they include guarded constraints in order to model JSR-14's special rules for raw types. Most work in type inference for OO languages is based on the theory of type constraint systems; a general theory of statically typed object-oriented languages is laid out in [36].

Our pointer analysis makes use of the conceptual framework of Wang and Smith [48]; we instantiate it with a particular set of choices for polymorphism that fit well with our problem. Plevyak and Chien [38] provide an iterative class analysis that derives control and data flow information simultaneously, with the goal of optimizations such as static binding, inlining, and unboxing. Some representative applications are statically discharging run-time casts [8, 48], eliminating virtual dispatch [3], and alias analysis [33, 32].

9.3 Polymorphic type inference

There is a vast literature on polymorphic type inference dating from Milner [30], who introduced the notion in the context of the ML programming language. Our goal is quite different than that of Algorithm W, since we are not trying to infer generic types, only the type arguments with which existing generic types are instantiated. Subsequent work [34, 35] extends Hindley-Milner typechecking to object-oriented languages and to many other application domains. More recent work that extends it to object-oriented languages uses type constraints instead of equality constraints [19, 17], just as our S-UNIFY algorithm does, though the technical machinery is different. McAdam et al. [29] extend ML with Java's subtyping and method overloading. The application of type inference algorithms generally falls into two categories: (1) enabling the implementation of languages in which principal typings for terms are inferred automatically, which saves the programmer from writing them explicitly, and (2) as a means of static program analysis, e.g., to eliminate casts or to resolve virtual method dispatches.

Gagnon et al. [20] 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.

10. FUTURE WORK

We would like to extend our tool into an interactive application that would allow the user to manually correct suboptimal results, and iteratively re-solve the constraint system after the user's annotations have been incorporated. This would make it very easy for users to achieve the ideal results.

Once the Java 1.5 specification is finalized, we would like to update the tool to include support for it. It is largely a superset of the version of JSR-14 we have been studying; its most significant difference is the introduction of wildcard types, as discussed in Section 6.5.1. We plan to make our tool available, once licensing issues are resolved (the implementation currently depends upon Sun's prototype compiler).

The C# language is a experiencing a parallel evolution towards generic types [49]; the ideas in our approach may be applicable to that language.

In order to further increase the number of casts eliminated from client code, we could replace our type constraint solution procedure (Section 6.4) with one that performs more exhaustive search or other optimizations. Addressing the parameterization problem (i.e., introduction of type parameters to class definitions) is another area of potential future work, though in our experience that part of the process is not particularly difficult or time-consuming.

11. CONCLUSION

With the release of Java 1.5, many programmers will wish to convert their programs to take advantage of the improved type safety provided by generic libraries. We have presented a general algorithm for the important practical problem of converting non-generic Java sources to use generic libraries, and an implementation capable of translating real applications.

Our algorithm achieves the goals laid out in Section 3. It is sound: it never infers an unsafe type for a declaration. It is behavior-preserving: it does not change method signatures, the erasure of parameterized classes, or other observable aspects of a program. It is complete: it produces a valid result for arbitrary Java input and arbitrary generic libraries. It is compatible with the JSR-14 generics proposal: in particular, its type system addresses all features of the proposal, including raw types. It is practical: we have produced an implementation that automatically inserts type parameters into Java code, without any manual intervention. It is precise: it eliminated the overwhelming majority of generic casts in real-world applications, and the translation was little different than the result of manual annotation.

As another contribution, our approach is, to our knowledge, the first analysis for Java that uses generic type annotations for targeting the use of context-sensitivity, such targeting is critical to overcome poor scalability. Our application is type analysis, but this technique could equally well be used for many other abstractions, such as interprocedural dataflow problems.

Raw types require the use of conditional constraints, since the type rules for accessing members through raw types and through parameterized types are quite different. The presence of raw types in the type system is a loophole allowing potentially unsafe operations; analyzing the effects of such operations requires a whole-program analysis. (In the absence of raw types and unchecked operations, it would be possible to solve the type inference problem soundly — although perhaps not as precisely — without pointer analysis.) Because of unchecked operations, the assignability relation in JSR-14 is *not* antisymmetric; in other words, `x=y; y=x;` may be permitted even when the types of `x` and `y` are unequal. This has some subtle ramifications for subtype constraint-based analyses, as the assignment constraint graph may have no subtype interpretation in pathological cases. Our work is unique in supporting raw types, which is essential for producing good results without forbidding many realistic programs.

Additional details on our work can be found in [15].

Acknowledgments

Comments from Todd Millstein and the anonymous referees improved the presentation of our work. This research was funded in part by NSF grants CCR-0133580 and CCR-0234651, the Oxygen project, and gifts from IBM and NTT.

REFERENCES

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, Aug. 1996.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, June 1993.

[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA*, pages 324–341, Oct. 1996.

[4] G. Bracha, N. Cohen, C. Kemper, S. Mark, M. Odersky, S.-E. Panitz, D. Stoutamire, K. Thorup, and P. Wadler. Adding generics to the Java programming language: Participant draft specifi cation. Technical report, Sun Microsystems, Apr. 27, 2001.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. GJ specifi cation. `http://www.cis.unisa.edu.au/~pizza/gj/Documents/#gj-specification`, May 1998.

[6] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, pages 183–200, Oct. 1998.

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

[8] R. Cartwright and M. Fagan. Soft typing. In *PLDI*, pages 278–292, June 1991.

[9] R. Cartwright and G. L. Steele Jr. Compatable genericity with run-time types for the Java programming language. In *OOPSLA*, pages 201–215, Oct. 1998.

[10] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented language. In *PLDI*, pages 146–160, Portland, OR, USA, June 1989.

[11] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, pages 49–70, Oct. 1989.

[12] OmniCore CodeGuide. `http://www.omnicore.com/codeguide.htm`.

[13] B. De Sutter, F. Tip, and J. Dolby. Customization of Java library classes using type constraints and profi le information. In *ECOOP*, June 2004.

[14] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, Aug. 1995.

[15] A. Donovan. Converting Java programs to use generic libraries. Master's thesis, MIT Dept. of EECS, Sept. 2004.

[16] A. Donovan and M. D. Ernst. Inference of generic types in Java. Technical Report MIT/LCS/TR-889, MIT Lab for Computer Science, Mar. 22, 2003.

[17] D. Duggan. Modular type-based reverse engineering of parameterized types in Java code. In *OOPSLA*, pages 97–113, Nov. 1999.

[18] Eclipse project. `http://www.eclipse.org/`.

[19] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, pages 169–184, Oct. 1995.

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

[21] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specifi cation*. Addison Wesley, Boston, MA, second edition, 2000.

[22] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[23] M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, Jan. 2001.

[24] JetBrains IntelliJ IDEA. `http://www.intellij.com/idea/`.

[25] A. Igarashi, B. C. Pierce, and P. Wadler. A recipe for raw types. In *FOOL*, London, Jan. 2001.

[26] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *ECOOP*, pages 441–469, June 2002.

[27] JavaSoft, Sun Microsystems. Prototype for JSR014: Adding generics to the Java programming language v. 1.3. `http://jcp.org/jsr/detail/14.html`, May 7, 2001.

[28] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, Nov. 1994.

[29] B. McAdam, A. Kennedy, and N. Benton. Type inference for MLj. In *Scottish Functional Programming Workshop*, pages 159–172, 2001. Trends in Functional Programming, volume 2, Chapter 13.

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

[31] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, pages 132–145, Jan. 1997.

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

[33] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, May 1997.

[34] A. Ohori and P. Buneman. Static type inference for parametric classes. In *OOPSLA*, pages 445–456, Oct. 1989.

[35] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, Oct. 1991.

[36] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley and Sons, 1994.

[37] P. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser. *The C++ Standard Template Library*. Prentice Hall PTR, 2000.

[38] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, pages 324–340, Oct. 1994.

[39] M. Siff and T. Reps. Program generalization for software reuse: From C to C++. In *FSE*, pages 135–146, Oct. 1996.

[40] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

[41] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, special edition, 2000.

[42] F. Tip, R. Fuhrer, J. Dolby, and A. Kieżun. Refactoring techniques for migrating applications to generic Java container classes. IBM Research Report RC 23238, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, June 2, 2004.

[43] F. Tip, A. Kieżun, and D. Bäumer. Refactoring for generalization using type constraints. In *OOPSLA*, pages 13–26, Nov. 2003.

[44] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, Oct. 2000.

[45] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *SAC*, pages 1289–1296, Mar. 2004.

[46] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java bytecode optimization framework. In *CASCON*, pages 125–135, Nov. 1999.

[47] D. von Dincklage and A. Diwan. Converting Java classes to use generics. In *OOPSLA*, pages 1–14, Oct. 2004.

[48] T. Wang and S. Smith. Precise constraint-based type inference for Java. In *ECOOP*, pages 99–117, June 2001.

[49] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *POPL*, pages 39–51, Jan. 2004.