# Converting Java Programs to Use Generic Libraries

by

Alan A. A. Donovan

B.A., University of Cambridge, 1996



Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004



Author .....
Department of Electrical Engineering and Computer Science
September 10, 2004

Certified by .....
Michael D. Ernst
Douglas T. Ross Career Development Assistant Professor of Computer Software Technology
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

*To three Scientists and Mentors:*

*Peter Grove, Stephen Downes, John Bridle*

# Converting Java Programs to Use Generic Libraries

by

Alan A. A. Donovan

Submitted to the Department of Electrical Engineering and Computer Science
on September 10, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

## 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 practise 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: parameterisation (adding type parameters to class definitions) and instantiation (adding the type arguments at each use of a parameterised class). Parameterisation 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 safely handles all features of the JSR-14 type system, notably the raw types (which provide backward compatibility) and 'unchecked' operations on them. We have implemented our analysis in a tool that automatically inserts type arguments into Java code, and we report its performance when applied to a number of real-world Java programs.

Thesis Supervisor: Michael D. Ernst
Title: Douglas T. Ross Career Development Assistant Professor of Computer Software Technology

# Acknowledgements

Much of this thesis draws on work published in the (forthcoming) proceedings of OOPSLA 2004 [DKTE00].

This work would not have been possible without the support, guidance and collaboration of my adviser, Michael Ernst. He brought clarity during times of confusion, and encouragement at times of despair. He was always available and willing to discuss ideas and problems, and at both blackboard and keyboard, he put an enormous amount of effort into this work.

I also owe thanks to my colleagues and neighbours Scott Ananian, Chandra Boyapati, Viktor Kuncak, Patrick Lam, Darko Marinov, Stephen McCamant, Derek Rayside, Martin Rinard, Alex Salcianu for their technical expertise and practical advice; and in particular to my co-author and collaborator Adam Kieżun, for his input at every stage, from brainstorming to writing; and to Matthew Tschantz, for his tireless work on the implementation.

I am indebted to David Bacon, David Grove, Michael Hind, Stephen Fink, Rob O'Callahan and Darrell Reimer at IBM Research, for stimulating discussion and for broadening my understanding of the field.

This research was funded in part by NSF grants CCR-0133580 and CCR-0234651, the Oxygen project, and gifts from IBM and NTT.

Finally, special thanks are due to two people: to my brother Stephen, without whose encouragement, I would never have started; and to my lover Leila, without whose warmth and support, I would never have finished.

# Contents

# List of Figures

# Chapter 1

# Introduction

The next release of the Java programming language [GJSB00] will include support for generic types. Generic types (or *parametric polymorphism* [CW85]) make it possible to write a class or procedure abstracted over the types of its method arguments. This feature represents a major extension to Java; since its inception, support for generic types has been one of the most wished-for features in the Java community—in fact, its inclusion was the #1 request-for-enhancement for many years [Jav04].

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 two problems:

- **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. Problems of this kind are compounded by the spatial and temporal separation between the cause of the fault and the point at which it is discovered. Insertion of the wrong class of object into a container, for example, will always succeed, but it is not until the value is later extracted from the container and downcast, that the error will be reported. In the interval between these two events, which may be arbitrarily long, subtle violations of program invariants may occur. When the fault is finally reported, the location at which it is discovered may shed little light on where the problem was caused.

By using a more expressive type system, such errors can be eliminated at compile-time.

- **An incomplete specification:** The types in a Java program serve as a rather weak specification of the behaviour of the program and the intention of the programmer.

  In the absence of polymorphic types, programmers must insert the additional information in the form of casts and comments. It is the programmer's responsibility to ensure that these annotations remain accurate and consistent as the code evolves.

  By using a more expressive type system, the consistency of these annotations can be checked by the compiler, so their accuracy is guaranteed.

  Automated checking is not the only reason to prefer polymorphic types over the use of casts. Types are *declarative*, meaning that they state invariants of the program, whereas casts are *imperative*, meaning that they are operations that transform the state of the program. Programs without state are generally easier to write and easier to reason about once written.

Non-generic solutions to the problems (e.g., wrapper classes such as `StringVector`) are unsatisfying. They introduce non-standard and sometimes inconsistent abstrac-

tions 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 [BCK+01] that is being incorporated into Java 1.5) solves these problems while maintaining full interoperability with existing Java code.

## 1.1 Automatic translation

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 ('parameterisation'), and modifying uses of the classes to supply the type arguments ('instantiation').

Parameterisation 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. Without having seen the code before, we were able to fully annotate 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).

We believe the instantiation problem is the more important of the two for several reasons. 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. Therefore, this work focuses on the instantiation problem.

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 Java 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 behaviour remains unchanged, and (b) as many casts as possible can be removed.

## 1.2 Contributions of this thesis

The primary contributions of this thesis are (a) a general algorithm for the important practical problem of automatically converting existing non-generic Java sources to use generic libraries, and (b) a practical implementation of the algorithm, capable of translating real-world applications, which not only demonstrates the effectiveness of the algorithm, but is a useful software tool.

Our algorithm strives for both correctness and practicality, and achieves several goals (see Chapter 3) unmet by both prior and contemporary work. Of particular importance, its results are sound: even in the presence of JSR-14's raw types, and it does not change the observable behaviour of the program. It is also precise: according to the metric of cast elimination, it produces results very close to ideal. The algorithm is complete, producing valid results for arbitrary Java input and arbitrary generic libraries, is fully compatible with the JSR-14 generics proposal, and avoids arbitrary simplification of either the source or target language.

This work is the first to address the problem of type-constraint generation for JSR-14's raw types. Our solution makes use of conditional constraints, which are solved by a backtracking procedure.

A secondary contributions of this work is that this is the first analysis for Java (to

our knowledge) that uses generic type annotations for targeting the use of context-sensitivity in a pointer analysis. This technique could be equally well applied to many other interprocedural analyses.

## 1.3 Synopsis of the algorithm

In this work, we are trying to solve the problem of automated translation of non-generic programs that are clients of generic library classes, into semantically-equivalent JSR-14 programs that make effective use of generic types. For each use of a generic type throughout the input program, the goal is to select type arguments that are not only sound, but as precise as possible.

**Soundness:** A chosen parameterised type is *sound* if it represents an invariant of the program. For example, a variable `l` may be soundly ascribed the type List⟨String⟩ if it refers only to List instances, all of whose elements are instances of String. In other words, the type List⟨String⟩ is a conservative approximation to the values to which `l` can refer. As we shall see, JSR-14's *raw types* and *unchecked operations*, provided for legacy compatibility, are a loophole in the type system, allowing *unsound* typings: in certain cases, it is possible to ascribe the type List⟨String⟩ to a list, some of whose elements are *not* instances of String. Avoiding unsound typings is a major focus of our approach.

**Precision:** One way of ensuring soundness is to choose very conservative type arguments. However, more specific types are generally preferable as they contain more information and have the potential to make casts redundant, allowing them to be eliminated in the translated program. The more precise the inferred types, the more casts are eliminated.

We divide the task into two parts: *allocation type inference* and *declaration type inference*.

**Allocation type inference** proposes types for each allocation site (i.e., each use of `new`) in the client code. It does so in three steps. Firstly, it performs a context-sensitive pointer analysis that determines the set of allocation sites to which each

expression may refer. Secondly, for each *use* (method call or field access) of an object of a generic class, it unifies the pointer analysis information with the declared type of the use, thereby constraining the possible instantiation types of the relevant allocation sites. Thirdly, 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 sound parameterised type for each object allocation site.

To achieve soundness, the allocation type inference is a whole-program analysis, as local analysis alone 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—in other words, it is able to prove certain facts about the program behaviour that cannot by proven by the type system.

**Declaration type inference**, the second part, determines a consistent re-typing of all entities in the client code. It achieves this in two steps. The first step constructs a system of type constraints that captures the requirements of a well-formed JSR-14 program that is semantically equivalent to the input program. The second step then finds a solution to this constraint system. From the solution, the algorithm obtains the new parameterised types for all uses of generic classes, including variable declarations (fields, local variables, method formal parameters and results), casts and allocation sites.

The results of allocation type inference are incorporated into the system as type constraints, though the type eventually chosen for each allocation site need not be exactly the type proposed for it by allocation type inference.

The declaration type inference is context-insensitive, and its output is sound with respect to the JSR-14 type system. It can be supplied the whole of the client program, but can also be run on any part of a program, in which case it 'frames' the boundaries (i.e., constrains the types at the interface so that they will not change), although a smaller part leads to a more constrained problem, and gives the algorithm less freedom to choose new types.

We have implemented these algorithms in a fully-automatic tool, *Jiggetai*, which not only performs these type analyses, but also edits the original program's source

code to insert the inferred type arguments and excise the casts obviated by the new types.

## 1.4 The structure of this document

Throughout the text, we use a running example to illustrate our techniques. Figure 2-2 on page 26 shows a simple generic library; it defines the class `Cell`, which is a container holding one element, and its subclass `Pair`, which holds two elements, possibly of different types. Figure 2-3 on page 30 shows an example client of the generic classes in the library; the code defines a number of methods that create and manipulate `Cells` and `Pairs`. Figure 2-4 shows the client code after being automatically transformed by our tool. The syntax and semantics of JSR-14, the generic type system that is to be adopted for Java 1.5, are presented in Chapter 2, with a discussion of some important aspects of its design that make the translation problem more challenging, and comparison against several other implementations of generics in Java and other languages.

Chapter 3 lays out our design goals and assumptions, explains the concepts of soundness and precision in the context of this work, and motivates the translation problem in greater detail.

The two main parts of the type analysis algorithm, allocation type inference and declaration type inference, are explained in detail in Chapters 4 and 5 respectively.

To implement the type analysis algorithms in a practical tool, several significant technical and engineering challenges had to be addressed. Some of these are described in Chapter 6.

To understand how our algorithms and implementation perform in practise, we conducted experiments to determine how many of the generic casts present in the input program Jiggetai was able to eliminate, for a number of realistic programs. Chapter 7 presents the results of our experiments, an evaluation of the tool's performance and a discussion of ways in which the results could be improved.

Chapter 8 situates our research in relation to prior work in the fields of auto-

mated refactoring, polymorphic type inference, and the use of type constraint systems, and also to some contemporary attempts specifically related to the translation of Java to exploit generics.

Chapter 9 discusses avenues for future work on this problem, and we conclude in Chapter 10 with a summary of our contributions.

# Chapter 2

# Background: Java with generic types

Almost since the inception of the Java language, its lack of support for generic types, or *parametric polymorphism*, has been recognised as a problem. Proposals for the inclusion of some form of generics into the language abound [MBL97, OW97, AFM97, SA98, BD98, BOSW98b, CS98, FKF98, TKH99, VN00, AC02, ABC03], and many are almost as old as Java itself. This is not surprising, since Java otherwise bears many similarities with C++, whose *template* mechanism supports parametric polymorphism. Many other languages support generics, including the object-oriented languages Eiffel, CLU, Modula-3, Ada, Sather, Cecil, and also the functional languages ML, Miranda and Haskell.

In this chapter, we will examine the syntax and semantics of JSR-14 [BCK+01], the proposal for generic types in Java that has been adopted by Sun and will be incorporated into Java 1.5, pointing out a number of significant features of JSR-14 that make the problem of automatic translation more challenging. In order to gain a better understanding of how JSR-14 works and a rationale for the decisions made in its design, Section 2.4 explores and compares six of the more developed proposals from across the design space.

## 2.1 Parametric polymorphism

Parametric polymorphism is a mechanism that permits a programmer to define a single piece of 'generic' code that works uniformly, abstracted over many distinct types. For instance, the parameters of a generic method may have different types for different calls of the method; similarly, the members (fields and methods) of a generic class may have different types for different allocations of class instances. Type parameters stand for the types over which the code is (conceptually) instantiated.

Robin Milner, the author of the first polymorphically-typed language ML, pithily remarked that "well-typed programs cannot 'go wrong'" [Mil78], meaning that every type system statically eliminates a certain class of run-time errors. For parametric polymorphism, that class of errors includes, for example, inconsistent use of collection objects, as the extra expressiveness of polymorphic types captures information about the elements of collections.

With generic types, the programmer is able to specify a type List⟨String⟩ representing the set of lists whose elements are instances of String, and the type system distinguishes this from, say, List⟨Integer⟩, providing a different set of operations for each type. For example, one may insert a String, but not an Integer, into a List⟨String⟩; and the result type of the `get` method used to extract an element from a List⟨String⟩ is String, which is distinct from the type of `get()` for a List⟨Integer⟩.

In non-generic Java, the result type of `List.get`, Object, captures no information about the type of the list elements, so in the absence of polymorphic types, programmers must insert the additional information in the form of run-time type assertions (casts) and comments, and the burden of maintaining consistency is upon them.

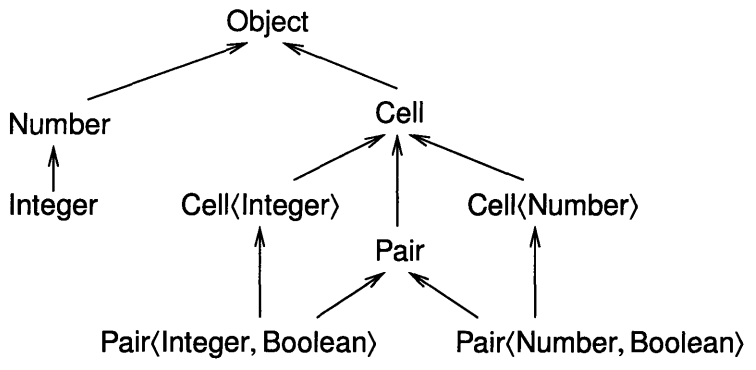



Figure 2-1: A portion of the subtype relation for JSR-14, which uses invariant parametric subtyping. Arrows point from subtypes to supertypes. Classes Cell and Pair are defined in Figure 2-2.

## 2.2 JSR-14 syntax

JSR-14 [BCK+01] is a generic type system using bounded parametric polymorphism (explicit nominal subtype constraints on parameters), implemented with a homogeneous translation based on type erasure.

Figure 2-2 shows the definition of two generic classes in JSR-14. 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. 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 `extends`-clause.

Figure 2-3 shows an example non-generic client of the library classes, and 2-3 shows the same program, translated to make use of generic types. A generic class may be instantiated (used) by supplying type arguments that are consistent with the bounds on the type variables. Type-checking ensures that the code is type-correct,

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

no matter what type arguments that satisfy the bounds are used. (See below for a caveat regarding raw types.)

Methods may also be generic, declaring their own additional type variables. In Figure 2-2, `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 2-4 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 2-4, parameter `c6` in `displayValue` is raw.) Raw types are a concession to backward compatibility, and they behave exactly like types in non-generic Java.

## 2.3 JSR-14 type system

This section informally overviews salient points of the type system of JSR-14. Figure 2-1 shows part of the subtype relation.

### 2.3.1 Type erasure

The type rules of JSR-14 suggest the implementation strategy of *type erasure*, in which after the parameterised 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)`.

### 2.3.2 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 behaviour 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 type arguments 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.

There are some further restrictions on the use of type variables within generic classes or methods: instances of type variables may not be used within allocation expressions, whether of class instances, such as `new V()`, or of arrays, such as `new V[10]`. Under homogeneous translation, each generic method in the source is implemented by a single compiled method, and the instantiation of the type variable `V` is therefore unknown during code generation. As a result, the identity of the constructor that must be called for `new V()` cannot be determined, so this code is impossible to compile; the desired effect can only be achieved via reflection.

### 2.3.3 Invariant parametric subtyping

Different instantiations of a parameterised type are unrelated by the subtype relation. For example, Cell⟨Integer⟩ is not a subtype of Cell⟨Number⟩, even though Integer is a subtype of Number. Though this may be initially surprising, this is the right choice because Cell⟨Integer⟩ does not support the `set(Number)` operation. Consider the alternative, *covariant* subtyping: if Cell⟨Integer⟩ were a subtype of Cell⟨Number⟩, we should be able to call `set(f)` where `f` is a Float; but this would violate the invariants of the type Cell⟨Integer⟩ and so would need to be prevented by a dynamic check. Since JSR-14 uses homogeneous translation by type erasure, run-time checking of the type arguments of a parameterised type is impossible, and soundness must be ensured statically; hence, invariant subtyping is a necessary consequence of the decision to use type erasure.[1]

### 2.3.4 Raw types

Great effort was expended in the design of JSR-14 to ensure maximum compatibility with existing non-generic code, in two directions. *Backward* compatibility guarantees that all previously legal Java programs remain legal, and semantically equivalent, in JSR-14. *Forward* compatibility ensures that libraries compiled using JSR-14 are callable even from non-generic code (although the parameterised types of JSR-14 are projected onto the ordinary Java type system — in other words, they appear *erased*).

The type system of JSR-14 subsumes that of non-generic Java, with the unparameterised types of generic classes such as `Cell` being known as *raw* types, and raw types being supertypes of parameterised versions. (A raw type can be considered roughly equivalent to a type instantiated with a bounded existential type, e.g., Cell⟨∃x . x ≤ Object⟩, because clients using a raw type expect *some* instantiation of

[1] By contrast, Java arrays *do* use covariant subtyping: Integer[] is a subtype of Number[] because Integer is a subtype of Number. In order to preserve type safety, the JVM implementation must perform a run-time check at every array store. This subtyping strategy is not only the bane of JVM implementors, but also a violation of the *substitutability principle* of subtyping [LW94].

the corresponding generic class, but have no information as to what it is [IPW01]. As we shall see, however, raw types differ from existential types in that the former permit the operations that the latter forbid; only an warning is issued.)

### 2.3.5 Unchecked operations

In non-generic Java, the type system ensures that the type of an expression is a conservative approximation of the 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.

A parameterised type such as Cell⟨String⟩ represents the invariant that variables of that type refer only to instances of Cell whose `value` field — or more generally, any occurrence of T within the declaration of Cell — is an instance of String.

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 parameterised types in the program, resulting in a ClassCastException being thrown during execution (see Section 3.1 on page 41 for an example). The compiler issues a warning when it compiles such operations, but it is the programmer's responsibility to ensure that all unchecked operations are in fact safe. (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 non-generic code, though it is less safe than use of non-raw parameterised types.)

As one example of an unchecked operation, implicit coercion is permitted from a raw type to any instantiation of it: that is, from a supertype to a subtype. 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.

```
1  static void example() {
2    Cell c1 = new Cell(new Float(0.0));
3    Cell c2 = new Cell(c1);
4    Cell c3 = (Cell) c2.get();
5    Float f = (Float) c3.get();
6    Object o = Boolean.TRUE;
7    Pair p =
8        new Pair(f, o);
9
10   Cell c4 = new Cell(new Integer(0));
11   c4.replaceValue(c1);
12
13   displayValue(c1);
14   displayValue(c2);
15
16   setPairFirst(p);
17
18   displayNumberValue(p);
19   displayNumberValue(c4);
20
21   Boolean b = (Boolean) p.second;
22 }
23 static void setPairFirst(Pair p2) {
24   p2.value = new Integer(1);
25 }
26 static void displayNumberValue(Cell c5) {
27   Number n = (Number) c5.get();
28   System.out.println(n.intValue());
29 }
30 static void displayValue(Cell c6) {
31   System.out.println(c6.get());
32 }
```

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

```
1  static void example() {
2    Cell<Float> c1 = new Cell<Float>(new Float(0.0));
3    Cell<Cell<Float>> c2 = new Cell<Cell<Float>>(c1);
4    Cell<Float> c3 = (Cell) c2.get();
5    Float f = (Float) c3.get();
6    Boolean o = Boolean.TRUE;
7    Pair<Number,Boolean> p =
8        new Pair<Number,Boolean>(f, o);
9
10   Cell<Number> c4 = new Cell<Number>(new Integer(0));
11   c4.replaceValue(c1);
12
13   displayValue(c1);
14   displayValue(c2);
15
16   setPairFirst(p);
17
18   displayNumberValue(p);
19   displayNumberValue(c4);
20
21   Boolean b = (Boolean) p.second;
22 }
23 static void setPairFirst(Pair<Number,Boolean> p2) {
24   p2.value = new Integer(1);
25 }
26 static void displayNumberValue(Cell<Number> c5) {
27   Number n = (Number) c5.get();
28   System.out.println(n.intValue());
29 }
30 static void displayValue(Cell c6) {
31   System.out.println(c6.get());
32 }
```

Figure 2-4: Example client code of Figure 2-3, after being automatically updated to use generic types. Changed declarations are underlined. Eliminated casts are struck through.

### 2.3.6 Versions of JSR-14

JSR-14 [Jav01] was inspired by Generic Java [BOSW98b, BOSW98a]. 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 [THE+04] 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 parameterised types whose arguments are bounded existential types, but wildcard types generalise this idea, allowing the bounds to express either subtype or supertype constraints [IPW01, IV02]. 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 5.5.1 for further detail. This will benefit both the union elimination (Section 5.5) and constraint resolution (Section 5.4) components of our algorithm.

A second, minor difference is that Java 1.5 beta 2 forbids array creation expressions for arrays of parameterised 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.

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 finalised.

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

## 2.4 Design space for generics in Java

In this section, we compare six of the more developed proposals for generics in Java, from across the design space. JSR-14 [BCK+01] is an evolutionary development of two previous proposals, called Pizza [OW97], and Generic Java (GJ) [BOSW98b]. The other three are PolyJ [MBL97], the 'Stanford proposal' [AFM97] and NextGen [CS98]. We will also draw comparisons with the approaches to parametric polymorphism taken by the languages C++ [SE90] and C♯ [YKS04]. (This section discusses work related to the design of JSR-14; see Chapter 8 for coverage of the work related to our own contributions.)

The proposals for generics in Java can be classified according to two principal taxonomic criteria. The first of these is the distinction between *homogeneous* and *heterogeneous* translation. The second criterion is the mechanism by which the language allows the specification of restrictions upon the types at which a type variable may be instantiated. We will examine these two criteria in turn.

### 2.4.1 Homogeneous vs. heterogeneous translation

There are two general strategies for the implementation of generic types: *homogeneous* translation, in which a single class-file serves as the implementation for all parametric types, and *heterogeneous* translation, in which each instantiation of a generic type is implemented by a separate class-file.

Homogeneous translation is based on *type erasure*: after type-checking is performed, the generic type annotations are discarded by the compiler (with casts inserted as required) yielding, conceptually, a program in traditional Java for which code is then generated as normal. Homogeneous translations have the advantage of simplicity and compactness (of generated code), but at a cost: there is no information available at run-time that can distinguish instances of two different instantiations of the same generic type — for example, a List⟨String⟩ from a List⟨Integer⟩ — and this has ramifications for the programming model; in particular, for casts, `instanceof` and reflection.

In contrast, heterogeneous translation can be considered akin to macro-expansion: each instantiation of the generic type causes a new copy of the class to be created with a fresh name; its type-variables are then $\beta$-reduced and code is generated anew. This approach is exemplified by C++'s template mechanism. Since each parametric type in a heterogeneous system is implemented by a distinct class, at run-time one can query an object for its full parametric type. However, the largely-redundant duplication of code can be expensive and implementations are somewhat complex. (Both of these are common criticisms of C++ templates.)

The choice of a homogeneous or heterogeneous translation also affects the semantics of static members of a class. With a homogeneous translation, the static members are shared between all instantiations of the generic type, so they are logically outside the scope of the type variables. With a heterogeneous translation, each instantiation receives a separate copy of the static members so they too can refer to type variables.

(The terms *homogeneous* and *heterogeneous* are due to Odersky & Wadler [OW97], who implemented both approaches in their proposal, Pizza. Strictly speaking, they refer to implementation strategies rather than semantic differences. In practise, however, these cannot be completely separated: it would be difficult to implement the semantics associated with the heterogeneous translation by using a homogeneous translation, and vice versa.)

In Java, the homogeneous approach has won out, and is used by JSR-14, its predecessors Pizza and Generic Java, and also by PolyJ. It is certainly conceptually closer to current (non-generic) practise, in which a single List class is used to implement all list instances, whether they contain String or Integer. Another crucial (though subtle) reason to use homogeneous translation, related to security, is discussed in the context of Generic Java [BOSW98b].

The NextGen and Stanford proposals are both based upon heterogeneous translation, but via quite different mechanisms; while the former resembles C++ in its duplication of class-files, the latter extends the class-file format and the JVM's class

loader to perform the required expansion on-the-fly during class loading, thereby keeping the static size of the code down.

Generics in C++ and $C^\sharp$ are also based upon heterogeneous translation. C++ uses a 'brute-force' expansion of the generated code, so that *every* distinct instantiation of a class creates a new copy of its code—even when the generated code is identical to a previous expansion, as is often the case; this can result in substantial code bloat. In contrast, $C^\sharp$ compiles first to a higher-level typed intermediate language and the heterogeneous expansion occurs only inside the virtual machine (as in the Stanford approach). The VM aggressively caches and re-uses existing compiled methods to avoid much of the overhead experienced with C++.

All of the languages based on heterogeneous implementations (NextGen, Stanford, C++, $C^\sharp$) allow the full parameterised type of an object to be queried at runtime, whether by casts, `instanceof` (and its C++ and $C^\sharp$ homologues) or reflection.

### Instantiation at primitive types

The fundamental distinction between primitive types (such as `int`) and reference types (such as `java.lang.Object`) poses a problem for implementations of generics, particularly ones based upon homogeneous translation: a single representation of a generic method cannot manipulate both primitive and reference values. For instance, primitive types do not support any of the methods of `Object`, such as `hashCode()`, nor are the primitive type domains lifted by a `null` element.

Two of the proposals (PolyJ and Stanford) support the instantiation of generics at primitive types (e.g. `List<int>`). PolyJ, which is based on homogeneous translation, uses implicit boxing conversions (i.e., conversions between `int` and `java.lang.Integer`) to allow instances of primitive types to be used by generic code that manipulates reference types. The Stanford proposal takes a different approach, and performs load-time translation, from bytecodes which manipulate reference types to bytecodes for the required primitive type; additional work is required to handle the 64-bit types `long` and `double` because of their size.

For the two other languages based upon heterogeneous translation, C++ and $C^\sharp$,

instantiation at primitive types is relatively straightforward: the body of the generic class or method is specialised with the type variable(s) substituted for a primitive type[3]; this specialisation results in significantly more efficient generated code than an approach based on implicit or explicit boxing.

All of the other proposals, including JSR-14, simply forbid the instantiation of generics at primitive types. (In the current Java 1.5 specification, implicit boxing conversions have been added across the language, so in practise, this restriction is not as onerous as it may seem.)

### 2.4.2 Constraints on type variables

Some generic classes place restrictions on the types with which they may be instantiated. For example, whereas a `List` can contain any kind of element, a `SortedList` might require that its elements implement an ordering. These restrictions form a part of the contract between the code that implements a generic class and code that instantiates it, and a number of models exist for how these constraints are specified.

#### Implicit structural subtyping

C++ templates, which are semantically similar to macro-expansion, are an example of *implicit* constraints: code within the scope of the type variables may freely invoke methods, and read or write fields of type variables, without explicitly specifying constraints on the validity of the type arguments that will instantiate its type variables. Only when the template is used, and the body of the template is macro-expanded with type variables substituted for instantiating types, can the method invocations and field accesses be type-checked.

This approach is not consistent with separate compilation, since the type-correctness of template code cannot be checked until it is instantiated, and type errors may be latent within it. Indeed, it is possible to write template code which

[3]For C++, the expansion occurs at the abstract-syntax level during compilation; for $C^\sharp$, it happens at the typed intermediate language level at load-time in the VM.

induces a contradictory set of constraints on its type variables and which consequently cannot be instantiated at all.

It is possible to analyse the uses of the type variables within the template to determine the complete set of constraints implicitly imposed by them [SR96]. In effect, these uses describe a record type $R$ containing all the fields and methods required by the uses, and the constraints can be considered a form of *structural subtype* constraint, since only types that are a structural subtype of $R$ are suitable. For example, a template which invokes a method `int i = x.f(1.0)` and reads a field `String s = x.g;` imposes the structural subtype constraint $[\![\texttt{x}]\!] \leq_S$ `[f: double → int; g: String]` on the type of `x`[4].

We call this approach implicit structural subtyping because it is equivalent to structural subtyping but the constraints are implicit — which means they are not available even during type-checking of the template code. No proposal for generic types in Java has made use of implicit constraints since this would come at the cost of separate compilation, a key feature of Java.

### Explicit structural subtyping

An alternative approach is *explicit structural subtyping*, in which the generic code states its assumptions about the instantiating types for each type variable in the form of structural subtyping constraints. Such constraints are called *where*-clauses [LSAS77], since they are typically of the form:

```
class C<T> where T extends [f: double → int]
```

With these constraints made manifest, the code within the generic class can then be completely type-checked without any further information about the types at which it is later instantiated. This permits separate compilation, allowing the generic code to be compiled once, and making the type-checking of instantiations much simpler.

[4] $[\![\texttt{x}]\!]$ is pronounced "the type of x"; see Chapter 5 for further detail.

### Explicit nominal subtyping

The third mechanism for expressing constraints on type variables is *explicit nominal subtyping*, which means that one or more types are specified as *upper bounds* for a type variable, and it may only be instantiated with types that are a subtype of each of these bounds under the normal (nominal) subtyping rules of the language. This approach is also known as *bounded parametric polymorphism*.

Since the constraints on type variables are explicit, the code of the generic class may be type-checked by treating the type of each type variable $T$ as an existential type that is a subtype of each of $T$'s bounds.

The relative advantages and disadvantages of using nominal versus structural subtyping as means of expressing bounds constraints for type variables are similar to those of using these two subtype relations throughout the language [CW85]. Structural subtyping allows one to use a type $\tau_1$ where another type $\tau_2$ is required, if $\tau_1$ has 'compatible' structure (i.e., methods and fields) with $\tau_2$, whether or not the definition of $\tau_1$ refers to $\tau_2$. This is especially useful if $\tau_1$ was defined before $\tau_2$, as it allows $\tau_1$ to be retrospectively considered an instance of $\tau_2$, perhaps in a way unanticipated when $\tau_1$ was defined.

However, this flexibility is also the main drawback of the structural approach: the canonical problematic example is the `draw()` method, which is an instance of verb 'overloading' in English. Mere possession of this method by a class would not indicate that the class actually implements the semantic contract of any one of the hypothetical `Cowboy`, `Curtain` or `Picture` interfaces, each of which could define a semantically distinct `draw()` method.[5] In essence, the type signature of a method is only a partial specification of its behaviour: just because the type of an overriding method is consistent with the super-method, does not mean that its semantics are also. In contrast, with nominal subtyping, the set of interfaces implemented by the class, if any, is made fully explicit, and forms a reliable semantic contract.

Most of the proposals use the bounded parametric polymorphism (nominal) ap-

[5] As a result, one could pass a `Cowboy` where a `Curtain` was expected, with potentially injurious results.

proach since this is more consistent with the choice of nominal subtyping throughout the language. However, the PolyJ proposal reveals its heritage from the language CLU [LSAS77] by its use of *where*-clauses for expressing structural constraints.

# Chapter 3

# Design principles

We designed our analysis in order to be sound, behaviour preserving, compatible, complete and practical. This chapter 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, in the presence of unchecked operations, simply satisfying the compiler's type-checker is not sufficient to 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. Here is one such program:

```
Cell<Float> cf = new Cell<Float>();
Cell c = cf;
c.set(new Integer()); // Compiler Warning: unchecked operation (unsafe)
Float f = c.get(); // Runtime Exception: failed cast from Integer to Float!
```

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 Behaviour preservation

The translation must preserve the dynamic behaviour 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 change only type declarations and types at allocation sites; no other modifications are permitted. Changing other program elements could change behaviour, cause the code to diverge from its documentation (and from humans' understanding) and degrade its design, leading to difficulties in comprehension 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 programmers 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. (A tool working under a closed-world assumption could offer the option to change field and method signatures as long as the behaviour 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.[1]

Finally, we do not permit any changes to the source code of the library or the generic information contained in the compiled 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 behaviour preservation. The new code differs only in its type erasure and in the types of local variables; neither of these differences 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 trade-offs. (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—see Chapter 2.) 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 trade-offs 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,

[1] Strictly speaking, even though the types of local variables are not captured in the byte codes, they can affect the resolution of overloaded methods, or which version of a field (that is re-declared to shadow one in a superclass) is accessed. Therefore, an implementation should take additional measures to ensure that such behavioural changes do not occur; whilst our implementation does not yet do so, it is not a significant problem in practise.

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 [TFDK04], 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` uses generics to provide a more specific type for calls to `newInstance()`, and the `java.lang.ref` package uses generics to provide support for typed 'weak' references. Our own implementation also uses them for I/O adaptors that convert an object of one type to another (say, type T to byte[]), and the C++ Standard Template Library [PSLM00] 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, 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 manual tweaking is error-prone and tedious.

```
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 3-1: Java code with multiple non-trivial JSR-14 translations.

## 3.6 Success metric: cast elimination

There are multiple type-correct, behaviour-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.

Figure 3-1 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 parameterised 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 high-level application invariants known to the programmer, 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 prioritise removing raw types over removing casts if desired. However, some use of raw types is often required in practise.

In practise, 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 3-1 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 3-1, 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 2-4. 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 parameterise either classes or methods in client code. (The type parameterisation problem is beyond the scope of this work and appears to be of less practical importance.)

Our analysis is whole-program rather than modular; this is necessary in order to optimise 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. The issue of modularity is discussed further in Section 6.2.3.

# Chapter 4

# 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.

For example, in Figure 2-4 on page 31, 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.

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 no longer type-check against the converted part. Worse, if it was already compiled (or if it used a raw Cell type for `c`), the inconsistency would go undetected and cause a failure at run-time. Alternatively, the pointer analysis can avoid examining the whole program by making conservative approximations for the unanalysed 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.

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

Figure 4-1: Type grammar for allocation type inference

## 4.1 Definitions and terminology

Figure 4-1 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 $obj(C_i)$ is created [WS01]. This type identifier represents the type of all objects created at that allocation site. Some allocation sites within generic library code may be analysed many times, due to context-sensitivity (see Section 4.4), 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, minimising 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.

## 4.2 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 parameterised type for each allocation site that conservatively approximates all subsequent uses of the allocated object.

**1. Pointer analysis** (Section 4.4) 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 [HP01]. We use a context-sensitive pointer analysis based on the Cartesian Product Algorithm [Age95].

**2. S-unification** (Section 4.5) combines the results of pointer analysis with the declarations of generic library classes in order to generate subtype constraints. 'S-unification' stands for 'unification with subtyping'. Its name comes from its similarity to conventional unification: both generate constraints by structural induction over types. 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 Cell⟨Cell⟨Integer⟩⟩), so if more information becomes available, previous s-unifications may need to be re-done.

The result of the s-unification step is a set of constraints on the values of the type variables at each generic class allocation site. For example, Cell⟨V⟩ has method `set(V)`. If we determine that for the code `c.set(x)`, POINTS-TO(x) = {obj(String)} and POINTS-TO(c) = {obj($\text{Cell}_2$)}, then we know that the instantiation of V in obj($\text{Cell}_2$) 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($\text{Cell}_2$). We write this as String $\leq \text{V}_{\text{obj}(\text{Cell}_2)}$; see Section 4.5.

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 parameterised types: it doesn't know that one is a Cell⟨Number⟩ while another is a Cell⟨Cell⟨Number⟩⟩. 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 4.6). 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 4-4, obj($\text{Cell}_{10}$) has two constraints, Integer $\leq \text{V}_{\text{obj}(\text{Cell}_{10})}$ and Float $\leq \text{V}_{\text{obj}(\text{Cell}_{10})}$; equivalently, we say that obj($\text{Cell}_{10}$) has the union type {Integer, Float}.

If the program being analysed uses generic types in a nested fashion, such as Cell⟨Cell⟨Float⟩⟩,[1] 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.

## 4.3 Example

We illustrate the algorithm with a code fragment from Figure 2-3:

[1]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) [GJSB00].

| Local | POINTS-TO set |
|---|---|
| c1 | {obj($\text{Cell}_2$)} |
| c2 | {obj($\text{Cell}_3$)} |
| c3 | {obj($\text{Cell}_2$)} |
| c4 | {obj($\text{Cell}_{10}$)} |
| f | {obj(Float)} |

Figure 4-2: POINTS-TO sets for local variables in the example of Figure 4-3

```
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 labelled $\texttt{Cell}_2$, $\texttt{Cell}_3$, and $\texttt{Cell}_{10}$, and their types are obj($\text{Cell}_2$), obj($\text{Cell}_3$), and obj($\text{Cell}_{10}$); subscripts refer to line numbers in the code. obj(Float) and obj(Integer) are not numbered: Float and Integer are not generic classes, so all of their instances are considered identical.

Figures 4-2–4-4 demonstrate the operation of the allocation type inference algorithm.

The first step is pointer analysis. Figure 4-2 shows the POINTS-TO sets (the output of the pointer analysis) for local variables, and Figure 4-3 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 $\texttt{Cell}_{10}\texttt{.value}$ 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 4-3 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 $rhs \leq lhs$'. Section 4.5 discusses s-unification for this example in more detail.

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

Figure 4-3: Example of s-unification, for lines 2–5 and 10–11 of Figure 2-3. 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 4.5) of obj($\texttt{Cell}_2$) is taken where indicated by the asterisk (*). Subscripts on arrows indicate the line number in Figure 4-5 at which the recursive call appears or the constraint is added.

---

The third step is resolution of the s-unification type constraints. Figure 4-4 illustrates this process. S-unification produced two different constraints for $V_{obj(\texttt{Cell}_{10})}$ — obj(Integer) $\leq$ V and obj(Float) $\leq$ V — so we represent the type of $V_{obj(\texttt{Cell}_{10})}$ by the union type {obj(Float), obj(Integer)}. Union types may be eliminated (Section 5.5) by selecting a most precise JSR-14 type that is a supertype of this union — in this case, it would be V ≡ Number, resulting in the type Cell⟨Number⟩ for obj($\texttt{Cell}_{10}$) — but this step is not required as union types may be passed on to the next phase of the algorithm.

| S-unification constraints | LBOUNDS values |
|---|---|
| $obj(Float) \leq V_{obj(Cell_2)}$<br>$obj(Cell_2) \leq V_{obj(Cell_3)}$<br>$obj(Integer) \leq V_{obj(Cell_{10})}$<br>$obj(Float) \leq V_{obj(Cell_{10})}$ | $V_{obj(Cell_2)} = \{obj(Float)\}$<br>$V_{obj(Cell_3)} = \{obj(Cell_2)\}$<br>$V_{obj(Cell_{10})} = \{obj(Integer), obj(Float)\}$ |
| **Resolved types** | **JSR-14 types** |
| $obj(Cell_2) = Cell\langle Float\rangle$<br>$obj(Cell_3) = Cell\langle Cell\langle Float\rangle\rangle$<br>$obj(Cell_{10}) = Cell\langle\{Integer, Float\}\rangle$ | $obj(Cell_2) = Cell\langle Float\rangle$<br>$obj(Cell_3) = Cell\langle Cell\langle Float\rangle\rangle$<br>$obj(Cell_{10}) = Cell\langle Number\rangle$ |

Figure 4-4: Resolution of s-unification constraints. The first cell shows the constraints arising from the S-UNIFY calls of Figure 4-3. The second cell shows the equivalent union types; note that $obj(Cell_3)$ depends on the type of $obj(Cell_2)$. The third cell shows the final allocation-site types after type resolution. The fourth cell shows what the result would be, if union types were eliminated at this stage.

## 4.4 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 [CU89, CUL89, BS96, TP00], construction of precise call graphs [DGC95, GC01], and static elimination of casts [WS01].

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. There are two principal kinds of context sensitivity, *call* and *data*, and our pointer analysis employs them both. Together, these enable it to distinguish effectively between 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 analyse a method more than once depending on where it was called from or what values were passed to it. Each

specialised 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, it must be analysed 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 (in which case all results that depend on the contour analysis must be updated if the analysis results change).

*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' [WS01]. 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 [Age95]. We briefly explain that algorithm, then explain our variation on it.

The Cartesian Product Algorithm (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 analysed.

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 analysed polymorphically. Thus, only generic methods, and methods of generic classes, may be analysed polymorphically. We do not employ a limit-based megamorphic threshold.

For example, `Cell.set(V)` may be analysed 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 analysed context-sensitively with respect to parameter `x`, but not `y`.

A few heavily-used non-generic methods, such as `System.arraycopy` and `Object.clone`, 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 analysed 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 [WS01]. Their framework permits use of different contour-selection functions and data context-sensitivity functions (such as their DCPA [WS01]); 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 POINTS-TO 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 parameterised 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.

## 4.5 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 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, such that the right operand is a subtype of the left operand; s-unification is therefore asymmetric. S-unification also differs in that it is a work-

```
1  // S-UNIFY unifies lhs with rhs, in the process constraining, in
2  //    LBOUNDS, the type variables of context so that rhs ≤ lhs.
3  // context is an allocation site of a generic class C.
4  // lhs is the type of a JSR-14 declaration appearing within class
5  //    C, typically containing free type variables of C.
6  // rhs is a type, typically a union of obj(Ci) types denoting a
7  //    POINTS-TO-set; it never contains free type variables.
8  procedure S-UNIFY(context, lhs, rhs)
9    if lhs has no free type variables then
10     return
11   // First, switch based on rhs
12   if rhs = Null then
13     return
14   else if rhs = {τ1,...,τn} then // Union type
15     for all τi ∈ rhs do
16       S-UNIFY(context, lhs, τi)
17     return
18   // Second, switch based on lhs
19   if lhs = T then // Type variable
20     if T is declared by a generic method then
21       for all b ∈ BOUNDS(T) do
22         S-UNIFY(context, b, rhs)
23       return
24     let tclass := the class that declares T
25     if tclass ≠ CLASS(context) then
26       let lhs' := instantiation expression of T in CLASS(context)
27       S-UNIFY(context, lhs', rhs)
28       return
29     LBOUNDS(context, lhs) := LBOUNDS(context, lhs) ∪ {rhs}
30     if LBOUNDS changed then
31       for all (c, l, r) ∈ REUNIFY | r = lhs do
32         S-UNIFY(c, l, r)
33     return
34   else if lhs = C⟨τ1, ..., τn⟩ then // Class type
35     if rhs = D⟨τ'1, ..., τ'm⟩ then
36       let rhs' := WIDEN(rhs, C) // rhs' = C⟨τ''1, ..., τ''n⟩
37       for 1 ≤ i ≤ n do
38         S-UNIFY(context, τi, τ''i)
39       return
40     else if rhs = obj(Ci) then
41       REUNIFY := REUNIFY ∪ {(context, lhs, rhs)}
42       S-UNIFY(context, lhs, SNAPSHOT(rhs))
43       return
44     else
45       error: This cannot happen
```

Figure 4-5: S-unification algorithm. Refer to Figure 4-6 for helper definitions.

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

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

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

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

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

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

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

Figure 4-6: S-unification helper definitions

list algorithm: as new information becomes available, it may be necessary to repeat some s-unifications.

S-unification is performed by the S-UNIFY procedure of Figure 4-5. It can be thought of as inducing the subtype constraint $rhs \leq lhs$ resulting from the Java assignment '*lhs* = *rhs*;'. The three parameters of the S-UNIFY 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 POINTS-TO set—that is, a union of allocation site types—for declaration *lhs* inferred by the pointer analysis of Section 4.4. Figure 4-6 lists several helper definitions used by the s-unification algorithm.

S-unification infers, for each type variable T of each distinct allocation site type $\mathsf{obj}(C_i)$, 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 LBOUNDS(*context*, *typevar*), where *context* is an allocation site type, and *typevar* is a type variable belonging to the class of the allocation. ($V_{obj(Cell_{10})}$ is shorthand for LBOUNDS(obj($Cell_{10}$), V).) All LBOUNDS are initialised to the empty union type, and types are added to them as s-unification proceeds.

After the pointer analysis of Section 4.4 is complete, S-UNIFY 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 S-UNIFY is called once for each. The information in Figure 4-3 is therefore data context-sensitive. In these calls to S-UNIFY, the *context* argument is the allocation site type, *lhs* is the declared type of the field, and *rhs* is the POINTS-TO set of the field. (See the last three rows of Figure 4-3.) In contrast, a single abstraction is used for all instances of Float, since it is non-generic (S-UNIFY 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). S-UNIFY is called once for each formal parameter and return parameter at each such call. The information in Figure 4-3 is therefore call context-sensitive. In these calls to S-UNIFY, the *context* argument is the allocation site type of the receiver expression (in our example it is the sole element of POINTS-TO(`this`)), *lhs* is the declared type of the method parameter, and *rhs* is the POINTS-TO 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 4.4 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 4-3.

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

In the second line of the `replaceValue` s-unification call, indicated by the asterisk (*) in Figure 4-3, S-UNIFY must unify Cell⟨U⟩ with obj($\text{Cell}_2$). However, the type of obj($\text{Cell}_2$) is not yet known—the goal of allocation type inference is to determine constraints on the obj types. To permit unification to proceed, S-UNIFY uses, in place of obj($\text{Cell}_2$), a *snapshot*: the type implied by its current constraints. In this case, because the only constraint on $\text{Cell}_2$ is obj(Float) $\leq$ $\text{V}_{\text{obj}(\text{Cell}_2)}$, the snapshot is Cell⟨{obj(Float)}⟩. If subsequent unifications add any new constraints on obj($\text{Cell}_2$), then the snapshot changes and the unification must be re-performed. Re-unification is not necessary in our example.

As can be seen from the duplicated entries in the S-unifications column of Figure 4-3, 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 POINTS-TO 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 POINTS-TO 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.)

### 4.5.1 S-unification algorithm details

This section discusses the S-UNIFY algorithm presented in Figure 4-5 on page 59. 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 4-5.

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 $\texttt{Cell}_{10}\texttt{.value}$ in Figure 4-3), 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 (parameterised) 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.[2]

The call to `replaceValue` gives a concrete example of a unification against a method variable. In the fourth call to S-UNIFY (see Figure 4-3), *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

[2]Some care must be exercised to prevent infinite recursion in the presence of F-bounded variables such as `T extends Comparable<T>`; for simplicity, this is not shown in the algorithm of Figure 4-5.

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($\text{Cell}_2$), 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. We consider these two cases in turn.

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 $\texttt{Cell}_{10}\texttt{.replaceValue(•)}$, the widening is the identity operation since the classes of *lhs* and *rhs* already match: they are both Cell.

If *rhs* is an allocation type (lines 40–43), then it 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 it is 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 $r = o$.

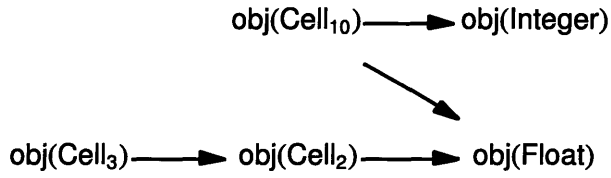
Since the process of s-unification is idempotent with respect to the same argument values, and monotonic with respect to larger *rhs* argument values, this is sound.

## 4.6 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 4-4. 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:




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 [CW85] could assign a type such as **fix** $\lambda x$.Cell$\langle x \rangle$. However, JSR-14

has no means of expressing such 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.

# Chapter 5

# Declaration Type Inference

The allocation type inference produces a precise parameterised 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 most specific type possible to each declaration.

The consistency requirement is enforced by type constraints [PS91], 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 the behaviour 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 ideally like to obtain a minimal solu-

$$
\begin{array}{lll}
\textit{class} & ::= & \texttt{class}\ C\langle T_1 \triangleleft S_1, \ldots, T_n \triangleleft S_n\rangle \ \triangleleft\ B\langle \tau_1, \ldots, \tau_m\rangle \\
& & \quad \{\, \textit{field} * \ \textit{method} * \,\} \\
\\
\textit{field} & ::= & \tau\ f; \\
\\
\textit{method} & ::= & \langle T_1 \triangleleft S_1, \ldots, T_n \triangleleft S_n\rangle\ \tau\ m(\tau_1\ x_1, \ldots, \tau_m\ x_m) \\
& & \quad \{\, \textit{stmt} * \,\} \\
\\
\textit{stmt} & ::= & \textit{expr}_l := \textit{expr}_r; \\
& & \mid \texttt{return}\ \textit{expr}; \\
& & \mid C\ \textit{var}; \\
\\
\textit{expr} & ::= & \texttt{this} \\
& & \mid \texttt{null} \\
& & \mid \textit{var} \\
& & \mid \textit{expr}.f \\
& & \mid \texttt{new}\ C \\
& & \mid \textit{expr}.m(\textit{expr}_1, \ldots, \textit{expr}_n) \\
& & \mid (C)\ \textit{expr}
\end{array}
$$

Figure 5-1: Grammar of a core subset of JSR-14. Note that '$\triangleleft$' is an abbreviation for `extends/implements`.

tion to the system of type constraints, if possible; however, as we have seen, there may be no unique minimal solution, so we have to content ourselves with solutions composed of local minima.

Sections 5.1 and 5.2 discuss the form of the type constraints. Section 5.3 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 5.4 and 5.5 how the system of type constraints can be solved to obtain a translated program.

For clarity, we illustrate the generation of type constraints for a core subset of the features of the JSR-14 language as shown in Figure 5-1. The ideas can be extended naturally to support all features of the real language, as in our implementation.

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

Figure 5-2: Type grammar for declaration type inference

## 5.1 Type Constraints

A constraint $\alpha_1 \xrightarrow{R} \alpha_2$ is a manifestation of a relationship $R$ between two terms $\alpha_1$ and $\alpha_2$. A constraint is *satisfied* if and only if the pair $(\alpha_1, \alpha_2)$ 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 [PS91] 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 $\tau$, shown in Figure 5-2. This grammar modifies the type grammar of Figure 4-1 by removing allocation site types and adding 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 $\alpha_2 \leq \alpha_1$.

Considered as a whole, our algorithm contains three different constraint systems (described in Sections 4.4, 4.5, and this chapter), because different parts of the algorithm have different purposes and require different technical machinery. The

pointer analysis (Section 4.4) is context-sensitive for precision in computing value flow; we adopt the constraints directly from previous work [WS01]. By contrast, the results of declaration type inference (this chapter) must satisfy the type-checker, which is context-insensitive, so that constraint system is most naturally context-insensitive. The s-unification constraints (Section 4.5) 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.

## 5.2 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 *parameterised 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 parameterised 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 $C'$.

$[\mathsf{T} := \tau]$ denotes the substitution of the type variable T with type (or type unknown) $\tau$. Substitutions are denoted by the metavariable $\theta$. We denote the empty substitution with $\emptyset$, the composition of two substitutions with $\theta \circ \theta'$, and the application of substitution $\theta$ to type $\tau$ with $\tau\theta$. The result of substitution applica-

Suppose we have declarations:

$\texttt{class}\ B\langle \mathsf{U}_1, \ldots, \mathsf{U}_i \rangle$

$\texttt{class}\ C\langle \mathsf{T}_1 \triangleleft S_1, \ldots, \mathsf{T}_n \triangleleft S_n \rangle \triangleleft B\langle \tau_1, \ldots, \tau_i \rangle$

where '$\triangleleft$' is an abbreviation for `extends/implements`

T and U are type variables

$S$ and $\tau$ are types

Then:

$\text{WIDENING}(C, C) = \emptyset$

$\text{WIDENING}(C, A) = [\mathsf{U}_1 := \tau_1, \ldots, \mathsf{U}_i := \tau_i] \circ \text{WIDENING}(B, A)$
where $A \neq C$

$\text{FRESH-SUBST}(\langle \mathsf{T}_1 \triangleleft S_1, \ldots, \mathsf{T}_n \triangleleft S_n \rangle) = \theta$
where $\theta = [\mathsf{T}_1 := X_1, \ldots, \mathsf{T}_n := X_n]$
$X_i$ are fresh
generates constraints $X_i \leq S_i\,\theta$

$\text{ELABORATE}(C) = C\langle \mathsf{T}_1, \ldots, \mathsf{T}_n \rangle\,\theta$
where $\theta = \text{FRESH-SUBST}(\langle \mathsf{T}_1 \triangleleft S_1, \ldots, \mathsf{T}_n \triangleleft S_n \rangle)$

$\text{SUBSTITUTION}(C\langle \tau_1, \ldots, \tau_i \rangle) = [\mathsf{T}_1 := \tau_1, \ldots, \mathsf{T}_i := \tau_i]$

$\text{RECEIVER}(E.x) = \text{SUBSTITUTION}([\![E]\!]) \circ \text{WIDENING}(C, A)$
where class $A$ declares member x

$\text{ERASURE}(C\langle \tau_1, \ldots, \tau_n \rangle) = C$

$\text{ERASURE}(\mathsf{T}_i) = \text{ERASURE}(S_i)$

Figure 5-3: Auxiliary definitions for declaration type inference

tion is a type (or a type unknown). For example, given class `Pair` of Figure 2-2, $\mathsf{Pair}\langle \mathsf{F}, \mathsf{S} \rangle[\mathsf{F} := X_1, \mathsf{S} := X_2] = \mathsf{Pair}\langle X_1, X_2 \rangle$.

Figure 5-3 defines auxiliary functions used by the analysis. The WIDENING function defines the widening conversion [GJSB00] of (generic) types: it indicates which instantiation of a superclass is a supertype of a given instantiation of a subclass.[1] For example, in the context of types shown in Figures 2-1 and 2-2,

[1] WIDENING is somewhat similar to WIDEN defined in the previous chapter. Whereas WIDENING defines the result of widening a parameterised type, WIDEN specifies the corresponding `extends`-clause substitution; i.e., $\text{SUBSTITUTION}(\text{WIDEN}(D\langle \tau_1, \ldots, \tau_n \rangle, C)) = \text{SUBSTITUTION}(D\langle \tau_1, \ldots, \tau_n \rangle) \circ \text{WIDENING}(D, C)$.

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 FRESH-SUBST function creates a fresh substitution of a given generic (class or method) declaration, in which each type variable of the declaration is replaced by a fresh type unknown. In addition, FRESH-SUBST generates type constraints to ensure that each fresh type unknown is within its bound. Since the bounds of a variable may themselves refer to type variables (such a variable is called *F-bounded* [CCH+89]), the substitution is applied to the bounds when generating these constraints.

The ELABORATE function takes a class type $C$ and returns a fresh 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. It uses FRESH-SUBST to create the type unknowns and add the required bounds constraints. 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 SUBSTITUTION function returns the instantiation of a parameterised type such as $C\langle \tau_1, \ldots, \tau_n \rangle$ in the form of a substitution of the type variables of $C$ for the type arguments $\tau_i$.

The RECEIVER function defines the substitution applied to the type of class instance members (fields or methods) due to the parameterised type of the receiver 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 2-4, 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 parameterisation 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 returns the erased [BOSW98b, Jav01] version of a parameterised type; the erasure of a type variable is the erasure of its bound.

## 5.3 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 2-2 (page 26) and the statement `Pair p = new Pair(f, o)` on lines 7–8 of Figure 2-3 (page 30). The declaration type inference creates four fresh type unknowns $X_1$–$X_4$, 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 of these constraints capture the flow of values through the program. Others are required to preserve the behaviour of the program during translation, by preserving the program's erasure, and ensuring that the method-overriding relationship is not altered when the types of methods change. Generation of these constraints is explained in Sections 5.3.1 and 5.3.2.

Some type constraints are unconditional. 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; conditional constraints are explained in Section 5.3.3.

The generation of constraints for casts and allocation sites are explained in Sections 5.3.4 and 5.3.5.

| program construct | implied type constraint(s) | |
|---|---|---|
| statement $E_1 := E_2$; | $[\![E_2]\!] \leq [\![E_1]\!]$ | 1 |
| statement `return` $E$; (in method $M$) | $[\![E]\!] \leq \mathit{Return}(M)$ | 2 |
| statement $C$ *var*; | $\mathit{Local}(var) \triangleq X_{var}$ ($X_{var}$ is fresh) | 3 |
| | $X_{var} \leq C$ | 4 |
| expression `this` (in class $B$) | $[\![\texttt{this}]\!] \triangleq B$ | 5 |
| expression `null` | $[\![\texttt{null}]\!] \triangleq \mathsf{Null}$ | 6 |
| expression *var* | $[\![var]\!] \triangleq \mathit{Local}(var)$ | 7 |
| expression `new` $C$ | $[\![\texttt{new } C]\!] \triangleq \text{ELABORATE}(C)$ | 8 |
| expression $E.f$ (field $F$) | $[\![E.f]\!] \triangleq \mathit{Field}(F)\ \theta$ | 9 |
| $\theta = \text{RECEIVER}(E.f)$ | | |
| expression $E.m(E_1, \ldots, E_n)$ (method $M$) | $[\![E.m(E_1, \ldots, E_n)]\!] \triangleq \mathit{Return}(M)\ \theta\ \theta_{fresh}$ | 10 |
| $\theta = \text{RECEIVER}(E.m)$ | $\forall i.\ [\![E_i]\!] \leq \mathit{Param}(M, i)\ \theta\ \theta_{fresh}$ | 11 |
| $M$ has type variables $\langle T_1 \triangleleft S_1, \ldots, T_k \triangleleft S_k \rangle$, $k \geq 0$ | | |
| $\theta_{fresh} = \text{FRESH-SUBST}(\langle T_1 \triangleleft S_1, \ldots, T_k \triangleleft S_k \rangle)$ | | |
| expression $(C)$ *expr* | $[\![(C)\ expr]\!] \triangleq X_{out}$ | 12 |
| (fresh $X_{in}, X_{out}$) | $[\![expr]\!] \leq X_{in}$ | 13 |
| | $X_{in} \lessdot_C X_{out}$ | 14 |
| method $M$ overrides method $M'$ | $\forall i.\ \mathit{Param}(M', i) = \mathit{Param}(M, i)$ | 15 |
| | $\mathit{Return}(M) \leq \mathit{Return}(M')$ | 16 |
| method $M$ is defined in library code as: | $\mathit{Return}(M) \triangleq \tau$ | 17 |
| $\langle T_1 \triangleleft S_1, \ldots, T_n \triangleleft S_n \rangle\ \tau\ M(\tau_1\ x_1, \ldots, \tau_n\ x_n)$ | $\forall i.\ \mathit{Param}(M, i) \triangleq \tau_i$ | 18 |
| method $M$ is defined in client code as: | $\mathit{Return}(M) \triangleq X$ | 19 |
| $\tau\ M(\tau_1\ x_1, \ldots, \tau_n\ x_n)$ | $\forall i.\ \mathit{Param}(M, i) \triangleq X_i$ | 20 |
| (fresh $X, X_i\ \forall i$) | $\forall i.\ \mathit{Local}(x_i) \triangleq \mathit{Param}(M, i)$ | 21 |
| | $\text{ELABORATE}(\tau) \leq X \leq \tau$ | 22 |
| | $\forall i.\ \text{ELABORATE}(\tau_i) \leq X_i \leq \tau_i$ | 23 |
| field $F$ is defined in library code as: $\tau\ F$ | $\mathit{Field}(F) \triangleq \tau$ | 24 |
| field $F$ is defined in client code as: $\tau\ F$ | $\mathit{Field}(F) \triangleq X_F$ | 25 |
| | $\text{ELABORATE}(\tau) \leq X_F \leq \tau$ | 26 |

Figure 5-4: Type constraints for key features of JSR-14. Meta-syntactic functions such as $[\![-]\!]$ (for expressions) and $\mathit{Field}(F)$ are defined using the notation $lhs \triangleq (\ldots)$. The generation of constraints is explained in Section 5.3.1. The three sections of the table show the constraints generated for statements, expressions and other declarations, respectively. The cast constraint $\lessdot_C$ is defined in Section 5.3.4.

### 5.3.1 Ordinary type constraints

Figure 5-4 shows type constraints induced by the key features of JSR-14. To cover the entire language, additional constraints are required for exceptions, `instanceof` expressions, arrays, 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 [TKB03].

Constraint generation is achieved by descent over the syntax of all the method bodies within the client code. Figure 5-4 defines the meta-syntactic function $[\![\cdot]\!]$, pronounced 'type of', which maps from expression syntax to types, generating constraints as a side effect. The figure also defines four other meta-functions, *Field*, *Param*, *Return* and *Local*, for the types of fields, method parameters and results, and local variables.

For each row whose 'program construct' table entry contains an expression $E$, the corresponding 'constraints' table entry includes a term of the form $[\![E]\!] \triangleq (\ldots)$. Such terms are not constraints, but together constitute the definition of $[\![\cdot]\!]$.

We now discuss the rules of Figure 5-4 in detail.

Rules **(1)–(3)** define constraint generation for program statements; there are three kinds of statement in Figure 5-1: assignment statements, return statements, and local variable declaration statements.

Rule **(1)** states that, in a valid JSR-14 program, the type of the right-hand side of an assignment must be a subtype of the left-hand side. Rule **(2)** says that the type of the returned expression must be a subtype of the result type of the method in which the `return` statement appears. *Return*(*M*) stands for the declared type of the result of method *M in the translated program*, and is defined for methods in client code by rule **(19)**. Rule **(3)**, for local variable declaration statements, partially defines the meta-function *Local*, which returns the analysis type for each local variable *var*. (For simplicity, local variable names are assumed to be globally unique.)

Rules **(5)–(14)** define constraint generation for program expressions; Figure 5-1 defines the following kinds of expression: `this` reference, `null` literal, local variable reference, instance field reference, object allocation, instance method invocation, and cast expression.

Rule **(5)** defines the type of `this` to be the type of the class in which it appears, and rule **(6)** defines the type of the `null` expression as Null. Rule **(7)** defines the type of a reference to a local variable, using the meta-function *Local*(*var*): if the local is a formal parameter (rule **(21)**), then this is an elaboration of the parameter

type; for locals declared in the method body (rule **(3)**), this is a fresh type unknown.

Rule **(8)** defines the type of an allocation expression as a fresh elaboration of the allocated class. (There are no constructors in our simplified grammar, but they can be modelled using the same approach as for ordinary methods.) Additional constraints on the fresh type unknowns will be created to incorporate the results of allocation type inference into the system; these are described in Section 5.3.5.

Rule **(9)** deals with accesses to fields and defines the type of a reference to field $F$ to be the declared type of the field in the translated program, $Field(F)$, with the RECEIVER substitution applied: if the declaring class of $F$ is generic, then the free type variables in the field type are substituted by the type arguments of the receiver expression. $Field(F)$ is defined by rule **(24)**, if $F$ is declared by library code, or by rule **(25)** otherwise.

Rules **(10)** and **(11)** deal with method invocations. Rule **(10)** defines the type of the method invocation expression as the return type of the method. Rule **(11)** specifies that the type of each actual parameter must be a subtype of the corresponding formal parameter. Just as with accesses to fields, accesses to methods are subject to a RECEIVER substitution if the method's declaring class is generic. In addition, if method itself is generic, the substitution $\theta_{fresh}$ binds fresh names to the method's type variables for this call only; additionally, FRESH-SUBST generates constraints to ensure that the method type arguments are within their declared bounds.

As an example, consider line 11 of Figure 2-3 on page 30: `c4.replaceValue(c1)`, where `c1`'s declaration, elaborated by the introduction of a type unknown, is $\mathsf{Cell}\langle X_1\rangle$ and the declaration of `c4` is elaborated to $\mathsf{Cell}\langle X_4\rangle$. Thus, we have that:

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

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

- $[\![\texttt{c1}]\!] \leq \mathsf{Cell}\langle \mathsf{U} \rangle[\mathsf{V} := X_4][\mathsf{U} := X_U]$

  i.e., $\mathsf{Cell}\langle X_1 \rangle \leq \mathsf{Cell}\langle X_U \rangle$
- $X_\mathsf{U} \leq X_4$

Rules **(12)**–**(14)**, for cast expressions, are explained in detail in Section 5.3.4.

Rules **(15)** and **(16)** preserve the semantics of method overriding relationships. Rule **(15)** enforces the invariance of type arguments in overriding methods; in other words, the formal parameter types in an overriding method method must exactly match those of the method which they override; this is required to maintain the existing virtual dispatch behaviour. Another new feature of JSR-14 is that it permits covariant specialisation of method return types. Therefore, in contrast to the equality constraint required for formal parameters, rule **(16)** generates only a subtype constraint over the methods' result types.

Rules **(17)** and **(18)** define the meta-syntactic functions *Return*(*M*) and *Param*(*M*, *i*) for methods defined in library code, while rules **(19)** and **(20)** do so for client code. For library code, these expressions are simply equal to the declared types of the specified formal parameter or method result. Parameter types and return types declared within client code, however, may be specialised by the analysis, so these parameter declarations are elaborated with fresh type unknowns. Rules **(24)** and **(25)** define the types of fields, and are analogous to the rules for method parameters. Rule **(21)** defines the meta-function *Local* for formal parameters.

Note the similarity between the constraints for allocation of a generic class (rule **(8)**) and those for the invocation a generic method (rules **(10)** and **(11)**): both instantiate the type variables of the generic declaration by a tuple of fresh unknowns, by calling FRESH-SUBST. (In the allocation site case, the elaboration of the generic class *C*, is performed via ELABORATE(*C*).)

(In many ways, each generic method declaration can be considered mere syntactic sugar for a declaration of a generic inner class, a temporary instance of which is created for each method call. For example, the method `Cell.replaceValue` in Figure 2-2, and the call to it on line 11 of Figure 2-4 could be equivalently re-written:

```
class Cell<V extends Object> {
    ...
    class TypeClosure<U extends V> {
        void replaceValue(Cell<U> that) { ... }
    }
}
// c4.replaceValue(c1);
c4.new TypeClosure<Float>().replaceValue(c1);
```

This isomorphism is the type-domain analogue of the isomorphism between explicit records and lexical closures in the value domain. In languages with block structure, the packaging of lexically-enclosing values into a closure is done automatically, and these values appear within the scope of the function body; in other languages, a record value must be created explicitly, and elements of this record are then referenced from within the function body. Similarly, JSR-14's generic methods act like closures in the type domain: without them, a type closure must be explicitly created, and then the type variables of the closure (e.g. U) referenced from within the generic method body.)

### 5.3.2 Framing constraints

In order to ensure that the erasure of the program is preserved, and thereby its semantics also (see principles presented in Chapter 3), we introduce 'framing' constraints. Framing constraints restrict the set of possible solutions to those in which the framed declarations have the same erasure as in the unmodified program.

For example, to frame the declaration of a method parameter `f(Cell c)`, we add two constraints, shown in rule **(23)**, one enforcing an upper bound and the other a lower bound: $\mathsf{Cell}\langle X_1\rangle \leq [\![\mathtt{c}]\!] \leq \mathsf{Cell}$ (the left-hand side is ELABORATE(Cell), so $X_1$ is fresh). The inferred type of $[\![\mathtt{c}]\!]$ will be equal to either its lower bound or its upper bound, since there are no types in between; in the latter case, the value assigned to $X_1$ is immaterial. Clearly, the only solutions for $[\![\mathtt{c}]\!]$ are those in which its erasure is Cell.

Within client code, method parameters, method results, and fields are all framed in this manner, as shown in rules **(23)**, **(22)** and **(26)**. On the other hand, local variables do not require such strict constraints, as their erasure does not contribute to the semantics of the program. Therefore rule **(4)** frames local variables only from above. We never infer a type less specific than that in the original program, but the solution may freely infer a more specialised type.

### 5.3.3 Guarded type constraints

Generic instantiations are of two kinds: parameterised types and raw types. For parameterised 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 elaboration of the type of a program 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, Object x) {
   x = c.get();
   c.set("foo");
}
```

If the declaration of c is parameterised (say, Cell$\langle X_1 \rangle$), then the constraint $X_1 \leq [\![\texttt{x}]\!]$ must be satisfied (rules **(1)** and **(10)** in Figure 5-4). On the other hand, if the declaration is raw, then the constraint ERASURE(*Return*(`Cell.get`)) = Object $\leq [\![\texttt{x}]\!]$ must be satisfied. Similar constraints arise from the call to `set`: if the declaration is parameterised, then $[\![\texttt{"foo"}]\!] \leq X_1$; otherwise, $[\![\texttt{"foo"}]\!] \leq$ 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 'parameterisedness' 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 $\mathsf{Object} \leq_{\neg \mathtt{c}} [\![\mathtt{x}]\!]$ (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.

For correct generation of guarded constraints, the definitions of *Param*, *Result* and *Field* must be modified slightly. As presented, they simply create and return a fresh elaboration of the type of the specified location (field or parameter), e.g. $\mathit{Param}(\mathtt{displayValue}, 1) \triangleq \mathsf{Cell}\langle X_1 \rangle$. Because ELABORATE for a generic type always returns a parameterised type, this precludes the possibility of these locations from ever having a raw type.

To solve this problem, rather than define the meta-functions to return a fresh elaboration *elab*, we modify them to return a new type unknown that is *guardedly less than elab*, where the condition or guard is the parameterisedness of *elab* itself. That is, $\mathit{Param}(\mathtt{displayValue}, 1) \triangleq X_2$ where $X_2 \leq_{\mathsf{Cell}\langle X_1 \rangle} \mathsf{Cell}\langle X_1 \rangle$.

The effect of this guarded constraint is to permit $X_2$ to be assigned a raw type (e.g. Cell), in which case the guarded edge is ignored. However, in a solution in which $X_1$ is not killed, then the guarded edge will be honoured as $X_2 \leq \mathsf{Cell}\langle X_1 \rangle$, and $X_2$ will be assigned a parameterised type. (The inversely-guarded conditional constraint is $X_2 \leq_{\neg \mathsf{Cell}\langle X_1 \rangle} \mathsf{Cell}$; however, this constraint is subsumed by the unconditional upper framing constraint for $X_2$, so it is not required.)

Similarly, rule **(3)** is modified so that the fresh type unknown created to represent the type of a local variable is guardedly less than the elaborated type of its declaration, $C$.

### 5.3.4 Cast constraints

Rules **(12)–(14)** generate constraints for casts, and introduce a new kind of constraint, $\alpha_1 \lessdot_C \alpha_2$. Cast constraints are more complex than ordinary $\leq$-constraints; this complexity derives from the complex rules for type-checking cast expressions in JSR-14. We will review the type-checking rules, and then describe the behaviour of cast constraints.

Semantically, a JSR-14 cast expression $(C)$ *expr* or $(C\langle\tau_1,\ldots,\tau_n\rangle)$ *expr* involves a test of the class of the value of *expr*, equivalent to *expr* `instanceof` $C$. If this condition is false, an exception is thrown; if true, the cast expression has the same value as *expr*, but its type becomes the *target* type (in parentheses). Casts can thus be thought of as class-based 'filters' of values.

Because JSR-14 uses homogeneous translation by type erasure, the type of a value cannot be determined at run-time, and it is impossible for the cast mechanism to distinguish between values of differently-instantiated types arising from the same class. For example, one cannot write `(Cell<String>) c` to determine whether `c` refers to a Cell whose element is a String.

Therefore, the language restricts the use of parameterised types in cast expressions to situations in which the correctness of the type arguments in the target type can be statically proven; in other words, the success of the dynamic `instanceof` test must imply the correctness of the given parameterisation. For example, the parameterised cast expression `(ArrayList<String>) ls`, where `ls` is of type List⟨String⟩, is valid, because ArrayList⟨String⟩ is the only subtype of List⟨String⟩ whose erasure is ArrayList, so the `instanceof` class-test is sufficient. In contrast, the parameterised cast `(Pair<String, Integer>) c`, where `c` has type Cell⟨String⟩, is not legal, since there are an infinite number of Pair subtypes of Cell⟨String⟩. Formally, a parameterised cast $(C\langle\tau_1,\ldots,\tau_n\rangle)$ *expr*) is legal if and only if the extends-clause substitution $\text{WIDENING}(C, B)$ is injective, where $B$ is the erasure of $[\![expr]\!]$.

The constraints generated for the cast operator reflect the semantics: for each cast, we create two fresh type unknowns, $X_{in}$ for the input and $X_{out}$ for the result,

connected by a $X_{in} \lessdot_C X_{out}$ edge, which acts as a class-based filter. For each type $\tau$ that reaches $X_{in}$, the *intersection* type is considered to reach $X_{out}$, as defined by the following cases:

1. If the target type $C$ and the reaching type $\tau$ are disjoint, then the intersection is empty;
2. If $\tau \leq C$ then the intersection is $\tau$;
3. If $\tau$ has a unique subtype $\tau_C$ whose erasure is $C$, then the intersection is $\tau_C$;
4. Otherwise, the intersection is raw $C$.

Additionally, framing constraints (Section 5.3.2) are applied to the cast output node to preserve its erasure. A degenerate case occurs when the target class $C$ of a cast is non-generic, in which case the framing constraints alone uniquely determine the type of the cast; the $\lessdot_C$-constraint has no additional effect.

Interestingly, we chose to ignore one type-checking rule applicable to both non-generic Java and JSR-14 when generating type constraints: a cast `(D) c` is illegal if `c` has type $C$, and $C$ and $D$ are unrelated *classes* (not interfaces). Such casts are doomed to fail, so the type-checker does not admit them. This code presents a very simple example, showing such a cast before and after the translation:

```
Object o = new Integer(1);        Integer o = new Integer(1);
String s = (String) o;            String s = (String) o; // ERROR
```

Rather than add constraints to prevent such solutions from arising, we allow them to occur, but we report them as errors, as they are invariably evidence of a bug in the original program.

### 5.3.5 Allocation Types

For soundness, the types of allocation sites must be consistent with the types inferred by the allocation type inference of Chapter 4. The most straightforward way to incorporate the results of allocation type inference into the constraint system

is simply to define the type arguments of each generic allocation site (as used in rule **(8)**) to be exactly equal to the type inferred for it.

This is simple and easy to implement (and is effectively what our implementation does). However, it is over-constrained beyond what is necessary for correctness. A more flexible approach would be to constrain each of these unknowns to be a supertype of the corresponding parameter of the type resulting from allocation type inference. This approach permits choosing a less specific assignment for a type unknown, which may be desirable, as illustrated by Figure 3-1. This is especially true in the case when Null appears in the results of allocation type inference (for example, it reports the type of the empty container as Cell⟨Null⟩).

For example, allocation type inference reports the type Pair⟨Number, Boolean⟩ for the allocation on line 8 of Figure 2-4. The first approach would simply make this the type of the `new` expression. The second approach would instead make the type of the expression Pair$\langle X_1, X_2 \rangle$, where $X_1$ and $X_2$ are fresh type unknowns constrained in the following way: Number $\leq X_1$, Boolean $\leq X_2$.

## 5.4 Solving the type constraints

The algorithm of Section 5.3 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 behaviour 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.[2] 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 practise. See Chapter 7 for the results.

The solver constructs a graph, initially containing edges only for the unconditional constraints. The solver 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 parameterised (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 solver backtracks.

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

As the edges are added, closure rules are applied; see Section 5.4.1. For example, if the graph contains a path from Cell$\langle X \rangle$ to Cell$\langle Y \rangle$, then the interpretation of this path is Cell$\langle X \rangle \leq$ Cell$\langle Y \rangle$, 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.

Once the conditional edges have been added, if the search is trying to infer a parameterised type for guard variable $g$, then for each unknown $X_u$ belonging to $g$,

[2]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.

the solver computes the union of the types that reach $X_u$ through paths in the graph. This is the set of lower bounds on $X_u$, and the algorithm greedily assigns to $X_u$ the least upper bound of this union.

### 5.4.1 Closure rules

The following closure rule enforces invariant parametric subtyping for class types:

$$\frac{t \leq C\langle x_1, \ldots, x_i\rangle \qquad t \leq D\langle y_1, \ldots, y_j\rangle \qquad D \leq C}{\forall i.\ z_i = x_i \text{ where } C\langle z_1, \ldots, z_i\rangle = \textsc{widen}(D\langle y_1, \ldots, y_j\rangle, C)}$$

A common special case of this rule is when $t = D\langle y_1, \ldots, y_j\rangle$, in which case the antecedent degenerates to $D\langle y_1, \ldots, y_j\rangle \leq C\langle x_1, \ldots, x_i\rangle$.

An additional rule (not shown) enforces *co*variant subtyping for built-in array types using a similar approach.

### 5.4.2 Dependency graph

This section describes how to order the guard variables so as to minimise 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 in the type constraint graph. The set of edges consists of every ordinary edge (from the type constraint graph of Section 5.3.1), every guarded edge (from Section 5.3.3), 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 an optimisation, within a strongly connected component, we decide any nodes that

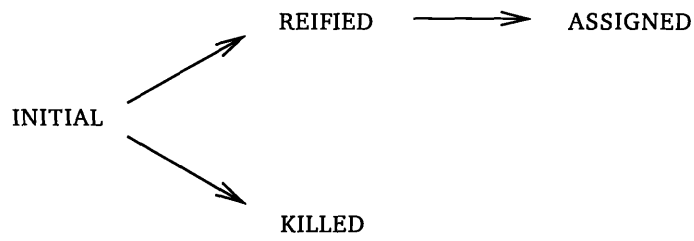



Figure 5-5: States in the declaration type inference algorithm for each type unknown. At each step, every type unknown is associated with a state: initially the INITIAL state, and at completion, either the ASSIGNED or KILLED state. Edges indicate the permitted transitions between states; only during backtracking is a previous state restored.

are guards for some constraints first, because such choices are likely to have the largest impact.

### 5.4.3 Deciding guards, assigning types

In the declaration type inference algorithm, each type unknown is in one of four states, illustrated in Figure 5-5. 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 solver. 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 solver decided to give a parameterised 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 solver adds the relevant conditional edges.

Finally, the ASSIGNED state means that the type is parameterised, and the type arguments have been decided upon. (The type arguments themselves are indicated by a separate table of assignments.) When the solver 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 5.5 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 solver 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 $X_2$; in the second, there is no constraining effect on $X_3$.

For an empty container, allocation type inference returns Null as its element type, 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.

### Null-elimination

After all the guards have been decided, any type unknowns that have no lower-bound constraints will be REIFIED but still unassigned. Conceptually, these unknowns could all be assigned the Null type, but of course this is not a valid type in Java source code. The solver has some freedom to choose type assignments for these unknowns, since any type that satisfies the upper-bound constraints is potentially suitable, and typically there is only a single such bound — usually Object, though it is sometimes a more specific type.

To type unknowns with a single upper bound, the solver simply assigns the upper bound type itself; this is easily achieved using the existing machinery by simply adding an additional constraint to make the upper bound a lower bound also, creating a cycle.

In rare cases where a type unknown $X$ has two (or more) distinct upper-bounds, the type assigned to it must be a subtype of both of them. In general, this is problematic, for the computation of *meets*, or greatest lower bounds, over parameterised types is even more complex than computing joins (which is the topic of Section 5.5). Our solver gets around this problem by computing a meet $M$ over the erased types of $X$'s upper bounds, using non-generic Java subtyping rules (the process is analagous to that for joins, below; it may involve an arbitrary choice). It then creates a fresh elaboration of $M$, and adds a subtype constraint $M \leq X$. The existing solving machinery is then able to infer assignments both for the type unknowns of $M$, and for $X$ itself.

This mechanism is thus able to infer the desired types (i.e., Cell⟨String⟩ throughout) for the following typical case:

```
Cell c1 = null;
Cell c2 = new Cell("foo");
c2 = c1;
```

Before Null-elimination, the solver determines that ⟦c1⟧ ≤ ⟦c2⟧ = Cell⟨String⟩, but the type unknown representing ⟦c1⟧ has no lower bounds. The Null-elimination step would merge ⟦c1⟧ with its upper bound, ⟦c2⟧.

In the rare cases where the solving machinery reaches a contradiction in either the single or multiple upper-bounds cases above, backtracking occurs as normal, but the solver actually does assign the type Null to the problematic type unknown. A surprising and serendipitous outcome of the type rules of JSR-14 is that one may use the raw type Cell to stand for the (logical) type Cell⟨Null⟩; while these two types are conceptually quite distinct, the resulting program nonetheless type-checks correctly ('unchecked' warnings aside). This 'pun' provides a straightforward means

of resolving the trickier cases; we have not seen this happen in practise, except in pathological test-cases constructed by hand.

### 5.4.4 Lazy vs. eager assignment

The solving algorithm described above may be characterised as *eager assignment*, because it tries to assign a type to each type unknown $X$ during the round of solving in which $X$ becomes REIFIED; the only exception is made for those type unknowns for which there are no constraints yet, in which case assignment is deferred until the round of solving in which the first constraint appears.

Recall Figure 3-1 on page 45, and constrast the alternative typings #1, #2 and #3; in the latter one, which results from eager assignment, the most specific possible types are chosen for `cb1` and `cb2`, even though this results in a less specific type for `c`. An alternative approach, which we call *lazy* assignment, may defer the assignment of a type unknown even when one or more constraints on it already exist, in the hope of achieving a better overall result. Typings #1 and #2 represent two different outcomes of lazy assignment. (In fact, both the eager and lazy approaches described can be regarded as points on a continuum of constraint solving strategies.)

The disadvantage of a more lazy approach is that it may cause too many types to become coupled. Consider a method `f(Cell c)` and its callers: the invariant subtyping rule ($\mathsf{Cell}\langle x\rangle \leq \mathsf{Cell}\langle y\rangle \Rightarrow x = y$) necessitates that if the inferred type of `c` is parameterised, then *all* callers of `f` must pass similarly-parameterised Cells. A lazy approach would therefore try to infer a type for `c` while simultaneously inferring types for the arguments of all of `f`'s callers. If `f` has many diverse callers, then often the only solution is Cell⟨Object⟩, which can be catastrophically overconstraining.

Informally, this difference can be understood in the following way: in the lazy approach, type constraint information flows *in both directions*, so the type at one caller of `f` can affect not only the type of `f`'s parameter `c`, but also the type inferred at a different call to to `f`. In contrast, in the eager approach, type constraint in-

formation flows only in the same direction as the flow of values, so the process of assignment starts from the 'roots' (allocation sites) and progresses from there.

Interestingly, the implementation difference between lazy and eager assignment is quite straightforward: for lazy assignment, we simply apply the same closure rules described in Section 5.4.1 to the dependency graph described in Section 5.4.2. In our example, this would result in the type of `f`'s parameter `c` and the type of the actual parameter at each call to `f` being all mutually dependent, and hence all these types would be decided together, in the same round of solving.

Several input-specific factors may determine whether a more lazy or more eager result is desirable for a particular declaration, and it is easy to imagine cases in which one is preferable to the other. However, without a notion of which declarations are the most central or significant to a given program, it is hard to generalise reliably. Both of these solving strategies are *greedy*, in the sense that they they look only at a certain set of related type unknowns, and decide upon types for them, before examining the next set. For problems such as this one that do not have optimal substructure[3], greedy algorithms generally find only locally-optimum solutions; an algorithm to achieve or approximate a global solution would be an interesting goal for future investigation.

## 5.5 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.

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

[3]A problem is said to have *optimal substructure* if the optimal solution to the problem contains within it the optimal solutions to the subproblems [CLR90].

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. Parameterised 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:

$$\text{Cell⟨\{Integer, Float\}⟩} \xrightarrow{\text{union elim}} \text{Cell⟨Number⟩}$$
$$\{\, \text{Cell⟨Integer⟩}, \text{Cell⟨Float⟩} \,\} \xrightarrow{\text{union elim}} \text{raw Cell}$$

Knoblock and Rehof [KR01] make a similar proposal: their subtype completion technique introduces new types where necessary in order to permit type elaboration, or inferring types from incompletely typed bytecode.

### 5.5.1 Wildcard types

The Java 1.5 specification has not yet been finalised, but it appears that it will include *wildcard types*, which generalise the use of bounded existentials as type arguments. Every parameterised 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 Cell⟨∃T. T ≤ Number⟩, which is the type of all Cells whose elements are some (unspecified) subtype of Number. It is therefore a supertype of Cell⟨Integer⟩, Cell⟨Float⟩ and Cell⟨Number⟩, 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 Cell⟨∃T. Number ≤ T⟩, 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 Cell⟨∃T. T ≤ Number⟩ has only the `get`-like methods while Cell⟨∃T. Number ≤ T⟩ has only 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.

# Chapter 6

# Implementation

We have implemented the algorithms described in the previous chapters as a fully-automated translation tool called Jiggetai. Jiggetai's output is a type-correct, behaviourally equivalent JSR-14 version of the original Java program. Figure 6-1 shows a schematic diagram of the tool's architecture. Excluding the lossless compiler and Soot components (described below), Jiggetai consists of about 17000 lines (7000 non-comment, non-blank) of JSR-14 code.

In this chapter, we will describe the design of the tool, and discuss some salient features of the implementation. In the first section, we explain the rather unusual decision to use the class-file representation for analysis, despite the fact that the translation occurs at source-level. The following three sections describe implementation details relating to the three main components of the tool in turn: allocation type inference, declaration type inference, and the source-file editor.

## 6.1 Program representation

Since the allocation-type inference is a whole-program analysis, and we cannot demand that source be available for pre-compiled libraries, this 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 implemented both analyses at the bytecode level.

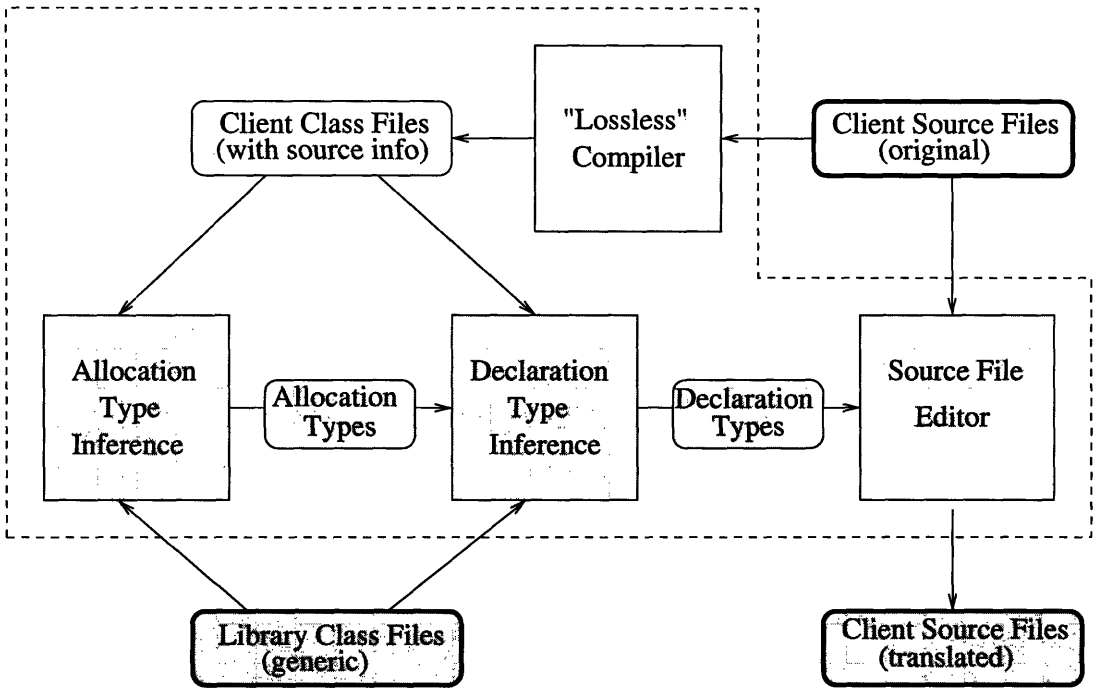



Figure 6-1: Architecture of the Jiggetai tool for automatic translation from Java to JSR-14. Rectangles denote program components; rounded boxes denote data. The dashed border delimits Jiggetai itself; inputs and outputs are shown with heavy outlines.

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 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, we disable the dead-code elimination optimisation in the lossless compiler. Since dead program statements are still subject to type checking, they must be visible to the analysis.

We have extended the Soot [VRHS+99] class-file analysis package to permit 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.

A single local variable declaration statement in the source code may declare more than one variable, and each variable may have more than one disjoint live range or *web*. This suggests that, to obtain the most precise results, the analysis should operate at the JVM register level of abstraction; this would allow it to infer distinct types for such co-declared variables, or for different webs of the same variable. For example, in this code fragment:

```
Iterator i, j;
for(i = a.iterator(); i.hasNext(); )   f(i.next());
for(i = b.iterator(); i.hasNext(); )   g(i.next());
for(j = c.iterator(); j.hasNext(); )   h(j.next());
```

variables `i` and `j` are co-declared (they share the same declarator), and variable `i` has two webs (it is used for iteration over `a`, then later re-used for `b`). A JVM register-level analysis could transform the code above into this:

```
Iterator<A> i_a;  Iterator<B> i_b;  Iterator<C> j;
...
```

in which the variable `i` has been split into two, and in which the declaration of `i` and `j` has also been split, allowing all three local variables to be typed independently.

We anticipated that both co-declaration and variable re-use would be a significant source of *over*-constraint in the system, so we implemented the constraint generation to allow a choice between the source-local and JVM-register levels of abstraction. When we conducted our experiments, we were surprised to discover only one instance each of co-declaration and of variable re-use that caused a degradation of the results. The source-editing problem becomes significantly more complex when declaration splitting and variable splitting must be performed[1], and we did

[1]Consider the problem of splitting this declaration:
`for(Iterator a = f(), b = g(); a.hasNext(); )`

not believe the extra precision justified this implementation effort; see Section 7.1 for further details. Of course, if the extra precision is critical, the user may elect to enable JVM-register level analysis and make any necessary fix-ups manually without great effort.

### 6.1.1 `Signature` class-file attributes

Since compilation of JSR-14 uses homogeneous translation by type erasure, generic type information has no effect on the semantics of the program, and is not required by the JVM during execution. However, the Java compilation model allows precompiled classfiles to be used as class declarations during subsequent compilation, so to enable proper JSR-14 type checking, generic type information must be preserved in the classfile in some form.

`Signature` attributes are the means by which this generic type information is saved; these are comments in the classfile that associate with each externally-visible declaration, such as a field, method parameter, or extends-clause, an encoding of its parameterised type. Also, they tabulate the names and bounds of type variables declared by generic classes and generic methods. No signature attributes are created for local variables or other declarations within the body of a method, since these are not visible to subsequent compilation. `Signature` attributes are ignored by the JVM.

### 6.1.2 Raw `extends`-clauses

When the analysis encounters a class declared in client code that `extends`[2] a generic library class, the question arises: how should the type in the `extends`-clause be instantiated? For example, a client class declared `class PersonList extends List` in Java would probably have been declared in JSR-14 as `extends List<Person>`. Unfortunately, both the allocation type inference and declaration type inference al-

[2]Throughout this thesis, we do not distinguish between `extends` and `implements`.

gorithms, as presented, are unable to infer type instantiations in `extends`-clauses, as this information is required *a priori* for the WIDENING helper function.

Therefore, we conservatively assume that all client classes that extend a generic class at a raw type are to have their superclass immediately instantiated at the least specific type, for example `extends List<Object>`. While this is indeed a source of imprecision in the results (see Section 7.1), client classes that extend generic collection classes, but are not themselves generic, are relatively uncommon.

## 6.2 Allocation Type Inference

In the implementation of the allocation type inference, two limitations in the JSR-14-1.3 prototype compiler resulted in partial information during allocation type inference, and required special effort to work around. The first was the inability to determine, from the class-files, which method a local class was enclosed by (or indeed that it was a method-local class). The second problem was related to *retrofitting*. Both of these are explained below, and are followed by an explanation of how the analysis can be made more modular.

### 6.2.1 Missing 'enclosing method' information

Despite the presence of `Signature` attributes in the class-file, the output of the JSR-14-1.3 prototype compiler nonetheless lacks certain information required for correct reconstruction of the types of the original program. Local classes declared within a generic method may refer to the method's type variables, but the classfile does not record the local class's enclosing method; thus, when parsing the local class, the tool is unable to establish the proper type environment in which to resolve references to free type variables it encounters.

We worked around this problem by requiring the user to provide an extrinsic specification of the enclosing generic method of each local class that refers to method type variables. In practise, such annotations are rarely required (the JDK

required none), and they could even be discovered automatically by a preprocessing step (by searching for the method in which each local class is allocated) although the the implementation effort did not seem justified.

We note that the latest version of the JSR-14 specification and compiler (e.g. Java 1.5 beta 2) addresses this problem directly by requiring that a newly-specified `EnclosingMethod` class-file attribute to be emitted in such cases, providing exactly the missing information.

### 6.2.2 Imprecision due to superficial retrofitting

A significant feature of the JSR-14 toolchain is that it allows the user to *retrofit* generic types into pre-existing non-generic library classfiles, allowing smoother integration of separate libraries into an application. This is especially useful for third-party libraries, for which the source code may be unavailable, and for legacy libraries for which the maintainer (if any) has no intention of translating them to exploit generic types. (Of course, it is the responsibility of the client to retrofit the *correct* types onto the library.)

Without the ability to retrofit generic types, the application programmer would be confronted by numerous warning messages whenever their code using parameterised types interacted with non-generic declarations in the library. Notably, though, retrofitting is usually *superficial*: it is limited to the externally-visible declarations. Generic type information is typically not retrofitted to private library members, nor to unnameable ones such as local and anonymous classes.

Although more recent versions of the compiler come with libraries that have been re-written using generic types throughout, the JDK libraries that shipped with the JSR-14-1.3 compiler were superficially retrofitted versions of the non-generic libraries.[3]

[3]In actuality, the JSR-14-1.3 libraries were not even retrofitted: they consisted of the normal Java 1.4 libraries, with a small library of vestigial implementations—lacking method bodies—of the `java.util` classes, whose sole purpose was to provide superficial generic type information for the package during compilation. We were able to 'fuse' these to produce the retrofitted library ourselves.

As explained in Section 4.4, the policy of our pointer analysis is to exploit the presence of generic type information (from `Signature` attributes) to guide its use of call- and data-context-sensitivity. The superficial retrofitting of the JDK library thus presents a problem, because the lack of generic type information within the library results in a loss of context-sensitivity.

Of particular concern was the non-public inner-class `HashMap<K,V>.Entry`: since all instances of this class were modelled as a single value, we noticed a serious loss of precision with inference of `HashMap` allocations.

We 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.'

2. Create a type-correct stub version of the library, and use it in place of the real library when compiling. (This approach is conceptually similar to that taken by Tip et al. [TFDK04].) This approach is labour-intensive, unsound and error-prone, because care is required to ensure that the stub method bodies induce the same generic type constraints as the original library would have done. We implemented it to compare its performance and results with the retrofitting approach.

   (Note that type-correct stubs are not the same as the vestigial libraries mentioned above: vestigial libraries are merely placeholders for `Signature` attributes; in contrast, type-correct stubs are method implementations that in-

duce *semantically-equivalent type constraints* in the abstract semantics of the analysis.)

Aggregated over all the benchmarks in Figure 7-2, 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 run-time was small, typically less than 10%.

### 6.2.3 Modularising allocation type inference

As with all whole-program static analyses, our pointer analysis requires hand-written annotations to summarise the effects of calling native methods and reflective code; without them, soundness cannot be ensured. With the exception of `Object.clone` and `System.arraycopy`, we currently use very naïve and conservative annotations for such methods; none of our benchmarks makes significant use of them.

Whole-program analysis may be undesirable for very large and complex systems, and for this reason, it may be preferable to provide a summary of the effects of whole packages in a similar form to the summaries used for reflection and native methods. At first, providing abstract-semantics-equivalent summaries of large quantities of code might seem to be a daunting task. However, the inherent modularity of separate packages means that this is actually unnecessary in many cases.

The modularity of the `java.util` package is typical in this regard: code in this package defines several generic collection classes, but rarely calls on code in other packages of the JDK. Likewise, code in other packages of the JDK typically uses classes from `java.util` only as an ordinary client. Furthermore, with the exception of package `java.awt`, the other JDK code rarely accepts or returns instances of these collections in its public interface.

This means that apart from these two packages, the rest of the JDK can be safely excluded from the pointer analysis. We can assume with some confidence that unanalysed code does not interfere with — in other words, has no side-effects relevant

to—the implementation of `java.util`'s generic classes; and the analysis makes pessimistic assumptions about the elements of instances of `java.util` collections returned from unanalysed code, but there are few of them. Of course, making the judgement that this reduction in scope is safe requires a degree of understanding of the code, and is in general unsound, but in our experience, almost all classes written in a generic style implement quite modular abstractions.

## 6.3 Declaration type inference

As stated in Section 3.7, our algorithm requires that the client code contains no generic class or method declarations. However, it works equally well on client code containing *uses* of parameterised types; currently, it simply ignores any existing type arguments and infers new ones. It would be a relatively straightforward extension to the constraint generation phase to treat pre-existing parameterised types in the client code as 'fixed' points (constants) in the constraint system, and doing so would have a twofold benefit.

Firstly, it would allow the tool to operate on code that has already been partially translated (e.g. by hand) to make use of parameterised types. Secondly, it would enable the tool to be used for successive refinement of the solution based upon user interaction: after the first iteration, the user fixes the declarations where the analysis inferred an undesirable result, and then the analysis is re-executed, until the ideal result is converged upon.

## 6.4 Source file editor

The source-file editor component (Figure 6-1) takes as input the solution to the type constraint system, which was the result of the declaration type inference and which provides the new types for each declaration in the client code. The editor uses the lexical-extent information added by the lossless compiler (and carried along by the declaration type inference) to construct a set of edits to be applied to the original

Java source-file, replacing non-generic uses such as Cell with generic ones such as Cell⟨String⟩.

In contrast to many tools that generate source code as an intermediate format, the source file editor modifies hand-written Java code, so its output is permanently visible to programmers. Therefore, its design strives to be minimally intrusive, and to preserve all formatting, comments, and other non-semantic details of the source-file that might be lost by a tool that merely pretty-printed a transformed syntax tree. In particular, the source editor inserts types using the same notation that a programmer would, based upon the set of names `imported` into a given source file, e.g. type `java.lang.String` would be written `String`. The editor uses the shortest name possible without causing ambiguity.

# Chapter 7

# Experiments

In order to evaluate our analyses and tools, we ran our implementation over the programs listed in Figure 7-1. The programs are as follows: *antlr* is a scanner/parser generator toolkit[1]; *htmlparser* is a library of parsing routines for HTML[2]. *JavaCUP* is an LALR parser generator[3]; *JLex* is a lexical analyser generator[4]; *junit* is a unit-testing framework[5]; *TelnetD* is a Telnet daemon[6]; *v_poker* is a video poker game[7]. Figure 7-1 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 7-2 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

[1] `http://www.antlr.org/`
[2] `http://htmlparser.sourceforge.net/`
[3] `http://www.cs.princeton.edu/~appel/modern/java/CUP/`
[4] `http://www.cs.princeton.edu/~appel/modern/java/JLex/`
[5] `http://www.junit.org/`
[6] `http://telnetd.sourceforge.net/`
[7] `http://vpoker.sourceforge.net/`

| 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 7-1: Benchmark 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 7-2: 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.

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. (To confirm these counts, we performed a complete manual generic translation of four of the benchmarks, and counted the number of casts eliminated; they agreed.)

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 unoptimised implementation took no more than 8 minutes to translate any program.

## 7.1 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` parameterisation

As described in Section 6.1.2, whenever the analysis encounters a client class that extends a generic library class, the `extends`-clause is parameterised 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 specialise `clone`, since it does not require that its instances be cloneable.[8] 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.

[8] Or, for compatibility, `clone` may not have been covariantly specialised, as is the case for `HashSet`.

### 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 the two webs of `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%).

### 'Filter' idiom

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

```
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[9].

The filter idiom is heavily used by the *htmlparser* benchmark. This problem could perhaps be addressed by dataflow analysis, by applying a limited form of declaration splitting to blocks dominated by an `instanceof` test. At the source level, this would require the application of a `(String)` cast to the variable `o` each time it is

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

referenced within the `if`-statement. (While such casts would change the program's erasure, the changes would be safe as the casts are guaranteed to succeed.)

### 7.1.1 Time/space performance

The execution time of the larger benchmarks was dominated by the naïve implementation of the resolution algorithm of Section 5.4. 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.

For all programs, the other phases (class-file parsing, pointer analysis, s-unification, constraint generation) completed in under 40 seconds.

The degree of backtracking actually used over the benchmarks was very small, and consequently, the proportion of raw types in the solution is low, suggesting that they are rarely required in practise. (This is consistent with our own experience of programming in JSR-14.)

# Chapter 8

# Related Work

Our primary contribution 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 pre-existing components. Then, we briefly discuss work related to our techniques: type constraint systems and type inference.

See Section 2.4 for a brief survey of work related to the evolution of generic types in Java.

## 8.1 Generalisation for re-use

Two notable previous papers [SR96, Dug99] use automated inference of polymorphism with the goal of source-code generalisation 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 optimisation, 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 [SR96] aim to translate C to C++. They use type inference to detect latent polymorphism in C functions designed for use with parameters of primi-

tive type, and the result of generalisation 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 generalised function on its argument. (They give as an example the $x^y$ function `pow()`, which is defined only for numbers but could be applied to any type for which multiplication is defined, such as `Matrix` or `Complex`.) Their work focuses exclusively on generic functions, not classes, and tries to detect latent reusability; in contrast, our work seeks to enforce stronger typing where reusability was intended by the programmer. The problem domain is quite different to 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 [Dug99] presents a type analysis for inferring genericity in a Java-like language. Duggan gives a modular (intra-class) constraint-based parameterisation analysis that translates a monomorphic object-oriented kernel language called Mini-Java into a polymorphic variant, PolyJava, that permits abstracting classes over type variables. The translation creates generic classes and parameterised 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 [vDD04] address both the parameterisation 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 generalisation 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 analysed 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 parameterised 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 parameterisation and instantiation problems [DE03]. We restricted ourselves to a sound approach, and abandoned the combined approach after discovering that heuristics useful in specific circumstances caused unacceptable loss of generality in others.)

Tip et al. [TFDK04] 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 [Ecl] 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 (pos-

sibly dangerous) unchecked warnings and raw references without sacrificing the number of eliminated casts. For example, the method `displayValue` in Figure 2-3 could be changed into a generic method, rather than leaving the reference raw. The authors do not discuss soundness or behaviour preservation.

Tip, Kieżun, and Baümer [TKB03] present the use of type constraints for refactoring (i.e., modifying the program's source code without affecting its behaviour). While their work focused on refactoring for generalisation, ours can be seen as refactoring for *specialisation*, changing types from raw to non-raw.

The CodeGuide [Cod] 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 [IDE] IDE also supports a Generify refactoring; again, no details about the analysis techniques are available, and we have not experimented with this tool.

## 8.2 Type constraint systems

Both our allocation type inference and declaration type inference are type-constraint-based algorithms in the style of Aiken and Wimmers [AW93], 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 [PS94].

Our pointer analysis makes use of the conceptual framework of Wang and Smith [WS01]; we instantiate it with a particular set of choices for polymorphism that fit well with our problem. Plevyak and Chien [PC94] provide an iterative class analysis that derives control and data flow information simultaneously, with the goal of optimisations such as static binding, inlining and unboxing. Some representative

applications are statically discharging run-time casts [CF91, WS01], eliminating virtual dispatch [BS96] and alias analysis [OJ97, O'C01].

## 8.3 Polymorphic type inference

There is a vast literature on polymorphic type inference dating from Milner [Mil78], who introduced the notion in the context of the ML programming language. Our goal is quite different to 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 [OB89, PS91] 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 [EST95, Dug99], just as our S-UNIFY algorithm does, though the technical machinery is different. McAdam et al. [MKB01] 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. [GHM00] 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.

Henglein and Rehof [HR95] present an efficient and modular algorithm for inferring ML-like principal types in Scheme code. Their algorithm accepts all legal Scheme programs, even those which cannot be directly typed in ML: *coercion* operations (casts) are inserted as required in such cases. One application of their algorithm is the debugging of Scheme programs: type inference can identify operations that are destined to fail according to the ML type abstraction; in this regard,

it is similar to *soft typing* [CF91], but unlike the latter, it does not require whole-program analysis.

# Chapter 9

# Future work

There are several interesting directions for future work in this area. We plan to make the implementation of our tool available, once licensing issues are resolved (the implementation currently depends upon Sun's prototype compiler), and to make a number of improvements to it.

The Java 1.5 specification has not yet been finalised, but we are currently working to update the tool to include support for the latest published revision. 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 5.5.1. Because wildcard types do not support all of the operations of the corresponding raw type — for example, $\mathsf{Cell}\langle\exists T.\, T \leq \mathsf{Number}\rangle$ has no `set` method — additional analysis is required to determine whether a wildcard type is a suitable alternative to a raw type in a particular declaration. The analysis would require an examination of which methods are called on each variable, transitively; if a `set`-like method is called on a given variable `v`, or a variable it flows to, then `v` cannot be ascribed a wildcard-`extends` type. (Similarly, the analysis must take account of which instance fields are read or written on `v`.)

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. The changes to the constraint

generation machinery would be quite straightforward: whenever a parameterised type is encountered within client code, it should be regarded as 'fixed', i.e., a type constant. The constraint solving algorithm would not need to change at all; however, if the solver could be optimised so as to complete within a matter of seconds, this style of interaction with the tool would be made much more useful. We believe this goal is quite achievable.

The solving algorithm described in Section 5.4 is 'greedy': it decomposes the problem into many subproblems, each consisting of a set of mutually-dependent declarations. The solver finds optimal or near-optimal solutions to these subproblems containing no more raw types than necessary. However, when the solutions to the subproblems are composed, it does not necessarily yield an optimal solution to the whole constraint system. While this did not to be appear to be a problem in any of the medium-sized programs we have encountered, it is easy to construct pathological cases in which the overall solution is significantly sub-optimal. Therefore, an interesting problem would be to introduce a degree of global optimisation into the solving algorithm to further increase the number of casts eliminated from within client code.

The $C^\sharp$ language is a experiencing a parallel evolution towards parametric polymorphism [KS01, YKS04], although its generics are based upon heterogeneous translation. Nonetheless, the language bears many similarities to Java, and the ideas in our approach are readily applicable to $C^\sharp$. Indeed, in the absence of raw types and the loophole they cause, the problem is dramatically simpler: allocation type inference can be dispensed with altogether.

Perhaps the most general problem in this area is that of accurate inference of both generic classes *and* instantiations simultaneously. In our previous exploration of this combined problem [DE03], we were unable to find an algorithm that was both general and precise: without heuristics, inferred generics were *over*-generalised, having an overwhelming number of type parameters. On the other hand, it is hard to find a set of heuristics to reduce the unwanted generality without also precluding generalisation in many desirable ways; a similar observation is

made in [vDD04]. While challenging, the parameterisation problem is, in our experience, not the most difficult or time-consuming part of the problem of migrating towards generics.

# Chapter 10

# 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 Chapter 3. It is sound: it never infers an unsafe type for a declaration. It is behaviour-preserving: it does not change method signatures, the erasure of parameterised 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 to the result of manual annotation.

By meeting these goals, our work is more general and, we hope, more useful than other approaches to this problem. In particular, [TFDK04] targets only the standard `java.util` libraries and cannot infer instantiations for generics defined in other libraries or within applications and [vDD04] makes a number of unsound

assumptions. Our benchmarks demonstrate the applicability of this approach for several real-world programs.

Context-sensitive pointer analysis is a widely-used technique in program analysis, but naïve implementations have long suffered from poor scalability; much work has been done to make 'adaptive' analyses that carefully target the use of context-sensitivity to overcome this problem. Our approach is, to our knowledge, the first analysis for Java that uses generic type annotations for targeting the use of context-sensitivity. Our application is type analysis, but this technique could equally well be used for many other abstractions, such as inter-procedural dataflow problems.

Raw types require the use of conditional constraints, since the type rules for accessing members through raw types and through parameterised types are quite different. The presence of raw types in the type system is a loophole allowing potentially unsafe operations; analysing 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.

# Bibliography


[ABC03] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 96–114. ACM Press, 2003.

[AC02] Eric Allen and Robert Cartwright. The case for run-time types in generic Java. In *Proceedings of the inaugural conference on the Principles and Practice of Programming, 2002 and Proceedings of the second workshop on Intermediate Representation Engineering for Virtual Machines, 2002*, pages 19–24. National University of Ireland, 2002.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 49–65, Atlanta, GA, USA, October 5–9, 1997. ACM Press.

[Age95] Ole Agesen. The Cartesian Product Algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95, the 9th European Conference on Object-Oriented Programming*, pages 2–26, Aarhus, Denmark, August 5–8, 1995. Springer-Verlag.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 9–11, 1993.

[BCK+01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Mark, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kreste Thorup, and Philip Wadler. Adding generics to the Java programming language: Participant draft specification. Technical report, Sun Microsystems, April 27, 2001.

[BD98] Boris Bokowski and Markus Dahm. Poor man's genericity for Java. In *Proceedings of JIT'98*, Frankfurt, Germany, November 12–13, 1998. Springer-Verlag.

[BOSW98a] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ specification. `http://www.cis.unisa.edu.au/~pizza/gj/Documents/#gj-specification`, May 1998.

[BOSW98b] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 183–200. ACM Press, 1998.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, San Jose, CA, USA, October 6–10, 1996.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional Programming Languages and Computer Architecture*, pages 273–280. ACM Press, 1989.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, Cambridge, Massachusetts and New York, New York, 1990.

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

[CS98] Robert Cartwright and Guy L. Steele Jr. Compatable genericity with run-time types for the Java programming language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 201–215, Vancouver, BC, Canada, October 20–22, 1998.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, USA, June 19–23, 1989.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 49–70, New Orleans, LA, USA, October 1–6, 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[DE03] Alan Donovan and Michael D. Ernst. Inference of generic types in Java. Technical Report MIT/LCS/TR-889, MIT Laboratory for Computer Science, Cambridge, MA, March 22, 2003.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95, the 9th European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, August 5–8, 1995.

[DKTE00] Alan Donovan, Adam Kieżun, Matthew Tschantz, and Michael Ernst. Converting Java programs to use generic libraries. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*. ACM Press, 200.

[Dug99] Dominic Duggan. Modular type-based reverse engineering of parameterized types in Java code. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, pages 97–113, Denver, Colorado, November 3–5, 1999.

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

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–184, Austin, TX, USA, October 1995.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, November 2001.

[GHM00] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium*, pages 199–219, Santa Barbara, CA, USA, June 2000.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, second edition, 2000.

[HP01] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, January 2001.

[HR95] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *Functional Programming Languages and Computer Architecture*, pages 192–203, La Jolla, California, United States, 1995. ACM Press.

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

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Informal Proceeings of the Eighth International Workshop on Foundations of Object-Oriented Languages (FOOL 8)*, London, January 2001.

[IV02] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *ECOOP 2002 — Object-Oriented Programming, 16th European Conference*, pages 441–469, Málaga, Spain, June 12–14, 2002.

[Jav01] 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.

[Jav04] JavaSoft, Sun Microsystems. RFE 4064105: Compile-time type safety with Generics, 1997–2004. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4064105`.

[KR01] Todd B. Knoblock and Jakob Rehof. Type elaboration and subtype completion for Java bytecode. *ACM Transactions on Programming Languages and Systems*, 23(2):243–272, March 2001.

[KS01] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2001.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, January 15–17, 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[MKB01] Bruce McAdam, Andrew Kennedy, and Nick Benton. Type inference for MLj. In *Scottish Functional Programming Workshop*, pages 159–172, 2001. Trends in Functional Programming, volume 2, Chapter 13.

[OB89] Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 445–456, New Orleans, LA, USA, October 1–6, 1989.

[O'C01] Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 2001.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, MA, May 1997.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159. ACM Press, 1997.

[PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–340, Portland, OR, USA, October 1994.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 146–161, Phoeniz, AZ, USA, October 1991.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley and Sons, 1994.

[PSLM00] P.J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library*. Prentice Hall PTR, 2000.

[SA98] Jose H. Solorzano and Suad Alagic. Parametric polymorphism for Java: a reflective solution. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 216–225. ACM Press, 1998.

[SE90] Bjarne Stroustrup and Margaret A. Ellis. *C++: The Annotated Reference Manual.* Addison-Wesley, Boston, MA, 1990.

[SR96] Michael Siff and Thomas Reps. Program generalization for software reuse: From C to C++. In *Proceedings of SIGSOFT '96 Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 135–146, San Francisco, CA, USA, October 16–18, 1996.

[TFDK04] Frank Tip, Robert Fuhrer, Julian Dolby, and Adam 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.

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1289–1296, Nicosia, Cyprus, March 14–17, 2004.

[TKB03] Frank Tip, Adam Kieżun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 13–26, Anaheim, CA, USA, November 6–8, 2003.

[TKH99] Krishnaprasad Thirunarayan, Günter Kniesel, and Haripriyan Hampapuram. Simulating multiple inheritance and generics in Java. *Computer Languages*, 25(4):189–210, 1999.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, pages 281–293, Minneapolis, MN, USA, October 15–19, 2000.

[vDD04] Daniel von Dincklage and Amer Diwan. Converting Java classes to use generics. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, Vancouver, BC, Canada, October 26–28, 2004.

[VN00] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, pages 146–165, Minneapolis, MN, USA, 2000. ACM Press.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java bytecode optimization framework. In *CASCON*, pages 125–135, Mississauga, Ontario, Canada, November 8–11, 1999.

[WS01] Tiejun Wang and Scott Smith. Precise constraint-based type inference for Java. In *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, pages 99–117, Budapest, Hungary, June 18–22, 2001.

[YKS04] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .NET common language runtime. In *Proceedings of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–51, Venice, Italy, January 14–16, 2004.