RC24804 (W0906-043) June 9, 2009
Computer Science

IBM Research Report

Refactoring using Type Constraints

Frank Tip, Robert M. Fuhrer
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA

Adam Kiezun
Massachusetts Institute of Technology
32 Vassar Street
Cambridge, MA 02139
USA

Michael D. Ernst
University of Washington
Department of Computer Science and Engineering
Box 352350
Seattle, WA 98195-2350
USA

Ittai Balaban
World Evolved Services
130 Madison Avenue
New York, NY 10016
USA

Bjorn De Sutter
Ghent University and Vrije Universiteit Brussel
Electronics and Information Systems Department
Sint-Pietersnieuwstraat 41
9000 Gent
Belgium

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Refactoring using Type Constraints

FRANK TIP and ROBERT M. FUHRER
IBM T.J. Watson Research Center
ADAM KIEŻUN
Massachusetts Institute of Technology
MICHAEL D. ERNST
University of Washington
ITTAI BALABAN
World Evolved Services
and
BJORN DE SUTTER
Ghent University and Vrije Universiteit Brussel

Type constraints express subtype relationships between the types of program expressions, for example those relationships that are required for type correctness. Type constraints were originally proposed as a convenient framework for solving type checking and type inference problems. This paper shows how type constraints can be used as the basis for practical refactoring tools. In our approach, a set of type constraints is derived from a type-correct program P. The main insight behind our work is the fact that P constitutes just one solution to this constraint system, and that alternative solutions may exist that correspond to refactored versions of P. We show how a number of refactorings for manipulating types and class hierarchies can be expressed naturally using type constraints. Several refactorings in the standard distribution of Eclipse are based on our work.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming, program editors*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Refactoring, type constraints, program transformation

Authors addresses: F. Tip and R. M. Fuhrer, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights,NY 10598; email: ftip@us.ibm.com; rfuhrer@us.ibm.com. A. Kieżun, MIT CSAIL, 32 Vassar St., Cambridge, MA 02139, USA; email: akiezun@mit.edu. M. D. Ernst, University of Washington, Dept. of Computer Science and Engineering, Box 352350, Seattle WA 98195-2350, USA; email: mernst@cs.washington.edu. I. Balaban, World Evolved Services, 130 Madison Avenue, New York, NY 10016; email: ibalaban@reloadnyc.com. B. De Sutter, Electronics and Information Systems Department, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium; email: brdsutte@elis.ugent.be.

1. INTRODUCTION

Refactoring is the process of applying behavior-preserving transformations (called "refactorings") to a program's source code with the objective of improving the program's design. Common reasons for refactoring include eliminating duplicated code, making existing program components reusable in new contexts, and breaking up monolithic systems into components. Refactoring was pioneered in the early 1990s by Opdyke et al. [Opdyke 1992; Opdyke and Johnson 1993] and by Griswold et al. [Griswold 1991; Griswold and Notkin 1993]. The profile of refactoring received a major boost with the emergence of code-centric design methodologies such as extreme programming [Beck 2000] that advocate continuous improvement of code quality. Fowler [1999] and Kerievsky [2004] authored popular books that classify many widely used refactorings, and Mens and Tourwé [2004] surveyed the field.

Refactoring is usually presented as an interactive process where the programmer first chooses a point in the program where a specific transformation should be applied. Then, the programmer must verify whether a number of specified preconditions hold, and, if so, apply a number of prescribed editing steps. Checking the preconditions may involve nontrivial analysis, and the number of editing steps may be significant. Therefore, automated tool support for refactoring is highly desirable, and has become a standard feature of modern development environments such as Eclipse (`www.eclipse.org`) and IntelliJ IDEA (`www.jetbrains.com/idea`).

The main observation of this paper is that, for an important category of refactorings related to the manipulation of class hierarchies and types, the checking of preconditions and computation of required source code modifications can be expressed by a system of type constraints. Type constraints [Palsberg and Schwartzbach 1993] are a formalism for expressing subtype relationships between the types of program elements that must be satisfied in order for a program construct to be type-correct. They were originally proposed as a means for expressing type checking and type inference problems. In our work, we derive a set of type constraints from a program P and observe that, while the types and class hierarchy of P constitute one solution to the constraint system, alternative solutions may exist that correspond to refactored versions of P. This paper shows how several refactorings for manipulating class hierarchies and types can be expressed in terms of type constraints. This includes refactorings that: (i) introduce interfaces and supertypes, move members up and down in the class hierarchy, and change the declared type of variables, (ii) introduce generics, and (iii) replace deprecated classes with ones that are functionally equivalent.

Scope and Assumptions

The project reported on in this paper was initiated in 2002, when one of the authors (A. Kieżun) was a member of the Eclipse development team. At this time, there was a desire to implement the EXTRACT INTERFACE refactoring in Eclipse. After devising a practical solution based on type constraints [Tip et al. 2003], we quickly realized that this type constraint model could be extended to serve as the basis for several other refactorings. From the outset, our goal has been to handle the entire Java programming language and to create realistic implementations that could be contributed to the Eclipse platform. Currently, several refactorings[1] in the Eclipse distribution are based on the research presented in this paper.

Our work makes a number of assumptions that are customary for refactoring tools. We assume that all the source code that needs to be refactored is available for analysis. The formalization in this paper omits several Java constructs including exceptions, auto-boxing, overloading, and annotations. However, in principle, our techniques are capable of handling the full Java programming language, and we expect that they could be adapted to handle other statically typed object-oriented languages such as C++, Eiffel, C#, and Scala, with varying degrees of effort. Our implementations were successfully applied to large Java applications (see Section 6), and in our experiments, we determined that program behavior was preserved by compiling and running the refactored programs where possible to ensure that program behavior was preserved. That said, our implementations have some known shortcomings and limitations. As with all refactoring tools, we cannot guarantee that behavior is preserved if applications use reflection or dynamic loading. Moreover, we assume that implementations of `clone()` are well-behaved in the sense that the returned object preserves any type arguments of the receiver expression. Finally, the Replace Class refactoring presented in Section 5 relies on a static points-to and escape analysis. While any static points-to or escape analysis can be used, the precision of the particular analysis affects the effectiveness of the refactoring.

Our previous papers [Tip et al. 2003; De Sutter et al. 2004; Fuhrer et al. 2005; Balaban et al. 2005; Kieżun et al. 2007; Tip 2007] presented these refactorings in detail, along with detailed experimental evaluations. This paper presents an overview of the work and a summary of the experimental results, using variations on a single running example to show how different refactorings require variations on a common type constraints model.

Organization of this Paper

The remainder of this paper is organized as follows. Section 2 presents the basic type constraint formalism that will be used as the basis for the refactorings presented in this paper. Section 3 shows how several refactorings related to generalization can be expressed using the basic type constraints model. Section 4 extends the basic type constraints model to model the inference of generics, and presents two refactorings for introducing generics into Java programs. In Section 5, we present a refactoring for replacing deprecated classes with functionally equivalent ones that replace them, based on some further extensions to the type constraints model. Section 6 summarizes experiments that measure the effectiveness of the refactorings presented in Sections 4 and 5. Finally, Section 7 discusses related work, and Section 8 presents conclusions and directions for future work.

2. TYPE CONSTRAINTS

Type constraints [Palsberg and Schwartzbach 1993] are a formalism for expressing subtype relationships between the types of declarations and expressions. Figure 1

[1]This includes the Extract Interface, Generalize Declared Type, Pull Up Members, and Infer Generic Type Arguments refactorings presented in this paper, among others.

Notation:

M, M', $\cdots$	methods	I, I', $\cdots$	interfaces
F, F', $\cdots$	fields	T, T', $\cdots$	types (classes or interfaces)
C, C', $\cdots$	classes	E, E', $\cdots$	expressions
$Param(M, i)$	the i^{th} formal parameter of method M		

Constraint variables:

α	::=	T	a type constant
		$[E]$	the type of E
		$[M]$	the declared return type of M
		$[F]$	the declared type of F
		$Dcl(M)$	the type in which M is declared
		$Dcl(F)$	the type in which F is declared

Type constraints (α, α' denote constraint variables):

$\alpha = \alpha'$	type α is the same as type α'
$\alpha < \alpha'$	type α is a proper subtype of type α'
$\alpha \leq \alpha'$	type α is the same as, or a subtype of, type α'
$\alpha \leq \alpha_1$ **or** $\cdots$ **or** $\alpha \leq \alpha_k$	$\alpha \leq \alpha_i$ holds for at least one i, $(1 \leq i \leq k)$

Fig. 1. Syntax of type constraints.

$$\frac{\text{assignment} \quad E_1 = E_2}{[E_2] \leq [E_1]} \quad (1) \qquad \frac{\text{access } E \equiv E_0.f \text{ to field } F}{\begin{array}{c}[E] = [F] \quad (2)\\ [E_0] \leq Dcl(F) \quad (3)\end{array}}$$

$$\frac{\begin{array}{c}\text{call } E \equiv E_0.m(E_1, \cdots, E_n) \text{ to method } M\\ RootDefs(M) = \{M_1, \cdots, M_k\},\ E'_i \equiv Param(M, i),\ 1 \leq i \leq n\end{array}}{\begin{array}{cr}[E] = [M] & (4)\\ [E_i] \leq [E'_i] & (5)\\ [E_0] \leq Dcl(M_1) \textbf{ or } \cdots \textbf{ or } [E_0] \leq Dcl(M_k) & (6)\end{array}}$$

$$\frac{\text{constructor call } E \equiv \texttt{new}\ C(E_1, \cdots, E_n) \text{ to constructor } M,\ E'_i \equiv Param(M, i),\ 1 \leq i \leq n}{\begin{array}{cr}[E] = C & (7)\\ [E_i] \leq [E'_i] & (8)\end{array}}$$

$$\frac{\begin{array}{c}M' \text{ overrides } M,\ M' \neq M,\ 1 \leq i \leq NumParams(M')\\ E_i \equiv Param(M, i),\ E'_i \equiv Param(M', i)\end{array}}{\begin{array}{cr}[E'_i] = [E_i] & (9)\\ [M'] \leq [M] & (10)\\ Dcl(M') < Dcl(M) & (11)\end{array}}$$

$$\frac{F' \text{ hides } F}{Dcl(F') < Dcl(F)} \quad (12) \qquad \frac{\texttt{return } E \text{ in method } M}{[E] \leq [M]} \quad (13)$$

$$\frac{\text{cast } (T)E}{[(T)E] = T} \quad (14) \qquad \frac{\text{downcast } (T)E}{[(T)E] \leq [E]} \quad (15) \qquad \frac{\text{upcast } (T)E}{[E] \leq [(T)E]} \quad (16)$$

$$\frac{\text{type } T}{\begin{array}{cr}T \leq \texttt{java.lang.Object} & (17)\\ [\texttt{null}] \leq T & (18)\end{array}} \qquad \frac{\text{implicit declaration of } \texttt{this} \text{ in method } M}{[\texttt{this}] = Dcl(M)} \quad (19)$$

Fig. 2. Type constraints for a set of core Java language features.

shows the syntax for type constraints used in this paper, which relies on the following two concepts:

—A *constraint variable* represents the type of a program construct. For example, a constraint variable $[E]$ represents "the type of expression E".

—A *type constraint* constrains the relationship between two or more constraint variables. For example, a type constraint $\alpha \leq \alpha'$ states that the type represented by constraint variable α must be the same as, or a subtype of, the type represented by constraint variable α'.

Type constraints are generated from a program's abstract syntax tree in a syntax-directed manner, and encode relationships between the types of declarations and expressions that must be satisfied in order to preserve type correctness or program behavior. Figure 2 shows rules that generate constraints from a representative set of program constructs.

In order to simplify the presentation of the refactorings in this paper, we will assume that (i) programs do not use overloading, meaning that there are no methods with the same name but different formal parameter types, and (ii) that if two classes C_1 and C_2 declare a method with the same signature, then C_1 is a subtype of C_2 or C_2 is a subtype of C_1. Note that any program that does not meet these requirements can be transformed into an equivalent program that does via one or more renamings. These assumptions help us avoid a number of problems, such as the accidental creation or deletion of overriding relationships.

We will now discuss the rules of Figure 2. Rule (1) states that, for an assignment $E_1 = E_2$, a constraint $[E_2] \leq [E_1]$ is generated. Intuitively, this captures the requirement that the type of the right-hand side E_2 be a subtype of the type of the left-hand side E_1 because the assignment would not be type-correct otherwise.

Rules (2) and (3) state the type constraints induced by field access operations. For a field access $E_0.f$ that refers to a field F, Rule (2) states that the type of the entire expression $E_0.f$ is defined to be the same as the declared type of field F. Rule (3) states that the type of expression E_0 must be a subtype of the type in which field F is declared.

We say that a call $E_0.m(E_1, \cdots, E_k)$ is to a method M if, at run time, the call will be dispatched to M or to some method that overrides M. Statically-dispatched, or direct, calls are handled by similar rules not shown here. Rule (4) defines the type of the call expression to be the same as M's return type.[2] Furthermore, the type $[E_i]$ of each actual parameter E_i must be the same as, or a subtype of, the type $[\mathit{Param}(M, i)]$ of the corresponding formal parameter $\mathit{Param}(M, i)$ (Rule (5)), and a method with the same signature as M must be declared in $[E_0]$ or one of its supertypes (Rule (6)). Note that Rule (6) relies on Definitions 2.1 and 2.2 below to determine a set of methods $M_1, \ldots, M_k$ overridden by M, and requires $[E_0]$ to

[2]Rules (2), (4), (7), and (19) *define* the type of certain kinds of expressions. While not very interesting by themselves, these rules are essential for defining the relationships between the types of expressions and declaration elements.

be a subtype of one or more[3] of $Dcl(M_1), \cdots, Dcl(M_k)$.

Definition 2.1. (Overriding) A method M in type C overrides a method M' in type B if M and M' have identical signatures and C is equal to B or C is a subtype of B.

Definition 2.2. (*RootDefs*) Let M be a method. Define:

$$RootDefs(M) = \{\ M' |\ M \text{ overrides } M', \text{ and there exists no } M''\ (M'' \neq M') \text{ such that } M' \text{ overrides } M''\ \}$$

The "is equal to" clause in Definition 2.1 ensures that *RootDefs* is not empty.

Example 2.3. Let us assume that we have a class hierarchy in which `C` is a subclass of `B`, where `C` also implements an interface `I`, and where a method `f()` is declared in `I` and `B`, but not in `C`. Then, we have that *RootDefs*(`C.f()`) = {`B.f()`, `I.f()`}.

Rules (7)–(8) are concerned with calls to constructor methods, and are analogous to the previously presented Rules (4)–(6).

Changing a parameter's type may affect virtual dispatch (and program) behavior, even if it does not affect type correctness. Hence, we require that the overriding relationships remain the same as they were in the original program: corresponding parameters of overriding methods were identical according to Definition 2.1 and must remain so (Rule (9)). For Java 5.0 and later, Rule (10) allows return types in overriding methods to be covariant. Rule (11) disallows solutions where two methods with the same name and signature end up in the same class. Similarly, Rule (12) prevents changes to hiding (shadowing) relationships between fields with the same name.

Rule (13) states that for a statement `return` E that occurs within the body of a method M, the type of expression E must be a subtype of method M's declared return type, $[M]$.

Rules (14)–(16) are concerned with cast expressions[4]. We define the type of a cast expression $(T)E$ to be type T (Rule (14)). Then, there are two cases: Rule (15) applies to downcasts in the original program and Rule (16) applies to upcasts. A cast is $(T)E$ is a *downcast* if T is a subtype of the type of E in the original program. Otherwise, if T is equal to, or a supertype of the type of E, it is an *upcast*. In each case, we generate a constraint that preserves the "direction" of the cast, ensuring that downcasts remain downcasts (or become no-ops), and upcasts remain upcasts[5].

In general, each refactoring must preserve the exceptional behavior of casts, i.e., the situations in which executing a cast results in a `ClassCastException`. For the

[3]In cases where a referenced method is not defined in any supertype of $[E]$, the *RootDefs*-set defined in Definition 2.2 will be empty, and an **or**-constraint with zero branches will be generated. Such constraints are never satisfied and do not occur in our setting because we assume the original program to be type-correct, and because Definition 2.1 states that a method overrides itself.

[4]The constraints for casts as shown in the figure are slightly more strict than what is necessary. In the presence of interface inheritance, casting between types that are neither subtypes nor supertypes of each other is allowed. Our implementation correctly handles this case.

[5]These rules are slightly more strict than what is necessary for preserving behavior for most refactorings we consider in this paper, but having this distinction between downcasts and upcasts makes the presentation of the type constraint systems used for different refactorings more uniform.

refactorings presented in Section 3 and 4, this does not require additional rules because these refactorings do not affect the run-time type of objects created by the program. The refactoring of Section 5 does affect object allocation, and relies on additional rules for preserving cast behavior.

Rules (15) and (16) use "$[(T)E]$" instead of just "T" to accommodate different definitions of $[(T)E]$ than that in Rule (14) later on. For example, rules ($R14_a$) and ($R14_b$) in Figure 26(a) provide an alternative definition of $[(T)E]$ that enables the migration of legacy types.

Rules (17) and (18) state that `Object` is a supertype of any type T, and that the type of the `null` expression is a subtype of any type T, respectively. Rule (19) defines the type of a `this` expression to be the class that declares the associated method.

3. REFACTORINGS FOR GENERALIZATION

Figure 3 shows a Java program that was designed to illustrate the issues posed by several different refactorings. The program declares a class `Stack` representing a stack, with methods `push()`, `pop()`, `isEmpty()`, and `contains()` with the expected behaviors, methods `moveFrom()` and `moveTo()` for moving an element from one stack to another, and a static method `print()` for printing a stack's contents. Also shown is a class `Client` that creates a stack, pushes the integers `1`, `2` and `3` onto it, and then creates another stack onto which it pushes the floating point value `4.4`. Two elements of the first stack are then moved to the second, the contents of the second stack are printed, and the elements of the first stack are transferred into a `Vector` whose contents are displayed in a tree. Hence, executing the program creates a graphical representation of a tree with a single node containing the value 1.

3.1 Extract Interface

One possible criticism about the code in Figure 3 is the fact that class `Client` explicitly refers to class `Stack`. Such explicit dependences on concrete data structures are generally frowned upon because they make code less flexible. The Extract Interface refactoring addresses this issue by (i) introducing an interface that declares a subset of the methods in a class and (ii) updating references in client code to refer to the interface instead of the class wherever possible. This change would enable the programmer to vary the implementation of the stack without having to change any reference that uses the interface type.

As an example, let us assume that the programmer has decided that it would be desirable to create an interface `IStack` that declares all of `Stack`'s instance methods, and to update references to `Stack` to refer to `IStack` instead. In the resulting code in Figure 4, the code fragments changed by the application of Extract Interface are underlined. Observe that `s1`, `s3`, and `s4` are the only variables for which the type has been changed to `IStack`. Changing the type of `s2` or `s5` to `IStack` would result in type errors. In particular, changing `s5`'s type to `IStack` results in an error because field `v2`, which is not declared in `IStack`, is accessed from `s5` on line 49.

Using type constraints, it is straightforward to compute the declarations that can be updated to refer to `IStack` instead of `Stack`. In this case of the Extract Interface refactoring, we are looking for a way of assigning types to constraint

```
class Client {
  public static void main(String[] args){
    Stack s1 = new Stack();
    s1.push(new Integer(1));
    s1.push(new Integer(2));
    s1.push(new Integer(3));
    Stack s2 = new Stack();
    s2.push(new Float(4.4));
    s2.moveFrom(s1);
    s1.moveTo(s2);
    Stack.print(s2);
    Vector v1 = new Vector(); /* A1 */
    while (!s1.isEmpty()){
      Integer n = (Integer)s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    JTree tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
class Stack {
  private Vector v2;
  public Stack(){
    v2 = new Vector(); /* A2 */
  }
  public void push(Object o1){
    v2.addElement(o1);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(Stack s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public boolean contains(Object o2){
    return v2.contains(o2);
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    while (e.hasMoreElements())
      System.out.println(e.nextElement());
  }
}
```

Fig. 3. Example program P_1. The allocation sites for the two `Vector`s created by this program have been labeled `A1` and `A2` to ease the discussion of the Replace Class refactoring in Section 5.

```
class Client {
  public static void main(String[] args){
    IStack s1 = new Stack();
    s1.push(new Integer(1));
    s1.push(new Integer(2));
    s1.push(new Integer(3));
    Stack s2 = new Stack();
    s2.push(new Float(4.4));
    s2.moveFrom(s1);
    s1.moveTo(s2);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Integer n = (Integer)s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    Component tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(IStack s3);
  public void moveTo(IStack s4);
  public boolean isEmpty();
}
class Stack implements IStack {
   private Vector v2;
   public Stack(){
     v2 = new Vector();
   }
   public void push(Object o1){
     v2.addElement(o1);
   }
   public Object pop(){
     return v2.remove(v2.size()-1);
   }
   public void moveFrom(IStack s3){
     this.push(s3.pop());
   }
   public void moveTo(IStack s4){
     s4.push(this.pop());
   }
   public boolean isEmpty(){
     return v2.isEmpty();
   }
   public boolean contains(Object o2){
     return v2.contains(o2);
   }
   public static void print(Stack s5){
     Enumeration e = s5.v2.elements();
     while (e.hasMoreElements())
       System.out.println(e.nextElement());
   }
}
```

Fig. 4. The example program of Figure 3 after applying Extract Interface to class `Stack` (code fragments affected by this step are underlined), and applying Generalize Declared Type to variable `tree` (the affected code fragment is shown boxed).

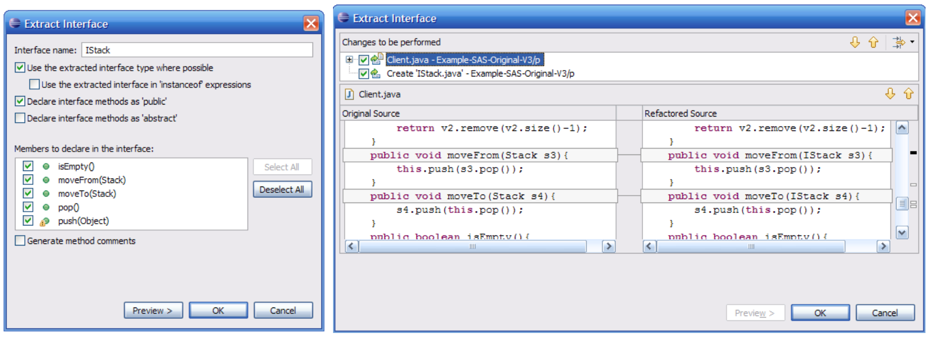

Fig. 5. Screenshots of the Extract Interface refactoring in Eclipse.

program construct	implied type constraint
declaration of method M (declared in type T)	$Dcl(M) = T$ (EI20)
declaration of field F (declared in type T)	$Dcl(F) = T$ (EI21)

Fig. 6. Additional constraint generation rules for Extract Interface.

variables that would maximize the use of the extracted interface while preserving type correctness. Furthermore, Extract Interface should not change the location of fields and methods in the class hierarchy. The latter requirement is enforced by the rules of Figure 6, which express that all constraint variables of the forms $Dcl(F)$ and $Dcl(M)$ should be bound to the type that originally declared the member under consideration.

Figure 5 shows some screenshots of the user-interface for the Extract Interface refactoring in Eclipse. The window shown on the left appears when a programmer selects a class, and then activates the `Refactor->Extract Interface...` menu entry. In this window, the programmer has to select the methods that should be declared in the extracted interface. After pressing the `Preview` button, the tool computes the set of variables that can be given type `IStack` using the previously described algorithm, and displays the corresponding source code modifications in a separate window. The programmer can then confirm the proposed changes by pressing the `OK` button. All methods were selected in the left window of Figure 5, which resulted in the suggested code modifications as shown in the right window in Figure 5.

We will now explain how the refactoring tool of Figure 5 automatically determines how the source code should be transformed. Consider Figure 7, which shows the relevant type constraints generated for declarations and expressions of type `Stack` in the program of Figure 3, according to the rules of Figures 2 and 6. It is important to note that these constraints were generated *after* adding interface `IStack` to the class hierarchy.

Now, from the constraints of Figure 7, it is easy to see that:

$$\texttt{Stack} \leq [\texttt{s2}] \leq [\texttt{s5}] \leq Dcl(\texttt{Stack.v2}) = \texttt{Stack}$$

line	constraint	rule
3	Stack≤[s1]	(7) (1)
4,5,6	[s1]≤*Dcl*(IStack.push())	(6)
7	Stack≤[s2]	(7) (1)
8	[s2]≤*Dcl*(IStack.push())	(6)
9	[s2]≤*Dcl*(IStack.moveFrom())	(6)
9 36	[s1]≤[s3]	(5)
10	[s1]≤*Dcl*(IStack.moveTo())	(6)
10 39	[s2]≤[s4]	(5)
11 48	[s2]≤[s5]	(5)
13	[s1]≤*Dcl*(IStack.isEmpty())	(6)

line	constraint	rule
14	[s1]≤*Dcl*(IStack.pop())	(6)
37	[s3]≤*Dcl*(IStack.pop())	(6)
40	[s4]≤*Dcl*(IStack.push())	(6)
49	[s5]≤*Dcl*(Stack.v2)	(3)
26	*Dcl*(Stack.v2)=Stack	(EI21)
30	*Dcl*(Stack.push())=Stack	(EI20)
33	*Dcl*(Stack.pop())=Stack	(EI20)
36	*Dcl*(Stack.moveFrom())=Stack	(EI20)
39	*Dcl*(Stack.moveTo())=Stack	(EI20)
42	*Dcl*(Stack.isEmpty())=Stack	(EI20)
—	*Dcl*(IStack.push())=IStack	(EI20)
—	*Dcl*(IStack.pop())=IStack	(EI20)
—	*Dcl*(IStack.moveFrom())=IStack	(EI20)
—	*Dcl*(IStack.moveTo())=IStack	(EI20)
—	*Dcl*(IStack.isEmpty())=IStack	(EI20)

Fig. 7. Type constraints generated for the application of the Extract Interface refactoring to the program of Figure 3 (only nontrivial constraints related to variables `s1`–`s5` are shown).

and hence that the types of `s2` and `s5` have to remain `Stack`. However, the types of `s1` and `s3` are less constrained:

$$[\texttt{s1}] \leq \mathit{Dcl}(\texttt{IStack.push()}) = \texttt{IStack}$$
$$[\texttt{s1}] \leq \mathit{Dcl}(\texttt{IStack.moveTo()}) = \texttt{IStack}$$
$$[\texttt{s1}] \leq \mathit{Dcl}(\texttt{IStack.isEmpty()}) = \texttt{IStack}$$
$$[\texttt{s1}] \leq \mathit{Dcl}(\texttt{IStack.pop()}) = \texttt{IStack}$$
$$[\texttt{s1}] \leq [\texttt{s3}] \leq \mathit{Dcl}(\texttt{IStack.pop()}) = \texttt{IStack}$$

implying that type `IStack` may be used for these variables. Note that, in general, the types of variables cannot be changed independently. For example, changing `s1`'s type to `IStack` but leaving `s3`'s type unchanged results in a type-incorrect program; this fact is captured by Rule (5).

Our algorithm for computing the set of declarations that can be updated is based on the above observations. In presenting the algorithm, *declaration element* refers to declarations of local variables, parameters in static, instance, and constructor methods, fields, and method return types, and to type references in cast expressions. Moreover, $\mathit{All}(P, C)$ denotes the set of all declaration elements of type C in program P.

Example 3.1. For the program P_1 of Figure 3, we have:

$$\mathit{All}(P_1, \texttt{Stack}) = \{\texttt{s1}, \texttt{s2}, \texttt{s3}, \texttt{s4}, \texttt{s5}\}$$

Definition 3.2 below defines $\mathit{TC}_{\mathrm{EI}}(P)$ to be the set of all type constraints generated for program P, according to the rules of Figure 2 and 6.

Definition 3.2 $\mathit{TC}_{\mathrm{EI}}(P)$. Let P be a program. Then, $\mathit{TC}_{\mathrm{EI}}(P)$ denotes the set of type constraints inferred for program P according to Rules (1)–(19) of Figure 2 and Rules (EI20)–(EI21) of Figure 6.

Example 3.3. Figure 7 shows the constraints in $\mathit{TC}_{\mathrm{EI}}(P_1)$ that pertain to expressions of type `Stack`.

Definition 3.4 below defines the set of declarations of type C whose type *cannot* be updated to refer to an interface I. This is accomplished by first identifying those declaration elements in program P for which changing the type to the new interface would violate a constraint in $TC_{\mathrm{EI}}(P)$. Then, additional declaration elements are identified as "non-updatable" because they are involved in constraints with declaration elements that were previously identified as non-updatable.

Definition 3.4 *Non-Updatable declaration elements.* Let P be a program, let C be a class in P, and let I be an interface in P such that C is the only class that implements I and I does not have any supertypes other than `Object`. Define:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{NonUpdatable}(P,C,I) = \{\, E \mid E \in \mathit{All}(P,C), \text{ and}} \\
\quad (\ (\text{``}[E] \leq T_1 \ \textbf{or} \ \cdots \ \textbf{or} \ [E] \leq T_k\text{''} \in TC_{\mathrm{EI}}(P), I \not\leq T_1, \cdots, I \not\leq T_k, k \geq 1) & \textbf{(a)} \\
\quad\ \text{or} & \\
\quad\ \text{``}[E] = C\text{''} \in TC_{\mathrm{EI}}(P) & \textbf{(b)} \\
\quad\ \text{or} & \\
\quad\ (\text{``}[E] \leq [E']\text{''} \in TC_{\mathrm{EI}}(P), E' \notin \mathit{All}(P,C), I \not\leq [E']) & \textbf{(c)} \\
\quad\ \text{or} & \\
\quad\ (\text{``}[E] = [E']\text{''} \in TC_{\mathrm{EI}}(P) \text{ or} & \textbf{(d)} \\
\quad\ \ \text{``}[E] \leq [E']\text{''} \in TC_{\mathrm{EI}}(P) \text{ or} & \\
\quad\ \ \text{``}[E] < [E']\text{''} \in TC_{\mathrm{EI}}(P),\ E' \in \mathit{NonUpdatable}(P,C,I))\ \ )\,\} &
\end{array}
$$

Part (a) of Definition 3.4 is concerned with constraints that are due to a method call $E.m(\cdots)$, and reflects situations where E cannot be given type I because a declaration of $m(\cdots)$ does not occur in (a supertype of) I. Part (b) reflects situations where the type of a declaration element is constrained to be exactly C. Such constraints may be generated due to several program constructs, including object allocation, and also due to the rules presented in Figure 6. Part (c) of Definition 3.4 deals with constraints $[E] \leq [E']$ due to assignments and parameter passing, and states that E cannot be given type I if the declared type of E' is not C, and I is not equal to or a subtype of E'. The latter condition is needed for situations where a declaration element of type C is assigned to a declaration element of type `Object`. Part (d) handles the propagation of "non-updatability" due to overriding, assignments, and parameter passing.

Example 3.5. For example program P_1 of Figure 3, we find that:

$$\mathit{NonUpdatable}(P_1, \texttt{Stack}, \texttt{IStack}) = \{\texttt{s2}, \texttt{s5}\}$$

This implies that the type of variables `s1`, `s3`, and `s4` can be updated to `IStack`. The corresponding changes to the source code were shown earlier in Figure 4.

3.2 Generalize Declared Type

Another possible criticism of the program of Figure 3 is the fact that the declared types of some variables are overly specific. This is considered undesirable because it reduces flexibility. The Generalize Declared Type refactoring in Eclipse lets a programmer select a declaration, and determines whether its type can be generalized without introducing type errors or behavioral changes. If so, the programmer may choose from the alternative permissible types. Using this refactoring, the type of variable `tree` can be updated to refer to `java.awt.Component` instead of `javax.swing.JTree` without affecting type correctness or program behavior, as

line	constraint	rule applied
20	JTree≤[tree]	(7),(1)
21	[tree]≤java.awt.Component	(8)
12	Vector≤[v1]	(7),(1)
15	[v1]≤*Dcl*(java.util.Collection.add())	(6)
20	[v1]≤Vector	(8)
28	Vector≤[v2]	(7),(1)
31	[v2]≤*Dcl*(java.util.Vector.addElement())	(6)
34, 43	[v2]≤*Dcl*(java.util.Collection.remove())	(6)
—	*Dcl*(java.util.Collection.add()) = java.util.Collection	(EI20)
—	*Dcl*(java.util.Vector.addElement()) = java.util.Vector	(EI20)
—	*Dcl*(java.util.Collection.remove()) = java.util.Collection	(EI20)

Fig. 8. Type constraints used for the application of GENERALIZE DECLARED TYPE (only constraints related to variables `tree`, `v1`, and `v2` are shown). Line numbers refer to Figure 3, and rule numbers to rules of Figures 2 and 6.

is indicated by a box in Figure 4. This, in turn, would enable one to vary the implementation to use, say, a `JList` instead of a `JTree` in `Client.main()`.

However, in some situations, the type of a variable cannot be generalized. For example, changing the type of `v2` to `Collection` or to any other supertype of `Vector` would result in a type error because line 31 invokes the method `addElement()`, which is not declared in any supertype of `Vector`. Furthermore, the type of `v1` cannot be generalized because line 20 passes `v1` as an argument to the constructor `JTree(Vector)`. `JTree` is part of the standard Java libraries for which we cannot change the source code, and the fact that its constructor expects a `Vector` implies that a more general type cannot be used.

Figure 8 shows the constraints generated from the example program of Figure 3 for variables `tree`, `v1`, and `v2`. Note that, for parameters of methods in external classes such as the constructor of `JTree`, we must include constraints that constrain these parameters to have their originally declared type, because the source code in class libraries cannot be changed. Therefore, we have that:

$$\texttt{JTree} \leq [\texttt{tree}] \leq \texttt{java.awt.Component}$$
$$\texttt{Vector} \leq [\texttt{v1}] \leq \texttt{Vector}$$
$$\texttt{Vector} \leq [\texttt{v2}] \leq \texttt{Vector}$$

In other words, the types of `v1` and `v2` must be exactly `Vector`, but for `tree` we may choose any supertype of `JTree` that is a subtype of `java.awt.Component`: `javax.swing.JComponent`, `java.awt.Container`, or `java.awt.Component`. The user-interface of GENERALIZE DECLARED TYPE in Eclipse includes a tree view of the class hierarchy, in which the names of permissible types are shown in black, and where non-permissible types are "grayed out", as is shown in Figure 9. Figure 4 shows the refactored code after selecting `java.awt.Component`, which is the most general of the three permissible types.

3.3 PULL UP MEMBERS

PULL UP MEMBERS is a refactoring for moving fields and methods from a class to its superclass, without changing type declarations. We will illustrate this refactoring

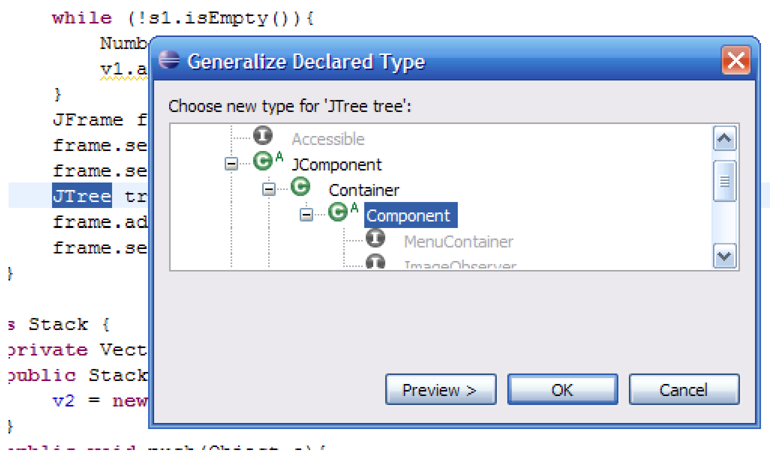

Fig. 9. Screenshot of the GENERALIZE DECLARED TYPE refactoring in Eclipse.

using the example program of Figure 10, which is a variation on the example program of Figure 3. In particular, we now have a class `BasicStack` that declares the methods `push()`, `pop()`, and `isEmpty()`, and a subclass `FullStack` of `BasicStack` that declares methods `moveFrom()` and `print()`. Note that `moveFrom()` has been changed to return `this` in order to enable the chaining of method calls, and `print()` has been changed to an instance method.

Now suppose that the programmer has decided that it would be desirable to migrate the `moveFrom()` method from `FullStack` to `BasicStack`. Does moving this method from `FullStack` to `BasicStack` preserve type correctness and program behavior? In general, moving members from a type to its supertype may result in type errors because the type of the special variable `this` is defined as the class in which the surrounding method is declared. Moving a method from a type to its supertype *changes* the type of `this`, which may result in type errors if the type of `this` is constrained otherwise (e.g., when `this` is used as the receiver of method calls, or as a return value).

To answer the specific question of whether the `moveFrom()` method can be pulled up into class `BasicStack`, we need a slightly different set of type constraints than what we used previously. This is the case because here, we need to permit changing the locations of members in the class hierarchy while leaving the declared types of members unchanged. To accomplish this, we add type constraint rules for "fixing" the types of declarations and fields, and of method return types as is shown in Figure 11. Definition 3.6 below defines $TC_{\mathrm{PM}}(P)$ to be the set of all type constraints generated for a program P, according to the rules of Figures 2 and 11.

Definition 3.6 $TC_{\mathrm{PM}}(P)$. Let P be a program. Then, $TC_{\mathrm{PM}}(P)$ denotes the set of type constraints inferred for program P according to Rules (1)–(19) of Figure 2 and Rules (PM22)–(PM24) of Figure 11.

Figure 12 shows the set of type constraints $TC_{\mathrm{PM}}(P_2)$ that was generated for the

```
1:  class BasicStack {
2:    public BasicStack(){
3:      v2 = new Vector();
4:    }
5:    public void push(Object o){
6:      v2.addElement(o);
7:    }
8:    public Object pop(){
9:      return v2.remove(v2.size()-1);
10:   }
11:   public boolean isEmpty(){
12:     return v2.isEmpty();
13:   }
14:   protected Vector v2;
15: }
16: class FullStack extends BasicStack {
17:   public FullStack(){
18:     super();
19:   }
20:   public FullStack moveFrom(FullStack s3){
21:     this.push(s3.pop());
22:     return this;
23:   }
24:   public void print(){
25:     Enumeration e = v2.elements();
26:     while (e.hasMoreElements())
27:       this.printElement(e);
28:   }
29:   private void printElement(Enumeration e){
30:     System.out.println(e.nextElement());
31:   }
32: }
```

Fig. 10. Example program P_2.

program construct	**implied type constraint**	
explicit declaration of variable or method parameter $T\ v$	$[v] = T$	(PM22)
declaration of method M with return type T	$[M] = T$	(PM23)
declaration of field F with type T	$[F] = T$	(PM24)

Fig. 11. Additional constraint generation rules for PULL UP MEMBERS.

line	**constraint**	**rule applied**
20	$[\texttt{FullStack.moveFrom()}] = \texttt{FullStack}$	(PM23)
22	$[\texttt{this}] \leq [\texttt{FullStack.moveFrom()}]$	(13)
22	$[\texttt{this}] = \mathit{Dcl}(\texttt{FullStack.moveFrom()})$	(19)
27	$[\texttt{this}] \leq \mathit{Dcl}(\mathit{FullStack.printElement}())$	(6)
27	$[\texttt{this}] = \mathit{Dcl}(\texttt{FullStack.print()})$	(19)

Fig. 12. Type constraints used for applications of PULL UP MEMBERS to the example program P_2 of Figure 10 (only constraints related to method declarations are shown). Line numbers refer to Figure 10, and rule numbers to rules of Figures 2 and 11.

example program P_2 of Figure 10. From these constraints, it can be seen that:

$$\mathit{Dcl}(\texttt{FullStack.moveFrom()}) = [\texttt{this}] \leq [\texttt{FullStack.moveFrom()}] = \texttt{FullStack}$$

In other words, the method `moveFrom()` must remain declared in `FullStack` or a subtype of `FullStack`, of which there are none in this example. Any attempt to move `moveFrom()` into class `BasicStack` would render the return statement `return this` on line 22 type-incorrect, and applying the PULL UP MEMBERS refactoring to `moveFrom()` produces an error message indicating that the refactoring cannot be performed safely.

From the constraints in Figure 12, it can also be seen that:

$$\mathit{Dcl}(\texttt{FullStack.print()}) = [\texttt{this}] \leq \mathit{Dcl}(\texttt{FullStack.printElement()})$$

indicating that `printElement()` must be declared in a supertype of the type in which `print()` is declared. In other words, applying Pull Up Members to method `print()` by itself results in a type-incorrect program, and is therefore not allowed. However, applying Pull Up Members to the methods `print()` and `printElement()` simultaneously is permitted.

3.4 Other refactorings for generalization

It is clear that `moveFrom()` can be pulled up in Figure 10 if its return type and the type of its parameter are changed to `BasicStack`. This can be done by first applying the Generalize Declared Type refactoring, followed by the Pull Up Members refactoring. For the simple code of Figure 10 it seems straightforward to combine both into one automated refactoring. However, this combination can become very complex when overridden methods or interface inheritance are present. For this reason, we believe it is better to keep the individual refactorings as simple as possible, and to handle complex cases by applying sequences of simple refactorings.

Two more such simple refactorings related to generalization have also been implemented. The Extract Superclass refactoring for extracting a superclass from a class is similar to Extract Interface in the sense that updatable declarations must be computed. The Use Supertype Where Possible refactoring enables programmers to replace references to a type with references to a supertype of that type. The Extract Superclass and Use Supertype Where Possible refactorings both rewrite multiple declarations to use a single new type and require the same analysis as the one we presented for Extract Interface. By contrast, Generalize Declared Type rewrites one declaration in isolation, but considers all possible types that can be given to that declaration, so for those reasons it requires a different analysis.

4. REFACTORINGS THAT INTRODUCE GENERICS

Generics were introduced in Java 5.0 to enable the creation of reusable class libraries with compiler-enforced type-safe usage. For example, an application that instantiates `Vector<E>` with, say, `String`, obtaining `Vector<String>`, can only add and retrieve `String`s. In the previous, non-generic version of this class, `Vector.get()` is declared to return `Object`. The compiler cannot ensure the type safety of vector operations, and therefore downcasts to `String` are needed to recover the type of retrieved elements. When a programmer makes a mistake, such downcasts fail at runtime, with `ClassCastException`s.

Java's generics have been designed with backward compatibility in mind. To this end, programmers are allowed to refer a parameterized class without explicitly specifying the type arguments that are bound to the formal type parameters of that class. This feature, commonly referred to as "raw types", essentially amounts to instantiating each formal type parameter with its bound, and enables existing applications to continue to work as before after library classes that they depend upon have become parameterized.

Donovan et al. [Donovan et al. 2004] identified two refactoring problems related to the introduction of generics. The *parameterization problem* consists of adding type parameters to an existing class definition so that it can be used in different contexts without the loss of type information. For instance, parameterization converts the declaration of `Vector` into `Vector<E>`. Once a class has been parameterized, the *instantiation problem* is the task of determining the type arguments that should be given to instances of the generic class in client code. For instance, instantiation may convert a use of `Vector` into `Vector<String>`. The former problem subsumes the latter, because the introduction of type parameters often requires the instantiation of generic classes. Section 4.2 presents the INFER GENERIC TYPE ARGUMENTS refactoring that solves the instantiation problem [Fuhrer et al. 2005]. Section 4.3 presents the INTRODUCE TYPE PARAMETER refactoring that solves the parameterization problem [Kieżun et al. 2007], given the programmer's selection of a declaration whose type is to be replaced with a new formal type parameter. As we shall see shortly, this may involve nontrivial changes to other declarations (e.g., by introducing wildcard types [Torgersen et al. 2004]).

4.1 Motivating Example

Figure 13 shows class `Stack` after applying both INTRODUCE TYPE PARAMETER and INFER GENERIC TYPE ARGUMENTS.

Applying INTRODUCE TYPE PARAMETER to the formal parameter of method `Stack.push()` causes the changes in the right column. For the purposes of this example, it is assumed that class `Stack` is analyzed in isolation. As can be seen in the figure, a new type parameter `T1` was added to class `Stack`, and `T1` is used as the type for field `v2`, the parameter of `Stack.push()`, and the return type of `Stack.pop()`. A more interesting change can be seen in the `moveFrom()`, `moveTo()`, and `print()` methods: their parameters now have wildcard types `Stack<? extends T1>`, `Stack<? super T1>`, and `Stack<?>`, respectively. As we shall see in the next paragraph, this allows for greater flexibility when refactoring class `Client` because it enables the transfer of elements between the two stacks without the loss of precision in the declared types of the stacks. Note also that the type of parameter `o2` of method `Stack.contains()` has remained `Object`. This is generally considered good Java style and is followed in the standard class libraries available with the Java Development Kit (JDK). It allows clients to call `contains()` with an argument of any type. The solution in which the type of `o2` is `T1`, while perhaps more intuitive, is actually overly restrictive. Consider for example a call `s1.contains(o)` where the type of variable `o` is `Object`. The call would make it impossible to instantiate `s1` as `Stack<Number>`.

The left column of Figure 13 shows the result of applying INFER GENERIC TYPE ARGUMENTS to the example program after the parameterization of `Stack`. The types of `s1` and `s2` are now `Stack<Integer>` and `Stack<Number>`, respectively. The downcast on line 14 that was present originally has been removed, which was enabled by the introduction of wildcard types in `Stack.moveFrom()` and `Stack.moveTo()`. If the formal parameters of these methods had been changed to `Stack<T1>` instead, Java's typing rules would have required `Vector<Number>` for the types of both `s1` and `s2`, thus making it impossible to remove the downcast.

4.2 INFER GENERIC TYPE ARGUMENTS

The INFER GENERIC TYPE ARGUMENTS refactoring requires extensions and changes to the type constraint formalism of Section 2. Most significantly, we need a new kind of constraint variable in order to reason about type parameters, as is shown in Figure 14. These *typeparam* constraint variables are of the form $T(x)$, representing

```
class Client {
  public static void main(String[] args){
    Stack<Integer> s1 = new Stack<Integer>();
    s1.push(new Integer(1));
    s1.push(new Integer(2));
    s1.push(new Integer(3));
    Stack<Number> s2 = new Stack<Number>();
    s2.push(new Float(4.4));
    s2.moveFrom(s1);
    s1.moveTo(s2);
    Stack.print(s2);
    Vector<Integer> v1 = new Vector<Integer>();
    while (!s1.isEmpty()){
      Integer n = (Integer) s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    JTree tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
```

```
class Stack<T1> {
   private Vector<T1> v2;
   public Stack(){
     v2 = new Vector<T1>();
   }
   public void push(T1 o1){
     v2.addElement(o1);
   }
   public T1 pop(){
     return v2.remove(v2.size()-1);
   }
   public void moveFrom(Stack<? extends T1> s3){
     this.push(s3.pop());
   }
   public void moveTo(Stack<? super T1> s4){
     s4.push(this.pop());
   }
   public boolean isEmpty(){
     return v2.isEmpty();
   }
   public boolean contains(Object o2){
     return v2.contains(o2);
   }
   public static void print(Stack<?> s5){
     Enumeration<?> e = s5.v2.elements();
     while (e.hasMoreElements())
       System.out.println(e.nextElement());
   }
}
```

Fig. 13. The example program of Figure 3 after refactorings to introduce generics. Underlining and strikethroughs indicate changes. Application of Introduce Type Parameter to the formal parameter of `Stack.push()` caused the changes in the right-hand column. Then, application of Infer Generic Type Arguments to the entire program caused the changes in the left-hand column.

$T(x)$	the type that is bound to formal type parameter T in the type of x

Fig. 14. Typeparam constraint variable used for solving the instantiation problem, in addition to the variables of Figure 1.

the type that is bound to formal type parameter T in the type of x. For example, if we have a parameterized class `Vector<E>` and a variable v of type `Vector<String>`, then $\texttt{E}(v) = \texttt{String}$.

Furthermore, the generation of type constraints for method calls now depends on whether or not the invoked method is declared in a parameterized class. This requires two changes. First, the rules of Figure 2 that govern method calls are restricted to methods declared in non-parameterized classes, using a predicate $\mathit{IsParameterizedType}(C)$, which returns true if and only if class C is a parameterized class. Figure 15 shows the updated rules for method calls and for overriding. The rules for constructor calls are updated similarly. Second, we introduce a new set of rules for generating constraints for calls to methods in parameterized classes. Section 4.2.1 presents some motivating examples, and Section 4.2.2 presents the definition of these rules.

4.2.1 *Examples.* We now give a few examples to illustrate what constraints are needed in the presence of calls to parameterized classes. In giving these examples, we assume that class `Stack` has already been parameterized as in the right column of

$$\frac{\begin{array}{c}\text{call } E \equiv E_0.m(E_1, \cdots, E_k) \text{ to method } M,\\ \mathit{RootDefs}(M) = \{M_1, \cdots, M_k\},\ E'_i \equiv \mathit{Param}(M, i),\ 1 \leq i \leq n,\\ \boxed{C = \mathit{Dcl}(M), \neg \mathit{IsParameterizedType}(C)}\end{array}}{\begin{array}{c}[E] = [M] \\ [E_i] \leq [E'_i]\end{array}} \quad \begin{array}{r}(\text{I4}_a)\\ (\text{I5}_a)\end{array}$$

$$\frac{\begin{array}{c}M' \text{ overrides } M,\ M' \neq M,\ 1 \leq i \leq \mathit{NumParams}(M'),\\ E_i \equiv \mathit{Param}(M, i),\ E'_i \equiv \mathit{Param}(M', i),\\ \boxed{C = \mathit{Dcl}(M), \neg \mathit{IsParameterizedType}(C)}\end{array}}{\begin{array}{c}[E_i] = [E'_i] \\ [M'] \leq [M] \\ \mathit{Dcl}(M') < \mathit{Dcl}(M)\end{array}} \quad \begin{array}{r}(\text{I9})\\ (\text{I10})\\ (\text{I11})\end{array}$$

Fig. 15. Updated type constraint rules for method calls and for method overriding. Changes from Figure 2 are shaded. Rules prefixed by a letter replace the previous versions. Subscripted rules are partial replacements; in this case, Figure 18 defines Rules (I4_b) and (I5_b). Omitted rules, such as Rule (6), are retained without change.

program construct	**constraints**	
method call E_1.`push`(E_2) to `void Stack<T1>.push(T1)`	$[E_2] \leq \texttt{T1}(E_1)$	(GEN-I25)
method call E.`pop`() to `T1 Stack<T1>.pop()`	$[E.\texttt{pop}()] = \texttt{T1}(E)$	(GEN-I26)
method call E_1.`moveFrom`(E_2) to `void Stack<T1>.moveFrom(Stack<? extends T1>)`	$\texttt{T1}(E_2) \leq \texttt{T1}(E_1)$	(GEN-I27)
method call E_1.`moveTo`(E_2) to `void Stack<T1>.moveTo(Stack<? super T1>)`	$\texttt{T1}(E_1) \leq \texttt{T1}(E_2)$	(GEN-I28)
method call E_1.`add`(E_2) to `boolean Vector<E>.add(E)`	$[E_2] \leq \texttt{E}(E_1)$	(GEN-I29)

Fig. 16. Additional constraint generation rules needed for the Infer Generic Type Arguments refactoring. Only constraints for methods used in the example program are shown. The constraint generation rules (GEN-I25)–(GEN-I29) are not built into the refactoring as with all other constraint generaton rules in this paper. Rather, they are specific to the program of Figure 3 and were automatically derived from method signatures using the rules of Figure 18.

Figure 13. This can be done either manually, or automatically using the Introduce Type Parameter refactoring that will be presented in Section 4.3.

Example 1. Consider the method call `s1.push(new Integer(1))` on line 4 in Figure 3. This call refers to the method `void Stack<T1>.push(T1 o)`. If `s1` is of a parameterized type, say, `Stack<`α`>`, then this call can only be type-correct if `Integer` $\leq \alpha$. This requirement is expressed by Rule (GEN-I25) in Figure 16.

Example 2. Similarly, the call `s1.pop()` on line 14 refers to method `T1 Stack<T1>.pop()`. If `s1` is of some parametric type, say `Stack<`α`>`, then $[\texttt{s1.pop()}] = \alpha$. This requirement is expressed by Rule (GEN-I26).

Example 3. Consider the call `s2.moveFrom(s1)` on line 9. If we assume that `s1` and `s2` are of parameterized types `Stack<`α_1`>` and `Stack<`α_2`>`, for some α_1, α_2, then the call is type-correct if we have that $\alpha_1 \leq \alpha_2$. This requirement is expressed by Rule (GEN-I27).

Figure 17 shows the constraints generated for the example of Figure 3 according to the rules of Figure 16. From these constraints, it follows that `Integer` $\leq$ `T1(s1)`,

line	constraint	rule
4,5,6	`Integer` $\leq$ `T1(s1)`	(7),(GEN-I25)
8	`Float` $\leq$ `T1(s2)`	(7),(GEN-I25)
9	`T1(s1)` $\leq$ `T1(s2)`	(GEN-I27)
10	`T1(s1)` $\leq$ `T1(s2)`	(GEN-I28)
14	[`s1.pop()`] = `T(s1)`	(GEN-I26)
14	`Integer` $\leq$ `E(v1)`	(GEN-I29)

Fig. 17. Type constraints generated for the example program using the rules of (a). Only nontrivial constraints relevant to the inference of type parameters in uses of `Stack` and `Vector` are shown. Line numbers refer to Figure 3, and rule numbers refer to Figures 16 and 2.

`Float` $\leq$ `T1(s2)`, and `T1(s1)` $\leq$ `T1(s2)`, and hence that `Integer` $\leq$ `T1(s2)`. Since `Number` is a common supertype of `Integer` and `Float`, a possible solution to this constraint system is:

`T1(s1)` $\leftarrow$ `Integer`, `T1(s2)` $\leftarrow$ `Number`

Generating the refactored source code that was shown in the left column of Figure 13 is now straightforward. The type of variable `s1` in the example program, for which we inferred `T1(s1)` = `Integer`, is rewritten to `Stack<Integer>`. Similarly, the types of `s2` and `v1` are rewritten to `Stack<Number>` and `Vector<Integer>`, respectively. Furthermore, all downcasts are removed for which the type of the expression being cast is a subtype of the target type. For example, for the downcast `(Integer)s1.pop()` on line 14, we inferred [`s1.pop()`] = `Integer` enabling us to remove the cast.

4.2.2 *Constraint Generation.* As can be seen from Figure 16, the rules for generating constraints have a regular structure, in which occurrences of type parameters in method signatures give rise to different forms of constraints, depending on where these references occur in the method signature and on whether or not wildcards are used. While such rules can be written by the programmer, this is tedious and error-prone. As before, our approach will be to generate type constraints directly from language constructs in the subject program. To this end, we generalize the constraint generation rules of Figure 2, as discussed below. Conceptually, these generalized rules "encode" the same case analysis on where type parameters occur in method signatures as the one that was used to construct the rules of Figure 16. However, we skip the intermediate step of generating these rules but instead generate the constraints directly.

Figure 18(a) shows how the previously shown Rules (4) and (5) of Figure 2 are adapted to handle calls to methods in generic classes. For a given call, Rule ($I4_b$) creates constraints that define the type of the method call expression, and Rule ($I5_b$) creates constraints that require the type of actual parameters to be equal to or a subtype of the corresponding formal parameters. A recursive helper function *CGen*, shown in Figure 18(b), serves to generate the appropriate constraints, and is defined by case analysis on its third argument, α'. Case (c1) applies when α' is a non-generic class, e.g., `String`. Case (c2) applies when α' is a type parameter. In the remaining cases the function is defined recursively. Cases (c3) and (c4) apply when α' is an upper or lower-bounded wildcard type, respectively. Finally, case (c5) applies when α' is a generic type.

We will now give a few examples that show how the rules of Figure 18 are used to generate type constraints such as those shown in Figure 17.

$$\frac{\text{call } E \equiv E_0.m(E_1,\ldots,E_k) \text{ to method } M, 1 \le i \le k \quad C = Dcl(M), IsParameterizedType(C)}{\begin{array}{ll} CGen([E],\ =,\ [M]_{\mathcal{P}},\ E_0)\ \cup & (\text{I}4_b) \\ CGen([E_i],\ \le,\ [\text{Param}(M,i)]_{\mathcal{P}},\ E_0) & (\text{I}5_b) \end{array}}$$

(a)

$$CGen(\alpha, op, \alpha', \alpha'') = \begin{cases} \{\alpha\ op\ C\} & \text{when } \alpha' \equiv\ C & \text{(c1)} \\ \{\alpha\ op\ T_i(\alpha'')\} & \text{when } \alpha' \equiv\ T_i & \text{(c2)} \\ CGen(\alpha, \le, \tau, \alpha'') & \text{when } \alpha' \equiv\ ?\ extends\ \tau & \text{(c3)} \\ CGen(\alpha, \ge, \tau, \alpha'') & \text{when } \alpha' \equiv\ ?\ super\ \tau & \text{(c4)} \\ \{\alpha\ op\ C\} \cup & \text{when } \alpha' \equiv C\langle\tau_1,\ldots,\tau_m\rangle & \text{(c5)} \\ CGen(W_i(\alpha), =, \tau_i, \alpha'') & \text{and } C \text{ is declared as} & \\ & C\langle W_1,\ldots,W_m\rangle, 1 \le i \le m & \end{cases}$$

(b)

$$\frac{\alpha_1 \le \alpha_2 \qquad T_1(\alpha_1) \text{ or } T_1(\alpha_2) \text{ exists}}{T_1(\alpha_1) = T_1(\alpha_2)} \quad (\text{I30}) \qquad \frac{\begin{array}{c} T_1(\alpha) \text{ exists} \\ C_1\langle T_1\rangle \ \texttt{extends/implements}\ C_2\langle T\rangle \\ C_2 \text{ is declared as } C_2\langle T_2\rangle \end{array}}{T_2(\alpha) = T(\alpha)} \quad (\text{I31})$$

(c)

Fig. 18. Extensions to the type constraint formalism of Figure 2 required by the Infer Generic Type Arguments refactoring, together with Figure 15. Part (a) shows how rules (4) and (5) of Figure 2 for method calls are adapted to handle calls to generic methods. These rules make use of an auxiliary function *CGen* that is shown in Part (b). Part (c) shows closure rules that impose constraints on actual type parameters due to language constructs such as assignments. In the rules of part (c), α, α_1, and α_2 denote constraint variables that are not type constants.

Example 1. Let us again consider the call `s1.push(new Integer(1))` on line 4 in Figure 3. Applying Rule ($\text{I}4_b$) of Figure 18 yields *CGen*([`s1.push(new Integer(1))`], =, `void`, `s1`), and applying case (c1) of the definition of *CGen* produces {[`s1.push(new Integer(1))`] = `void`}. Likewise, applying Rule ($\text{I}5_b$) yields *CGen*([`new Integer(1)`], ≤, `T1`, `s1`), and applying case (c2) and Rule (7) produces {`Integer` ≤ `T1(s1)`}. This result is shown on the first line of Figure 17.

Example 2. Consider the call `s2.moveFrom(s1)` to method `void Stack<T1>.moveFrom(Stack<? extends T1>)` on line 9. Applying Rule ($\text{I}4_b$) of Figure 18 yields *CGen*([`s2.moveFrom(s1)`], =, `void`, `s2`), and applying case (c1) of the definition of *CGen* produces {[`s2.moveFrom(s1)`] = `void`}. Furthermore, applying Rule ($\text{I}5_b$) produces *CGen*([`s1`], ≤, `Stack<? extends T1>`, `s2`), and an application of case (c5) produces {[`s1`] ≤ `Stack`} ∪ *CGen*(`T1(s1)`, =, `? extends T1`, `s2`). The second part of this term evaluates to *CGen*(`T1(s1)`, ≤, `T1`, `s2`) using an application of case (c3), and then to {`T1(s1)` ≤ `T1(s2)`} using case (c2). This result is shown in Figure 17. In summary, for the call to `moveFrom()` on line 9, the following set of constraints is generated: {[`s2.moveFrom(s1)`] = `void`, [`s1`] ≤ `Stack`, `T1(s1)` ≤ `T1(s2)`}.

4.2.3 *Closure Rules.* Thus far, we introduced additional typeparam constraint variables such as $T(E)$ to represent the actual type parameter bound to T in E's type, and we described how calls to methods in generic libraries give rise to constraints on typeparam variables. However, we have not yet discussed how types inferred for actual type parameters are constrained by language constructs such as assignments and parameter passing. For example, consider an assignment `a = b`, where `a` and `b` are both declared of type `Vector<E>`. The invariant subtyping on Java generics[6] implies that $\texttt{E(a)} = \texttt{E(b)}$. The situation becomes more complicated in the presence of inheritance relations between generic classes. Consider a situation involving class declarations[7] such as:

```
interface List<E_l> { ... }
class Vector<E_v> implements List<E_v> { ... }
```

and two variables, `c` of type `List` and `d` of type `Vector`, and an assignment `c = d`. This assignment can only be type-correct if the same type is used to instantiate $\texttt{E}_l$ in the type of `c` and $\texttt{E}_v$ in the type of `d`. In other words, we need a constraint $\texttt{E}_l(\texttt{c}) = \texttt{E}_v(\texttt{d})$. The situation becomes yet more complicated if generic library classes are assigned to variables of non-generic supertypes such as `Object`. Consider the program fragment:

```
Vector v1 = new Vector();
v1.add("abc");
Object o = v1;
Vector v2 = (Vector)o;
```

Here, we would like to infer $\texttt{E}_v(\texttt{v1}) = \texttt{E}_v(\texttt{v2}) = \texttt{String}$, which would require tracking the flow of actual type parameters through variable `o`[8].

The required constraints are generated by the closure rules of Figure 18(c). These rules infer, from an existing system of constraints, a set of additional constraints that unify the actual type parameters as outlined in the examples above. Rule (I30) states that, if a subtype constraint $\alpha_1 \leq \alpha_2$ exists, and another constraint implies that the type of α_1 or α_2 has formal type parameter T_1, then the types of α_1 and α_2 must have the same actual type parameter T_1[9]. This rule thus expresses the invariant subtyping among generic types. Observe that this has the effect of associating type parameters with variables of non-generic types, in order to ensure that the appropriate unification occurs in the presence of assignments to variables of non-generic types. For the example code fragment, a constraint variable $\texttt{E}_v(\texttt{o})$ is created by applying rule (I30). Values computed for variables that denote type arguments of non-generic classes (such as `Object` in this example) are disregarded at the end of constraint solution.

---

[6] In the presence of wildcard types, Java uses the more relaxed "containment" subtyping [Gosling et al. 2005]: `? extends Number` is *contained* in `? extends Object` and therefore `Set<? extends Number>` is a subtype of `Set<? extends Object>`). In this paper and in our implementation, we conservatively assume invariant subtyping even with wildcard types.

[7] In the Java collections library, the type formal parameters of both `Vector` and `List` have the same name, `E`. In this section, for disambiguation, we subscript them with v and l, respectively.

[8] In general, a cast to a parameterized type cannot be performed in a dynamically safe manner because type arguments are erased at run time. In this case, however, our analysis is capable of determining that the resulting cast to `Vector<String>` would always succeed.

[9] That is, unless wildcard types are inferred, which we do not consider in this section.

Rule (I31) is concerned with subtype relationships among generic library classes such as the one discussed above between classes `Vector` and `List`. The rule states that if a variable $T_1(\alpha)$ exists, then constraints are created to relate $T_1(\alpha)$ to the types of actual type parameters of its superclasses. For example, if we have two variables, `c` of type `List` and `d` of type `Vector`, and an initial system of constraints $[\mathtt{d}] \leq [\mathtt{c}]$ and $\mathtt{String} \leq \mathtt{E}_v(\mathtt{d})$, then using the rules of Figure 18(c), we obtain the additional constraints $\mathtt{E}_v(\mathtt{d}) = \mathtt{E}_v(\mathtt{c})$, $\mathtt{E}_l(\mathtt{d}) = \mathtt{E}_v(\mathtt{d})$, $\mathtt{E}_l(\mathtt{c}) = \mathtt{E}_l(\mathtt{d})$, and $\mathtt{E}_l(\mathtt{c}) = \mathtt{E}_v(\mathtt{d})$.

Note that we require that the constraint variables α, α_1, and α_2 in Rules (I30) and (I31) are not type constants. This requirement is necessary to ensure that different occurrences of a parameterized type can be instantiated differently. Type constants may arise due to several program constructs such as casts, method calls, and field accesses.

4.2.4 *Pragmatic Issues.* Several pragmatic issues needed to be addressed in our implementation of Infer Generic Type Arguments.

The constraint system is typically underconstrained, and there is usually more than one legal type associated with each constraint variable. For instance, the constraints shown in Figure 17 also allow the following uninteresting solution:

$$\mathtt{T1(s1)} \leftarrow \mathtt{Object},\ \mathtt{T1(s2)} \leftarrow \mathtt{Object}$$

When faced with a choice, our solver relies on heuristics to guide it towards preferred solutions. The two most significant of these heuristics are preferring more specific types over less specific ones, and avoiding marker interfaces such as `Serializable`.

In some cases, the actual type parameter inferred by our algorithm is equal to the bound of the corresponding formal type parameter, which typically is `Object`. Since this does not provide any benefits over the existing situation (no additional casts can be removed), our algorithm leaves raw any declarations and allocation sites for which this result is inferred. The opposite situation, where the actual type parameter of an expression is completely unconstrained, may also happen, in particular for incomplete programs. In principle, any type can be used to instantiate the actual type parameter, but since each choice is arbitrary, our algorithm leaves such types raw as well.

There are several cases where raw types must be retained to ensure that program behavior is preserved. When an application passes an object *o* of a generic library class to an external library[10], nothing prevents that library from writing values into *o*'s fields (either directly, or by calling methods on *o*). In such cases, we cannot be sure what actual type parameter should be inferred for *o*, and therefore generate an additional constraint that equates the actual type parameter of *o* to be the bound of the corresponding formal type parameter, which has the effect of leaving *o*'s type raw. Finally, Java does not allow creation of arrays of generic types [Bracha et al. 2004]. For example, `new Vector<String>[10]` is not allowed. Our algorithm generates constraints that equate the actual type parameter to the bound of the corresponding formal type parameter, which has the effect of preserving rawness. Section 6 presents a summary of results obtained with our implementation of Infer Generic Type Arguments on a suite of Java benchmarks.

[10]The situation where an application receives an object of a generic library type from an external library is analogous.

α	$::=$	α_{nw}	non-wildcard variable
		$?\,\texttt{extends}\,\alpha_{nw}$	wildcard type upper-bounded by α_{nw}
		$?\,\texttt{super}\,\alpha_{nw}$	wildcard type lower-bounded by α_{nw}
α_{nw}	$::=$	α_{ctxt}	contexts
		T	type parameter
		$\mathcal{I}_{\alpha_{ctxt}}(\alpha)$	α, interpreted in context α_{ctxt}
α_{ctxt}	$::=$	$[E]$	type of expression E
		$[M]$	return type of method M
		$[Param(M, i)]$	type of i^{th} formal parameter of method M
		C	monomorphic type constant

Fig. 19. Constraint variables used for the INTRODUCE TYPE PARAMETER refactoring.

4.3 INTRODUCE TYPE PARAMETER

The INTRODUCE TYPE PARAMETER refactoring requires a further adaptation of the type constraint formalism of Section 4.2. This adaptation includes replacing the *typeparam* constraint variables of Section 4.2 with the more general notion of a *context* constraint variable, the introduction of a new form of constraint variable called *wildcard variables*, changes to the constraint generation rules, and a specialized constraint solver. This section discusses these extensions and illustrates them on the class `Stack` of Figure 3. The preferred parameterization of this class was previously shown in the right column of Figure 13.

4.3.1 *Extensions to the Type Constraints Model.* Figure 19 shows how the formalism of Figure 1 is extended with two new kinds of constraint variables, *context variables* and *wildcard variables*, in order to accommodate the INTRODUCE TYPE PARAMETER refactoring. Note that we do not use the *typeparam* variables that were introduced in Section 4.2 in this section.

A *context variable* is of the form $\mathcal{I}_{\alpha'}(\alpha)$ and represents the *interpretation* of a constraint variable α in a *context* given by another constraint variable, α'. We give the intuition behind this new form of constraint variables by examples.

Example 4.1. Consider the JDK class `List<E>`. References to its type parameter `E` only make sense within the definition of `List`. In the context of an instance of `List<String>`, the interpretation of `E` is `String`, while in the context of an instance of `List<Number>`, the interpretation of `E` is `Number`.

Example 4.2. Consider the type `Stack<T1>` declared in Figure 13. For a variable `x` of type `Stack<Number>`, the interpretation of `T1` in the context of the type of `x` is `Number`. We will denote this fact by $\mathcal{I}_{[\texttt{x}]}(\texttt{T1}) = \texttt{Number}$. Here, $\mathcal{I}_{[\texttt{x}]}$ is an *interpretation function*. An interpretation function is subscripted by a constraint variable that corresponds to a program entity of a parameterized type, and maps each of the formal type parameters of that parameterized type to the types with which they are instantiated. For the example being considered, $\mathcal{I}_{[\texttt{x}]}$ maps the formal type parameter[11] `T1` of `Stack` to the type, `Number`, with which it is instantiated in

[11]For parameterized types with multiple type parameters such as `HashMap`, the interpretation function provides a binding for each of them [Kieżun et al. 2007].

type [x].

Example 4.3. Consider the method call `s1.push(new Integer(1))` on line 4 of Figure 3. For this call to be type-correct, the type `Integer` of actual parameter `new Integer(1)` must be a subtype of the formal parameter `o1` of `Stack.push()` *in the context of the type of* `s1`. This is expressed by the type constraint $\texttt{String} \leq \mathcal{I}_{[\texttt{s1}]}([\texttt{o1}])$. Note that it is incorrect to simply require that $\texttt{Integer} \leq [\texttt{o1}]$ because when `Stack` becomes a parameterized class `Stack<T1>`, and the type of `o1` becomes `T1`, then `T1` is out of scope at the call site. In addition, `Integer` is not a subtype of `T1`.

Note that, in some cases, a context α_{ctxt} is irrelevant. For example, $\mathcal{I}_{\alpha_{ctxt}}(\texttt{String})$ always resolves to `String`, regardless of the context α_{ctxt} in which it is interpreted.

Context variables can be viewed as a generalization of the typeparam variables that were introduced in Section 4.2. Recall that, in Section 4.2, a typeparam variable $E(x)$ was used for expressions `x` whose type was an instantiation of a parameterized class such as `Vector<E>`, where it denoted the type with which the formal type parameter `E` was instantiated. For example, if `x` is declared as `Vector<String>`, then `E<X>` = `String`. Context variables take this idea one step further and are concerned with situations where we are trying to *infer* a new type parameter for class `Vector`. The interpretation function $\mathcal{I}_{[\texttt{x}]}$ is a variable in our constraint system for which our solver will attempt to find a suitable value: a mapping from formal type parameters to the types with which they are instantiated. When such a mapping can be found, generating a name for such a new type parameter is trivial.

A *wildcard variable* is of the form $\texttt{? extends}\,\alpha$ or $\texttt{? super}\,\alpha$ (where α is another constraint variable), and is used in cases where Java's typing rules *require* the use of wildcard types. Consider the following class that extends the library class `java.util.Vector<E>`.

```
class SubVector extends Vector {
  @Override
  public boolean addAll(Collection c){ ··· }
}
```

In this example, `SubVector.addAll()` overrides `java.util.Vector.addAll()`. If `SubVector` becomes a generic class with formal type parameter `T`, then preserving this overriding relationship requires the formal parameter `c` of `SubVector.addAll()` to have the same type as that of `Vector.addAll()`, which is declared in the Java standard libraries as `Vector.addAll(Collection<? extends E>)`. Three parts of our algorithm work together to accomplish this: (i) the type of `c` is represented, using a context variable, as $\texttt{Collection<}\mathcal{I}_{[\texttt{c}]}(E)\texttt{>}$, (ii) type constraint generation produces $\mathcal{I}_{[\texttt{c}]}(E) = \texttt{? extends}\ \mathcal{I}_{\texttt{SubVector}}(E)$, which uses a wildcard variable, and (iii) constraint solving resolves $\mathcal{I}_{\texttt{SubVector}}(E)$ to `T`.

As we shall see shortly, our algorithm may also introduce wildcard types in cases where doing so results in a more flexible solution.

4.3.2 *Type Constraint Generation.* Figure 20 shows the changes to the constraint generation rules necessary for the Introduce Type Parameter refac-

$$\frac{\text{call } E \equiv E_0.m(E_1,\ldots,E_k) \text{ to method } M \quad \mathit{inScope} \equiv \mathit{Dcl}(M) \in \mathit{TargetClasses}, 1 \le i \le k}{\begin{array}{ll}\mathit{CGen}([E], =, [M], [E_0], \mathit{inScope}) \cup & \text{(P4)} \\ \mathit{CGen}([E_i], \le, [\mathrm{Param}(M,i)], [E_0], \mathit{inScope}) & \text{(P5)}\end{array}}$$

(a)

$$\mathit{CGen}(\alpha, op, \alpha', \alpha'', \mathit{inScope}) = \begin{cases} \{\alpha \ op \ C\} & \text{when } \alpha'_{\mathcal{P}} \equiv C \text{ and } \mathit{inScope} \equiv \mathit{false} & \text{(c1')} \\ \{\alpha \ op \ \mathcal{I}_{\alpha''}(\alpha')\} & \text{when } \alpha'_{\mathcal{P}} \equiv C \text{ and } \mathit{inScope} \equiv \mathit{true} & \text{(c1'')} \\ \{\alpha \ op \ \mathcal{I}_{\alpha''}(T)\} & \text{when } \alpha'_{\mathcal{P}} \equiv T & \text{(c2')} \\ \mathit{CGen}(\alpha, \le, [\tau'], \alpha'', \mathit{inScope}) & \text{when } \alpha'_{\mathcal{P}} \equiv \texttt{? extends } \tau' & \text{(c3')} \\ \mathit{CGen}(\alpha, \ge, [\tau'], \alpha'', \mathit{inScope}) & \text{when } \alpha'_{\mathcal{P}} \equiv \texttt{? super } \tau' & \text{(c4')} \\ \{\alpha \ op \ C\} \cup \bigcup_{1 \le i \le m} \mathit{CGen}(\mathcal{I}_{\alpha}(W_i), =, [\tau_i], \alpha'', \mathit{inScope}) & \text{when } \alpha'_{\mathcal{P}} \equiv C\langle\tau_1,\ldots,\tau_m\rangle \text{ and } C \text{ is declared as } C\langle W_1,\ldots,W_m\rangle & \text{(c5')} \end{cases}$$

(b)

$$\frac{\alpha_1 \le \alpha_2 \qquad \mathcal{I}_{\alpha_1}(\alpha) \text{ or } \mathcal{I}_{\alpha_2}(\alpha) \text{ exists}}{\mathcal{I}_{\alpha_1}(\alpha) = \mathcal{I}_{\alpha_2}(\alpha)} \quad \text{(P30)}$$

$$\frac{\alpha_1 \le \alpha_2 \qquad \mathcal{I}_{\alpha'}(\alpha_1) \text{ or } \mathcal{I}_{\alpha'}(\alpha_2) \text{ exists}}{\mathcal{I}_{\alpha'}(\alpha_1) \le \mathcal{I}_{\alpha'}(\alpha_2)} \quad \text{(P31)}$$

(c)

Fig. 20. Extensions to the type constraint formalism required by the INTRODUCE TYPE PARAMETER refactoring. Part (a) shows (P4) and (P5) that generalize the corresponding rules ($I4_a$), ($I4_b$), ($I5_a$), and ($I5_b$) of Figures 15 and 18(a). Here, *TargetClasses* is a set of classes that should be parameterized by adding type parameters. Part (b) shows a generalized version of the function *CGen* of Figure 18(b), which has been extended to handle context variables, and by taking an additional parameter *inScope*. Here, $\alpha_{\mathcal{P}}$ denotes the type of the program construct corresponding to α in the original program $\mathcal{P}$. Part (c) presents closure rules (P30) and (P31) that generalize the corresponding rules of Figure 18(c).

toring. Rules (P4) and (P5) in part (a) of the figure generate the appropriate constraints for method calls, and are adaptations of the corresponding rules in Figures 15 and 18. Rule (P4) states that the type of the method call expression is the same as the return type of the method (in the context of the receiver). Rule (P5) relates the actual and formal type parameters of the call. The *TargetClasses* set is a user-supplied input to the parameterization algorithm that indicates which classes should be refactored by adding type parameters. For example, in Figure 3, class `Stack` is in *TargetClasses*. The auxiliary function *CGen*, defined in Figure 20(b), actually generates constraints.

Java's type rules impose certain restrictions on parametric types. Closure rules such as (P30) and (P31) in Figure 20(c) enforce those restrictions. Rule (P30) enforces invariant subtyping of parametric types: $C\langle\tau\rangle$ is a subtype of $C\langle\tau'\rangle$ iff $\tau = \tau'$ (see discussion in footnote 6). Rule (P31) requires that, given two formal type parameters[12] T_1 and T_2 such that $T_1 \le T_2$ and any context α in which either actual type parameter $\mathcal{I}_\alpha(T_1)$ or $\mathcal{I}_\alpha(T_2)$ exists, the subtyping relationship

[12] These can also be constraint variables that could become formal type parameters.

Notation:

$\|Est\|(\alpha)$	a set of types, the type estimate of constraint variable α
$\alpha_{\mathcal{P}}$	type of constraint variable α in the original program
$Sub(\tau)$	set of all non-wildcard subtypes of τ
$\|Wild\|(X)$	set of wildcard types (both lower- and upper-bounded) for all types in type estimate X
U_E^S	universe of all types, including all wildcard types (i.e., both <u>s</u>uper and <u>e</u>xtends wildcards)

```
Subroutine parameterize():
  initialize()
  // P is a set of variables known to be type parameters
  P ⟵ {automatically- or user-selected variable}
  repeat until all variables have single-type estimates
    while P is not empty do
      α_‖tp| ⟵ remove element from P
      if ‖Est|(α_‖tp|) contains a type parameter then
        ‖Est|(α_‖tp|) ⟵ {type parameter from ‖Est|(α_‖tp|)}
      else
        ‖Est|(α_‖tp|) ⟵ {create new type parameter}
      propagate()
    if ∃α.|‖Est|(α)| > 1 then
      ‖Est|(α) ⟵ {select a type from ‖Est|(α)}
      propagate()

  // Set initial type estimate for each constraint variable
  Subroutine initialize():
    foreach non-context variable α do
      if α cannot have wildcard type then
        ‖Est|(α) = Sub(α_P)
      else
        ‖Est|(α) = Sub(α_P) ∪ ‖Wild|(Sub(α_P))
    foreach context variable I_α'(α) do
      ‖Est|(I_α'(α)) = U_E^S

    // Reconcile the left and right sides of each type inequality
    Subroutine propagate():
      repeat until fixed point (i.e., until estimates stop changing)
        foreach constraint α ≤ α' do
          Remove from ‖Est|(α) all types that are not a subtype of a type in ‖Est|(α')
          Remove from ‖Est|(α') all types that are not a supertype of a type in ‖Est|(α)
          if ‖Est|(α) or ‖Est|(α') is empty then
            stop: "No solution"
      foreach context variable I_α'(α) do
        if ‖Est|(I_α'(α)) is a singleton set with type parameter T and ‖Est|(α) does not
        contain T then
          add α to P
```

Fig. 21: Pseudo-code for the constraint solving algorithm.

$\mathcal{I}_\alpha(T_1) \leq \mathcal{I}_\alpha(T_2)$ must also hold. To illustrate this rule, consider a class `C<T1, T2 extends T1>` and any instantiation `C<C1, C2>`. Then, `C2` $\leq$ `C1` must hold, implying that, e.g., `C<Number, Integer>` is legal but that `C<Integer, Number>` is not.

4.3.3 *Algorithm.* A solution to the system of type constraints is computed using the iterative worklist algorithm of Figure 21. During solving, each variable α has

line	constraint	
31	[o1] $\leq$ E(v2)	(i)
34	E(v2) $\leq$ [Stack.pop()]	(ii)
37	$\mathcal{I}_{[s3]}$([Stack.pop()]) $\leq$ [o1]	(iii)
40	[Stack.pop()] $\leq$ $\mathcal{I}_{[s4]}$([o1])	(iv)
46	[o2] $\leq$ Object	(v)

(a)

constraint variable	inferred type
[o1]	T1
E(v2)	T1
[Stack.pop()]	T1
$\mathcal{I}_{[s3]}$([Stack.pop()])	? extends T1
$\mathcal{I}_{[s4]}$([o1])	? super T1
[o2]	Object

(b)

Fig. 22. (a) Type constraints generated for class `Stack` of Figure 3 when applying Introduce Type Parameter. (b) Solution to the constraints computed by our algorithm.

an associated type estimate *Est*(α). An estimate is a set of types. Each estimate is initialized to the set of all possible non-parametric types and shrinks monotonically as the algorithm progresses. When the algorithm terminates, each estimate consists of exactly one type. Because type estimates do not contain parametric types, they are finite sets, and algebraic operations such as intersection can be performed directly. As an optimization, our implementation uses a symbolic representation for type estimates.

The algorithm begins by initializing the type estimate for each constraint variable, at lines 2 and 15–22 in Figure 21. A workset P is used to contain those constraint variables that it has decided shall become type parameters, but for which that decision has yet to be executed. The set P is initially seeded with the constraint variable that corresponds to the declaration that is selected either by a heuristic or by the user on line 3. The inner loop of parameterize() on lines 5–11 repeatedly removes an element from P and sets its estimate to a singleton type parameter. For new type parameters, the upper bound is the declared type in the original (unparameterized) program.

Whenever a type estimate changes, those changes must be propagated through the type constraints, possibly reducing the type estimates of other variables as well. The propagate() subroutine performs this operation, ensuring that the estimates on both sides of a type constraint contain only types that are consistent with the relation. Whenever a context variable $\mathcal{I}_{\alpha'}(\alpha)$ gets resolved to a type parameter, α must also get resolved to type parameter on line 30. To see why, suppose that α gets resolved to a non-type parameter type, C. In that case, the context is irrelevant, and thus $\mathcal{I}_{\alpha'}(\alpha)$ also must get resolved to C (i.e., *not* a type parameter). This is a contradiction. Section 4.3.5 discusses an example that illustrates this situation.

4.3.4 *Example: Application of Algorithm to Class `Stack`.* Figure 22(a) shows the relevant set of constraints that our algorithm generates for class `Stack` of Figure 3. Figure 22(b) shows the result that the algorithm computes for the constraints in Figure 22(a). Our tool lets the user select a type reference to parameterize; Figure 22(b) assumes that the user selected the type of `o1` on line 31 of Figure 3. The solver works as follows. The solver creates a new type parameter `T1` for `o1` because the user selected the declaration of `o1`. Constraints (i) and (ii) in Figure 22 imply that `E(v2)` and [`Stack.pop()`] must each be a supertype of `T1`, and constraint (iii) implies that $\mathcal{I}_{[s3]}$([`Stack.pop()`]) must be a subtype of `T1`. The only possible choices for [`Stack.pop()`] are `T1` and `Object` because wildcard types are not permitted in this position (i.e., return type of a method), and the algorithm selects `T1` because choosing `Object` would lead to a violation of constraint (iii).

Taking into account constraint (ii), it follows that `E(v2) = T1`. Now, for $\mathcal{I}_{[\texttt{s3}]}$([`Stack.pop()`]), the algorithm may choose any subtype of `T1`, and it heuristically[13] chooses `? extends T1`. Likewise, the algorithm selects the type `? super T1` for $\mathcal{I}_{[\texttt{s4}]}$([`o1`]).

The type of variable `o2` is constrained only by `Object` according to constraint (v), and the solver therefore leaves the type unchanged. Note that this is the required solution for the parameter of the `contains()` method, as argued in Section 4.1.

The type estimates created during the constraint solution algorithm are all non-parametric, even for constraint variables that represent program entities whose type was parametric, such as `v2` in Figure 3, or will be parametric after refactoring, such as `s3` in Figure 3. Assembling these results into parametric types is straightforward. Figure 22(b) indicates that the type of `o1` and the return type of `Stack.pop()` become `T1`. Moreover, from `E(v2) = T1`, it follows that `v2` becomes `Vector<T1>`. The type of `s3` is rewritten to `Stack<? extends T1>` because the return type of `Stack.pop()` is `T1` and the type of $\mathcal{I}_{[\texttt{s3}]}$([`Stack.pop()`]) is `? extends T1`. By a similar argument, the type of `s4` is rewritten to `Stack<? super T1>`. The right column of Figure 13 shows the result.

A technical report [Kieżun et al. 2006] walks through a more detailed example of the solving algorithm.

4.3.5 *Example: Application of Algorithm to Interdependent Classes.* Often, parameterizing one class requires parameterizing other classes as well. For example, consider the code in Figure 13. Throughout Section 4.3 we assumed that the class `Vector` had been parameterized before (as part of the standard Java collections). However, had the class not been parameterized before, then to parameterize `Stack`, it would have been required to parameterize `Vector` as well.

The parameterization algorithm can parameterize multiple classes simultaneously. Lines 30–32 of the algorithm in Figure 21 handle propagating type parameters to interdependent classes. That part of the algorithm can be understood as follows. Whenever an estimate of a context variable is narrowed to a single type, i.e., the solver finds a solution for the context variable on line 30, the variable to which the context refers must also be assigned a type parameter on line 32.

For example, consider the scenario in which `Vector` is *not* parameterized at the time when we run the algorithm for `Stack`. Here we discuss the crucial part of the algorithm that enables parameterizing both classes. To generate a constraint for line 34 in Figure 13, our algorithm uses the constraint generation Rule (c1″) from Figure 20 and generates $\mathcal{I}_{[\texttt{Vector.remove(int)}]}$([`v2`]) $\leq$ [`Stack.pop()`]. This constraint corresponds to constraint (ii) from Figure 22 that gets generated when class `Vector` had been already parameterized. During solving, the type estimate of [`Stack.pop()`] narrows down to `T1`, as shown in Figure 22. At that time, the addition on step 32 in the solving algorithm marks [`Vector.remove(int)`] as a variable to which a new type parameter will be assigned. Thus, parameterizing

[13]Other possible choices include `T1`, or a new type parameter that is a subtype of `T1`. The paper by Kieżun et al. [Kieżun et al. 2007] presents more details on the use of heuristics.

class `Stack` results in parameterizing class `Vector`, as is necessary. The technical report [Kieżun et al. 2006] describes the issue of parameterizing interdependent classes on a more detailed example.

4.3.6 *Use of Heuristics in the Algorithm.* The algorithm of Figure 21 makes an underconstrained choice on lines 3, 6, 12, and 13. (On line 8, there is only one possibility.) Any choice yields a correct (behavior-preserving and type-safe) result, but some results are more useful to clients, for example by permitting the elimination of more casts. Our implementation makes an arbitrary choice at lines 6 and 12, but uses heuristics at lines 3 and 13 to guide the algorithm to a useful result.

On line 3, our tool lets a user select a type to parameterize. Alternatively, the tool can apply the following heuristic.

(1) If a generic supertype exists, use the supertype's signatures in the subtype. This is especially useful for customized container classes.
(2) Parameterize the return value of a "retrieval" method. A retrieval method's result is downcasted by clients, or it has a name matching such strings as `get` and `elementAt`. Even classes that are not collections often have such retrieval methods [Donovan et al. 2004].
(3) Parameterize the formal parameter to an insertion method. An insertion method has a name matching such strings as `add` or `put`.

The heuristic further forbids selecting these uses of types:

(1) Type uses that are not in the public interface of the class.
(2) Parameters of overridden methods (such as `equals`), unless their type in the overridden class is a type parameter. To preserve method overriding, types of such parameters must remain unchanged, and cannot be parameterized.
(3) Type uses in interfaces or abstract classes. Their uses tend to be underconstrained and can lead to sub-optimal results.

On line 13, the algorithm uses a heuristic that minimizes the use of casts in client code, while preserving flexibility in cases where this does not affect type safety. It prefers (in this order):

(1) types that preserve type erasure over those that do not,
(2) wildcard types over non-wildcard types, and
(3) type parameters over other types, but only if such a choice enables inference of type parameters for return types of methods.

To justify the latter restriction, observe that assigning a type parameter or a parametric type to a method return type is beneficial, because doing so reduces the need for casts in clients of the class. Otherwise, introducing type parameters simply increases the apparent complexity of the class for clients.

4.3.7 *Miscellaneous Issues.* Some classes are not parameterizable by any tool [Kieżun et al. 2006]. If the presented algorithm is applied to such a class (e.g., `String`), then the algorithm either signals that parameterization is impossible on line 28 in Figure 21) or else produces a result in which the type parameter is used in only one or two places. An implementation could issue a warning in this case. For example, consider the following class.

```
class C {
  public String getText() { return "hello"; }
}
```

If the return type of `getText` is selected for parameterization, the type parameter would have to have a concrete lower bound: `T super String`. Such type parameters are disallowed in Java. Line 28 in Figure 21 detects cases in which no solution can be found.

As noted before, lines 30–32 of Figure 21 handle inter-class dependencies. Interfaces and abstract classes are handled by the same mechanism, i.e., our algorithm creates type constraints to preserve method overriding and treats `implements` and `extends` relationships as other inter-class dependencies.

Our algorithm and implementation fully support parameterization in the presence of generic methods, e.g., those in `java.util.Collections`, but we have not yet implemented *adding* type parameters to methods[14].

Native methods pose no special problems to our analysis, which can conservatively approximate the flow of objects between argument expressions and the return value, based on the native method's signature.

5. A REFACTORING FOR REPLACING CLASSES

As applications evolve, classes are occasionally deprecated in favor of others with roughly the same functionality. In Java's standard libraries, for example, class `Hashtable` has been superseded by `HashMap`, and `Iterator` is now preferred over `Enumeration`. In such cases it is often desirable to migrate client applications to make use of the new idioms, but manually making the required changes can be labor-intensive and error-prone. In what follows, we will use the term *migration* to refer to the process of replacing the references to a *source class* with references to a *target class*.

In the program of Figure 3, the type `Vector` is used for the declaration of variable `v1` on line 12, and for that of field `v2` on line 26. Class `ArrayList` was introduced in the standard libraries to replace `Vector`, and is considered preferable because its interface is minimal and matches the functionality of the `List` interface. `ArrayList` also provides unsynchronized access to a list's elements whereas all of `Vector`'s methods are `synchronized`, which results in unnecessary overhead when `Vector`s are used by only one thread. The example program illustrates several factors that complicate the migration from `Vector` to `ArrayList`, which will be discussed next.

Example 5.1. Some methods in `Vector` are not supported by `ArrayList`. For example, the program of Figure 3 calls `Vector.addElement()` on line 31, a method not declared in `ArrayList`. In this case, the call can be replaced with a call to

[14]Von Dincklage and Diwan used heuristics to handle generic methods [von Dincklage and Diwan 2004]—such heuristics may also be applicable to our work. In previous work, we used a context-sensitive version of the generic instantiation algorithm to parameterize methods [Fuhrer et al. 2005].

```
(S1)  new Vector(), unsynchronized                    → new ArrayList()
(S2)  new Vector(), synchronized                      → Collections.synchronizedList(
                                                          new ArrayList())
(S3)  boolean Vector:receiver.add(Object:v)           → boolean receiver.add(v)
(S4)  void Vector:receiver.addElement(Object:v)       → boolean receiver.add(v)
(S5)  Object Vector:receiver.firstElement()           → Object receiver.get(0)
(S6)  Object Vector:receiver.remove(int:i)            → Object receiver.remove(i)
(S7)  int Vector:receiver.size()                      → int receiver.size()
(S8)  boolean Vector:receiver.contains(Object:o)      → boolean receiver.contains(o)
(S9)  boolean Vector:receiver.isEmpty()               → boolean receiver.isEmpty()
(S10) void Vector:receiver.copyInto(Object:array)     → void Util.copyInto(receiver, array)
(S11) Enumeration Vector:receiver.elements()          → Iterator receiver.iterator()
(S12) boolean Enumeration:receiver.hasMoreElements()  → boolean receiver.hasNext()
(S13) Object Enumeration:receiver.nextElement()       → Object receiver.next()
```

Fig. 23. Specification used for migrating the example program.

`ArrayList.add()`, but other cases require the introduction of more complex expressions, or preclude migration altogether.

Example 5.2. Opportunities for migration may be limited when applications interact with class libraries and frameworks for which the source code is not under our control. For example, variable `v1` declared on line 12 serves as the actual parameter in a call to a constructor `JTree(Vector)` on line 20. Changing the type of `v1` to any supertype of `Vector` would render this call type-incorrect. Hence, the allocation site on line 12, labeled `A1`, cannot be migrated to `ArrayList`.

Example 5.3. Migrating one class may require migrating another. Consider the call on line 49 to `Vector.elements()`, which returns an `Enumeration`. `ArrayList` does not declare this method, but its method `iterator()` returns an `Iterator`, an interface with similar functionality.[15] In this case, we can replace the call to `elements()` with a call to `iterator()`, *provided that* we replace the calls to `Enumeration.hasMoreElements()` and `Enumeration.nextElement()` on lines 50 and 51 with calls to `Iterator.hasNext()` and `Iterator.next()`, respectively.

If a `Vector` is accessed concurrently by multiple threads, then preservation of synchronization behavior is important. This is accomplished by introducing *synchronization wrappers*. This issue does not arise in the program of Figure 3 because it is single-threaded; the paper by Balaban et al. [2005] presents an example.

We have developed a Replace Class refactoring that addresses all of these migration issues.

5.1 Migration Specifications

The Replace Class refactoring relies on a *migration specification* that specifies for each method in the source class *how* it is to be rewritten. Figure 23 shows the fragments of the specification for performing the migration from `Vector` to `ArrayList` and from `Enumeration` to `Iterator` needed for the example program, plus some other rewriting rules. Balaban et al. [2005] provide a complete specification. Migration specifications only have to be written *once* for each pair of (source, target) classes.

[15]The methods `hasNext()` and `next()` in `Iterator` correspond to `hasMoreElements()` and `nextElement()` in `Enumeration`, respectively. `Iterator` declares an additional method `remove()` for the safe removal of elements from the collection being iterated over.

```
public class Util {
  public static void copyInto(ArrayList v, Object target) {
    if (target == null)
      throw new NullPointerException();
    for (int i = v.size() - 1; i >= 0; i--)
      target[i] = v.get(i);
  }
}
```

Fig. 24. Auxiliary class that contains a method used for migrating `Vector.copyInto()`.

For example, Rule (S3) states that migrating calls to method `Vector.add()` requires no modification, since `ArrayList` defines a syntactically and semantically identical method. If methods in the source class are not supported by the target class, rewriting method calls becomes more involved. For example, `Vector` supports a method `firstElement()` not defined in `ArrayList`. Rule (S5) states that a call `receiver.firstElement()`, where `receiver` is an expression of type `Vector`, should be transformed into `receiver.get(0)`. In cases where it is not possible to express the effect of a method call on the source class in terms of calls to methods on the target class, calls may be mapped to methods in user-defined classes. For example, `Vector` has a method `copyInto()` that copies the contents of a `Vector` into an array. Since `ArrayList` does not provide this functionality, Rule (S10) transforms `receiver.copyInto(array)` into a call to method `Util.copyInto(receiver, array)` of the `Util` class shown in Figure 24. This strategy can also be used to migrate between methods that throw different types of exceptions. Specifically, a user-defined class method can be used to wrap a method in a target class in order translate between exception types. Finally, when an operation in a source class that is not supported by a target class cannot be modeled using an auxiliary class, migration may become impossible.

Rules (S1) and (S2) in Figure 23 are both concerned with rewriting allocation sites of the form `new Vector()`. The former applies in cases where thread-safety need not be preserved, and transforms the allocation site into an expression `new ArrayList()`. The latter applies in situations where thread-safety must be preserved, and transforms the allocation site into `Collections.synchronizedList(new ArrayList())` using a standard synchronization wrapper in the class `java.util.Collections`. Our tools rely on an escape analysis [Hind and Pioli 2001; Ryder 2003] to determine which of the two rules should be applied, and prefers (S1) over (S2) whenever possible.

It is important to realize that, while the specification of Figure 23 describes *how* program fragments are transformed, it does not state *when* the transformation is allowed.

5.2 Further Extensions to the Type Constraints Formalism

To analyze when transformations can be applied, another adaptation of the type constraint formalism of Figure 2 is needed. However, there is a significant difference with the variants of the type constraint formalism that were used for the refactorings we presented for generalization, and for introducing generics. The refactorings for generalization in Section 3 have the effect of making types of declarations more general, and the the refactorings for introducing generics in Section 4 can be seen

$$\alpha_1 \not\leq \alpha_2$$

$$(\alpha_1 = T) \rightarrow \alpha_2 \ op \ \alpha_3$$

Fig. 25. Additional forms of type constraints required by the REPLACE CLASS refactoring.

as making the declarations of declarations more specific (by replacing occurrences of type `Object` with type parameters). More formally, the refactorings from the previous sections move type declarations between partially ordered types S and T for which either $S \leq T$ or $T \leq S$. For the REPLACE CLASS refactoring, this is no longer the case. As a result, conjunctions of $\leq$-constraints cannot be used to constrain a declaration to the set of unordered types $\{S, T\}$. Our solution involves the introduction of two artificial types $S^{\top}$ and $S_{\perp}$ in the type hierarchy for each migration from S to T. $S^{\top}$ is an immediate supertype of S and T; $S_{\perp}$ is an immediate subtype of S and T; and the new types are unrelated to all other types in the program. Then, $x \leq S^{\top} \wedge S_{\perp} \leq x$ expresses that x needs to be either S or T.[16]

Figure 25 shows two extensions to the type constraint formalism for the implementation of REPLACE CLASS. This includes constraints of the form $\alpha_1 \not\leq \alpha_2$, indicating that the type α_1 is not a subtype of the type α_2. In addition, we introduce *implication constraints* of the form $(\alpha_1 = T) \rightarrow \alpha_2 \ op \ \alpha_3$, where *op* is one of the operators '=' or '$\leq$'. Intuitively, this means that the unconditional constraint $\alpha_2 \ op \ \alpha_3$ must hold if α_1 is bound to the type T.

Figure 26 shows how the constraint generation rules of Figure 2 are adapted and extended to accommodate the REPLACE CLASS refactoring. These rules fall into three categories: (a) adaptations of the basic type constraint generation rules of Figure 2 to take the migration of classes into account, (b) rules that generate additional constraints needed to preserve program behavior in the presence of migrations, and (c) additional rules that generate implication constraints for constructor calls and method calls that may be subject to migration. Below, we will discuss a representative sample of the rules in each of these categories. In these rules, the set *MigrationTypes* contains the source classes of migrations. Furthermore, we will use the notation $C \mapsto_C C'$ to denote the fact that class C is mapped to class C' in the migration specification, and $E \mapsto_E E'$ indicates that an expression E is rewritten to an expression E' according to the migration specification.

5.2.1 *Adapting Existing Type Constraint Rules.* Figure 26(a) shows how some of the rules of Figure 2 are adapted to take class migrations into account. The key idea is that different constraints are generated for migration types than for types that will not migrate. Conceptually, the original contraints only apply to types that will not migrate. For migration types, part (c) of the Figure generates conditional constraints that account for two possibilities-either a given type is migrated or it is not. The solver determines which set of constraints to use, and uses it.

Regarding the rules shown in Figure 26(a), assignments are modeled the same way

[16]These types are only used during constraint solving. In other words, they are never introduced in the refactored source code.

$$\frac{\begin{array}{c}\text{call } E \equiv E_0.m(E_1, \cdots, E_n) \text{ to method } M \\ \mathit{Dcl}(\mathrm{M}) = \mathrm{C},\ C \notin \mathit{MigrationTypes} \\ \mathit{RootDefs}(M) = \{M_1, \cdots, M_k\},\ E'_i \equiv \mathit{Param}(M, i),\ 1 \leq i \leq n\end{array}}{\begin{array}{cr}[E] = [M] & (\mathrm{R4}) \\ [E_i] \leq [E'_i] & (\mathrm{R5}_a) \\ [E_0] \leq \mathit{Dcl}(M_1) \ \textbf{or} \ \cdots \ \textbf{or} \ [E_0] \leq \mathit{Dcl}(M_k) & (\mathrm{R6})\end{array}}$$

$$\frac{\begin{array}{c}\text{constructor call } E \equiv \texttt{new } C(\cdots) \\ C \notin \mathit{MigrationTypes}\end{array}}{[E] = C} \quad (\mathrm{R7}_a)$$

$$\frac{\begin{array}{c}\text{constructor call } E \equiv \texttt{new } C(E_1, \cdots, E_n) \\ E'_i \equiv \mathit{Param}(M, i),\ 1 \leq i \leq n,\ C \notin \mathit{MigrationTypes}\end{array}}{[E_i] \leq [E'_i]} \quad (\mathrm{R8})$$

$$\frac{\text{cast } (T)E,\ T \notin \mathit{MigrationTypes}}{[(T)E] = T} \quad (\mathrm{R14}_a) \qquad \frac{\text{cast } (T)E,\ T \in \mathit{MigrationTypes}}{\begin{array}{cr}[(T)E] \leq T^{\top} & (\mathrm{R14}_b) \\ T_{\perp} \leq [(T)E] & (\mathrm{R14}_c)\end{array}}$$

(a)

$$\frac{\begin{array}{c}\text{call } E = E_0.m(E_1, \cdots, E_n) \text{ to method } M,\ \mathit{Dcl}(M) \in \mathit{ExternalTypes} \\ [E_i]_{\mathcal{P}} = T_i,\ 1 \leq i \leq n\end{array}}{[E_i] = T_i} \quad (\mathrm{R5}_b)$$

$$\frac{\begin{array}{c}\text{downcast } (C)E,\ E' \in \mathit{PointsTo}(P, E) \\ [E']_{\mathcal{P}} \leq [(C)E]_{\mathcal{P}}\end{array}}{[E'] \leq [(C)E]} \quad (\mathrm{R32}) \qquad \frac{\begin{array}{c}\text{downcast } (C)E,\ E' \in \mathit{PointsTo}(P, E) \\ [E']_{\mathcal{P}} \not\leq [(C)E]_{\mathcal{P}}\end{array}}{[E'] \not\leq [(C)E]} \quad (\mathrm{R33})$$

$$\frac{\begin{array}{c}\text{upcast } (C)E,\ E' \in \mathit{PointsTo}(P, E) \\ [E']_{\mathcal{P}} \leq [(C)E]_{\mathcal{P}}\end{array}}{[(C)E] \leq [E']} \quad (\mathrm{R34}) \qquad \frac{\begin{array}{c}\text{upcast } (C)E,\ E' \in \mathit{PointsTo}(P, E) \\ [E']_{\mathcal{P}} \not\leq [(C)E]_{\mathcal{P}}\end{array}}{[C(E)] \not\leq [E']} \quad (\mathrm{R35})$$

(b)

$$\frac{\begin{array}{c}\text{constructor call } E \equiv \texttt{new } C(\cdots) \\ C \in \mathit{MigrationTypes}\end{array}}{\begin{array}{cr}[E] \leq C^{\top} & (\mathrm{R7}_b) \\ C_{\perp} \leq [E] & (\mathrm{R7}_c)\end{array}}$$

$$\frac{\begin{array}{c}\text{constructor call } E \equiv \texttt{new } C(E_1, \cdots, E_n),\ C \mapsto_C C',\ E \mapsto_E E' \\ c \in \mathit{cons}_{\emptyset}(E),\ c' \in \mathit{cons}_{\emptyset}(E')\end{array}}{\begin{array}{cr}([E] = C) \rightarrow c & (\mathrm{R36}) \\ ([E] = C') \rightarrow c' & (\mathrm{R37})\end{array}}$$

$$\frac{\begin{array}{c}\text{call } E.m(E_1, \cdots, E_n),\ [E]_{\mathcal{P}} = C,\ C \mapsto_C C',\ E \mapsto_E E' \\ c \in \mathit{cons}_{\emptyset}(E),\ c' \in \mathit{cons}_{\emptyset}(E')\end{array}}{\begin{array}{cr}([E] = C) \rightarrow c & (\mathrm{R38}) \\ ([E] = C') \rightarrow c' & (\mathrm{R39})\end{array}}$$

(c)

Fig. 26. Constraint generation rules for Replace Class. Part (a) of the figure shows how some of the rules of Figure 2 are adapted to apply only to non-migration types — those whose final type in the program is fixed and hence known beforehand. Part (b) shows additional rules that are needed to ensure that program behavior is preserved. Part (c) shows additional rules that generate implication constraints. Part (c) uses part (a) as a subroutine twice for each call: once showing the consequences of migrating the given type is migrated and once showing the consequences on not migrating it. In part (c), note that E' is an arbitrary expression such as those shown on the right-hand side of Figure 23.

as before so we simply reuse Rule (1) from Figure 2. For method calls, Rules (R4), ($R5_a$), and (R6) restrict the corresponding Rules (4)–(6) of Figure 2 to classes that are not in the set *MigrationTypes*.

For constructor calls $E \equiv \texttt{new}\ C(E_1, \cdots, E_n)$, Rule (7) of Figure 2 is replaced by three Rules ($R7_a$)—($R7_c$). Rule ($R7_a$) restricts the original Rule (7) of Figure 2 to classes that are not being migrated. For calls to constructors of migrated classes, Rules ($R7_b$) and ($R7_c$) constrain the type of the entire constructor call expression to be a subtype of $C^\top$ and a supertype of $C_\perp$. Rule (R8) restricts the original Rule (8) of Figure 2 to classes that are not in the set *MigrationTypes*.

Rules ($R14_a$), ($R14_b$), and ($R14_c$) adapt the original Rule (14) of Figure 2 for casts of the form $(T)E$, restricting the original rule to classes that are not the source of a migration, and permitting the migration of casts to types that are subject to migration by constraining their type to be a subtype of $T^\top$ and a supertype of $T_\perp$.

In the remainder of this section, the notation $cons_{MigrationTypes}(E)$ denotes the set of constraints generated by the rules of Figure 26(a)[17] for expression E and its subexpressions, for a given set *MigrationTypes*.

5.2.2 *Additional Rules for Preserving Program Behavior.* Figure 26(b) shows rules that generate additional constraints that are needed for preserving program behavior. Here, $[E]_{\mathcal{P}}$ denotes the type of expression E in the original program P, and *ExternalTypes* is the set of external library classes.

As we discussed earlier, opportunities for migration may be limited when an application calls a method in an external class library. This is encoded by Rule ($R5_b$), which states that the type of an expression E_i that is used as an actual parameter in a call to a method in an external library must remain the same as in the original program.

Rules (R32)–(R35) in Figure 26 are needed to preserve the run-time behavior of casts when classes are migrated. Here, the notation $PointsTo(P, E)$ refers to the set of objects (identified by their allocation sites) that an expression E in program P may point to. Any of several existing algorithms [Hind and Pioli 2001; Ryder 2003] can be used to compute points-to information. Rule (R32) ensures that for each E' in the points-to set of E for which the downcast succeeds, the downcast will still succeed in P'. Likewise, Rule (R33) enforces that for each E' in the points-to set of E for which the downcast fails, the downcast will still fail in P'. The treatment of upcasts by the Rules (R34) and (R35) is completely symmetrical to that of downcasts.

5.2.3 *Rules for Generating Implication Constraints.* Figure 26(c) shows rules that generate implication constraints for constructor calls and method calls that may be subject to migration. Here, $C \mapsto_C C'$ denotes the fact that class C is mapped to class C' in the migration specification, and $E \mapsto_E E'$ indicates that an expression E is rewritten to an expression E' according to the migration specification.

Rules (R36) and (R37) are concerned with constructor calls of the form $E \equiv \texttt{new}\ C(E_1, \cdots, E_n)$, where C is mapped to C' in the migration specification, and

[17]Figure 26(a) only shows the adapted counterparts for some of the rules of Figure 2. The remaining rules of that figure are adapted similarly.

line	constraint	rule	
12	[A1] ≤ [v1]	(1)	(i)
12	[A1] ≤ $\texttt{Vector}^{\top}$	(R7$_b$)	(ii)
12	$\texttt{Vector}_{\perp}$ ≤ [A1]	(R7$_c$)	(iii)
20	[v1] ≤ Vector	(R5$_b$)	(iv)
28	[A2] ≤ [v2]	(1)	(v)
28	[A2] ≤ $\texttt{Vector}^{\top}$	(R7$_b$)	(vi)
28	$\texttt{Vector}_{\perp}$ ≤ [A2]	(R7$_c$)	(vii)
31	[v2] = Vector → [o1] ≤ Object	(R38)	(viii)
31	[v2] = ArrayList → [o1] ≤ Object	(R39)	(ix)
43	[v2] = Vector → [v2] ≤ Collection	(R38)	(x)
43	[v2] = ArrayList → [v2] ≤ Collection	(R39)	(xi)
49	[s5.v2] = Vector → [s5.v2.elements()] = Enumeration	(R38)	(xii)
49	[s5.v2] = ArrayList → [s5.v2.elements()] = Iterator	(R39)	(xiii)

Fig. 27. Some of the type constraints generated for the application of the REPLACE CLASS refactoring to the program of Figure 3.

where the expression E is mapped to an expression E' by the migration specification. There are two cases:

(1) For the case where the migration is not allowed, we first compute the constraints c that would be computed by the rules of Figure 26(a) for the expression E and its subexpressions if C were not a migration type. For each such constraint, an implication constraint $([E] = C) \rightarrow c$ is produced.

(2) For the case where the migration is allowed, we first compute the constraints c' that would be computed by the rules of Figure 26(a) for the expression E' and its subexpressions if C were not a migration type. For each such constraint, an implication constraint $([E] = C') \rightarrow c'$ is produced.

The generation of implication constraints for method calls by Rules (R38) and (R39) is analogous.

5.2.4 *Example.* Figure 27 shows some of the type constraints generated as a result of the application of REPLACE CLASS to the program of Figure 3.

Constraint (i) establishes the required subtype relationship between the type of variable `v1` and the allocation site labeled `A1`, using Rule (1).

Constraints (ii) and (iii) in Figure 27 are generated for the allocation site labeled `A1` on line 12 in Figure 3 by Rules (R7$_b$) and (R7$_c$), and have the effect of restricting the type of `A1` to be either `Vector` or `ArrayList`.

As a more interesting example, consider the call `s5.v2.elements()` on line 49 of Figure 3, which can be rewritten to an expression `s5.v2.iterator()` according to the migration specification of Figure 23. For this method call, we have that

$$[\texttt{s5.v2.elements()}] = \texttt{Enumeration} \in \mathit{cons}_{\emptyset}(\texttt{s5.v2.elements()})$$
$$[\texttt{s5.v2.iterator()}] = \texttt{Iterator} \in \mathit{cons}_{\emptyset}(\texttt{s5.v2.iterator()})$$

and therefore the implication constraints

$$[\texttt{s5.v2}]{=}\texttt{Vector} \rightarrow [\texttt{s5.v2.elements()}]{=}\texttt{Enumeration}$$
$$[\texttt{s5.v2}]{=}\texttt{ArrayList} \rightarrow [\texttt{s5.v2.elements()}]{=}\texttt{Iterator}$$

are generated. Informally, these constraints, shown as constraints (xii) and (xiii) in Figure 27, state that the type of the call expression `s5.v2.elements()` is `Enumeration` if the type of `v2` remains `Vector`, but becomes `Iterator` if the expression is rewritten to `s5.v2.iterator()`.

```java
class Client {
  public static void main(String[] args){
    Stack s1 = new Stack();
    s1.push(new Integer(1));
    s1.push(new Integer(2));
    s1.push(new Integer(3));
    Stack s2 = new Stack();
    s2.push(new Float(4.4));
    s2.moveFrom(s1);
    s1.moveTo(s2);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Integer n = (Integer)s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    JTree tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
```

```java
class Stack {
  private ArrayList v2;
  public Stack(){
    v2 = new ArrayList();
  }
  public void push(Object o1){
    v2.add(o1);
  }
  public void moveFrom(Stack s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack s4){
    s4.push(this.pop());
  }
  public Object pop(){
    return v2.remove(v2.size() - 1);
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public boolean contains(Object o2){
    return v2.contains(o2);
  }
  public static void print(Stack s5){
    Iterator e = s5.v2.iterator();
    while (e.hasNext())
      System.out.println(e.next());
  }
}
```

Fig. 28. The example program after the application of REPLACE CLASS refactoring.

In general, solving systems of implication constraints may require backtracking, and this is supported by our implementation, which was presented in detail in [Balaban et al. 2005]. However, it is often possible to perform simplifications that eliminate the need for implications. As an example, consider the call `v2.addElement(o1)` on line 31. If the type of `v2` remains `Vector`, we must constrain `o1` to be a subtype of the formal parameter of `Vector.addElement()`, which can be expressed by constraint (viii) in Figure 27:

$$[\texttt{v2}]\text{=Vector} \rightarrow [\texttt{o1}] \leq \texttt{Object}$$

Similarly, constraint (ix) in Figure 27 models the case where the type of `v2` becomes `ArrayList`:

$$[\texttt{v2}]\text{=ArrayList} \rightarrow [\texttt{o1}] \leq \texttt{Object}$$

These constraints can be combined into a single unconditional constraint $[\texttt{o}] \leq \texttt{Object}$. Our implementation also performs other simplifications, e.g., removing an implication constraint if an equivalent unconditional constraint exists.

From constraints (i) and (iv) in Figure 27, it follows that $[\texttt{A1}] \leq [\texttt{v1}] \leq \texttt{Vector}$, implying that the type `A1` must remain `Vector`. However, the typing $[\texttt{A2}] \leftarrow \texttt{ArrayList}$, $[\texttt{v2}] \leftarrow \texttt{ArrayList}$ satisfies the constraint system, indicating that allocation site `A2` can be migrated to `ArrayList`.

Producing the refactored source code requires keeping track of the choices made for implication constraints and consulting the migration specification to determine how expressions should be rewritten. The refactored source code for the example program is shown in Figure 28.

5.2.5 *Other Pragmatic Issues.* In order to preserve synchronization behavior, the proposed technique can rely on a simple escape analysis to determine whether `Vector`s may escape their thread. `Vector`s that do not escape are migrated to `ArrayList`s (if no constraints are violated). For escaping objects such as escaping `Vector`s, an alternative migration can be based on the introduction of synchronization wrappers as indicated by Rule (S2) of Figure 23. Hence, there are three alternatives for each `Vector` allocation site: it can remain a `Vector`, become an unwrapped `ArrayList`, or a wrapped `ArrayList`. This, and other pragmatic issues such as the handling of $\not\leq$-constraints, are discussed at greater length by Balaban et al. [2005].

6. EXPERIMENTAL RESULTS

We previously presented detailed evaluations of the effectiveness of the Infer Generic Type Arguments [Fuhrer et al. 2005], Introduce Type Parameter [Kieżun et al. 2007], and Replace Class [Balaban et al. 2005] refactorings. This section presents a summary of the experiments conducted and results obtained. For further detail, the reader is referred to our previous papers.

6.1 Evaluation of Introduce Generic Type Arguments

One of the main benefits of Introduce Generic Type Arguments is that it removes downcasts that have become unnecessary. To evaluate the effectiveness of the refactoring, we used Introduce Generic Type Arguments to infer actual type parameters for declarations and allocation sites that refer to the unparameterized standard collections in a suite of moderate-sized Java programs[18]. We then measured the percentage of downcasts that could be removed, and the percentage of "unchecked warnings" that were eliminated.

Table I shows that an average of 49% of all casts could be removed from each benchmark, and an average of 91% of all unchecked warnings were eliminated. When considering casts, the reader should note that the number of casts given in column (iv) of Table I includes casts that are not related to the use of generic types. However, a manual inspection of the results revealed that our tool removes the vast majority of generics-related casts, from roughly 75% to 100%. For example, we estimate that only one-fifth of *ANTLR*'s total number of casts relates to the use of collections, which is close to our tool's 19% removal rate.

6.2 Evaluation of Introduce Type Parameter

In order to evaluate the Introduce Type Parameter refactoring, we inferred type parameters in a set of non-parameterized libraries. Our evaluation uses a combination of 6 libraries that have already been parameterized by their authors, and 2 libraries that have not yet been made generic. For already-parameterized libraries, we first applied a tool that erased the formal and actual type parameters and added necessary type casts, and then compared the results produced by our

[18]For more details, see: www.junit.org, www.cs.princeton.edu/~appel/modern/java/JLex/, www.cs.princeton.edu/~appel/modern/java/CUP/, www.spec.org/osg/jvm98/, vpoker.sourceforge.net, telnetd.sourceforge.net, www.antlr.org, jbidwatcher.sourceforge.net, pmd.sourceforge.net, htmlparser.sourceforge.net, and www.ovmj.org/xtc/.

benchmark	size			generics-related metrics				removed	
	types	KLOC	casts	allocs	dcls	subt.	warn.	casts	warn.
JUnit	59	5.3	54	24	48	0	27	44%	93%
V_poker	35	6.4	40	12	27	1	47	80%	100%
JLex	22	7.8	71	17	33	1	40	68%	85%
Db	32	8.6	78	14	36	1	652	51%	100%
JavaCup	36	11.1	595	19	62	0	55	82%	96%
TelnetD	52	11.2	46	16	28	0	22	83%	100%
Jess	184	18.2	156	47	64	1	692	53%	99%
JBidWatcher	264	38.6	383	76	184	1	195	54%	97%
ANTLR	207	47.7	443	46	106	3	84	19%	94%
PMD	395	38.2	774	75	286	1	183	20%	89%
HTMLParser	232	50.8	793	72	136	2	205	22%	97%
Jax	272	53.9	821	119	261	3	583	19%	48%
xtc	1,556	90.6	1,114	330	668	1	583	36%	88%
							average:	48.6%	91.2%

Table I. Experimental results for Infer Generic Type Arguments. Column (i) is the name of the benchmark. The size of the application is measured in (ii) number of types, (iii) thousands of source lines, and (iv) number of casts. The generics-related metrics count the number of (v) allocation sites of generic types, (vi) generic-typed declarations, (vii) subtypes of generic types, and (viii) "unchecked warnings" issued by the compiler. Lastly, columns (ix) and (x) show the percentage of casts removed and unchecked warnings eliminated as a result of applying Infer Generic Type Arguments.

	parameterizable classes			comparison to manual		
library	classes	LOC	type uses	same	better	worse
concurrent	14	2715	415	353	37	25
apache	74	9904	1183	1011	116	56
jutil	9	305	80	65	15	0
jpaul	17	827	178	148	22	8
amadeus	8	604	129	125	1	3
dsa	9	791	162	158	4	0
antlr	10	601	140	n/a	n/a	n/a
eclipse	7	582	100	n/a	n/a	n/a
Total	148	16329	2387	1860	195	92

Table II. Experimental results for Introduce Type Parameter. From left to right, the columns of the table show: (i) the name of the library, (ii) the number of analyzed classes, including their nested classes, (iii) the number of lines of code, and (iv) the number of occurrences of a reference (non-primitive) type in the library; the latter is the maximal number of locations where a type parameter could be used instead. The "comparison to manual" columns (v)-(vii) indicate how our tool's output compares to the manual parameterization; the numbers count type uses, as in column (iv). These columns are empty for the last two libraries, which have not been manually parameterized by their authors.

implementation of Introduce Type Parameter against the existing parameterization. In the case of the libraries for which no generic version was available, we asked the developers to examine every change proposed by our implementation and to give their opinion of the result.

The leftmost column of Table II lists the class libraries used in this experiment.[19] Not all classes in these libraries are amenable to parameterization; we selected a subset of the library classes that we considered likely to be parameterizable. The experiments processed the classes of the library in the following order: we first built a dependence graph of the classes, and then applied our tool to each strongly connected component, starting with those classes that depended on no other classes still to be parameterized. This is the same order a programmer faced with the problem would choose.

Given an existing manual parameterization and one computed by our refactoring tool, we used two criteria to decide which was more precise. The first, and more important, is which one allows more casts to be removed—in clients or in the library itself. The secondary criterion is which one more closely follows the style used in the JDK collections; they were developed and refined over many years by a large group of experts and can be reasonably considered models of style[20]. The two criteria are in close agreement.

Our results for the already-parameterized libraries can be summarized as follows. For 87% of all type annotations, the output of our tool is identical or equally good as the existing parameterization. For 4% of annotations, the output of our tool is worse than that created by the human. For 9% of annotations, the tool output is better than that created by the human. For the *eclipse* and *antlr* libraries, no existing parameterization was available. A developer of Eclipse concluded that the changes were "good and useful for code migration to Java 5.0." Out of 100 uses of types in the Eclipse classes we parameterized, he mentioned only 1 instance where the inferred result, while correct, could be improved. A developer of ANTLR stated that the changes made by our tool are "absolutely correct". Out of 140 uses of types in the parameterized classes, he mentioned 1 instance where the inferred result, while correct, could be improved.

From the above results, it is clear that the parameterizations computed by our tool resemble the manually computed solutions very closely. A few examples where the output of our tool was worse are the following.

(1) In *concurrent*, our tool does not instantiate the field `next` in member type `LinkedBlockingQueue.Node` as `Node<E>`, but leaves it raw. Such a choice is safe, but it is less desirable than the manual parameterization.
(2) Our tool does not infer type parameters for methods; an example is *apache*'s `PredicatedCollection.decorate`.
(3) Our tool inferred two separate type parameters for interface `Buffer` in the *apache* library. In this case the manual parameterization had only one.

Moreover, a few examples where the output of our tool was better are the following.

[19] Here, *concurrent* is the `java.util.concurrent` package from Sun JDK 1.5, *apache* is the Apache collections library (`larvalabs.com/collections/`), *jutil* is a Java Utility Library (`cscott.net/Projects/JUtil/`), *jpaul* is the Java Program Analysis Utility Library (`jpaul.sourceforge.net`), *amadeus* is a data structure library (`people.csail.mit.edu/adonovan/`), *dsa* is a collection of generic data structures(`www.cs.fiu.edu/~weiss/#dsaajava2`), *antlr* is a parser generator (`www.antlr.org`), and *eclipse* is a universal tooling platform (`www.eclipse.org`).

[20] When multiple styles appear in the JDK, we did not count differences in either the "better" or "worse" category.

benchmark	types	KLOC	migration classes	declarations (migr./ unchanged)		alloc. sites (migr. desync/ migr. wrap/ unchanged)			call sites (migr./ unchanged)	
Hanoi	41	4.0	*V*	3	0	3	0	0	26	0
JUnit	100	5.3	*V*, *HT*, *E*	55	7	23	1	0	111	7
JLex	26	7.9	*V*, *HT*, *E*	29	10	12	0	4	167	18
JavaCup	36	10.6	*HT*, *E*	56	0	14	0	0	153	0
Cassowary	68	12.2	*V*, *HT*, *E*	121	18	44	0	2	692	36
Azureus	160	13.9	*V*	13	0	6	6	0	51	0
HTML Parser	115	17.1	*V*, *HT*	141	3	21	0	2	461	6
JBidWatcher	154	22.9	*V*, *HT*	67	4	32	1	3	291	3
SpecJBB	110	31.3	*V*, *HT*, *E*	22	6	13	0	2	78	10
Jax	309	53.1	*V*, *HT*, *E*	208	43	81	0	12	706	0

Table III. Experimental results for Replace Class. From left to right, the columns of the table show: (i) the name of the benchmark, (ii) the number of classes in the benchmark, (iii) the number of lines of source code (in thousands), (iv) the migration source classes used in this benchmark, where V denotes `Vector`, HT denotes `Hashtable`, and E denotes `Enumeration`. The last columns of the table show the experimental results: (v) the number of source declarations of legacy types that could be migrated and that had to be left unchanged, (vi) the number of allocation sites of legacy types that could be migrated and without synchronization wrappers, that could be migrated but needed synchronization wrappers, and that could not be migrated, and (vii) the number of legacy call sites that could be migrated and that could not be migrated.

In each case, the developers of the package agreed the inferred solution was better than their manual parameterization.

(1) Our tool adds a formal type parameter to member class `SynchronousQueue.Node` in *concurrent*. The parameter allows elimination of several casts inside `SynchronousQueue`.
(2) In method `VerboseWorkSet.containsAll` in *jpaul*, our tool inferred an upper-bounded type parameter wildcard for the `Collection` parameter. This permits more flexible use and fewer casts by clients, and also adheres to the standard collections style from the JDK.
(3) Our tool inferred `Object` as the type of the parameter of method `Canonical.getIndex` in *amadeus*. This is more flexible with fewer casts and follows the JDK style. A similar case occurred in *jpaul* in which our tool inferred `Object` for the parameter of `WorkSet.contains`.

6.3 Evaluation of Replace Class

We evaluated our implementation of Replace Class on a number of Java applications of up to 53 KLOC that we migrated from `Vector` to `ArrayList`, from `Hashtable` to `HashMap`, and from `Enumeration` to `Iterator`. Table III states the essential characteristics for each benchmark program.

The results are shown in last three columns of the table. For example, for the *Cassowary* benchmark we found that:

(1) 121 of the original 139 source class declarations were migrated, but 18 could not be migrated,
(2) 44 of the 46 source allocation sites could be migrated without inserting synchronization wrappers, and the remaining 2 source allocation sites could not be migrated at all, and

(3) 692 of the 728 source call sites could be migrated, and the remaining 36 call sites could not be migrated.

On average, 90% of source declarations and 97% of source call sites were migrated successfully. Furthermore, an average of 92% of all allocation sites can be migrated: 83% without the insertion of synchronization wrappers and 9% with the insertion of synchronization wrappers.

We now discuss a few cases out of the experiments illustrating a number of nontrivial aspects of migration.

Nontrivial rewriting. *JBidWatcher*, *JUnit*, and *SpecJBB* contain calls to the previously discussed method `Vector.copyInto()` which requires nontrivial rewriting and introduction of an auxiliary class. In *JBidWatcher*, *JUnit*, *SpecJBB*, and *Jax*, the percentages of migrated call sites for which the method's name or signature was changed are 30%, 75%, 73%, and 47%, respectively. Clearly, manual migration of these applications would involve a significant amount of error-prone editing work.

Interaction with external libraries. In *JUnit*, one of the `Vector`s is passed to the constructor of the external *Swing* library class `JList`, whose formal parameter is of type `Vector`. This flow of objects is not immediately evident from the code, as the allocated `Vector` is assigned to a variable that is elsewhere passed to the constructor. Similar cases occur in *SpecJBB* where various `Vector`s are not migrated because they are stored in other `Vector`s. With more insight into the implementation of `Vector`, it is evident that concrete types of its elements are irrelevant, which could be utilized by a more precise analysis.[21]

Synchronization preservation. The migration of *JUnit* includes a synchronization-wrapped allocation site. It is detected as escaping since it is assigned to a field whose declaring class declares a `Runnable` that references the field. The `Runnable` object is passed to *Swing*, which would cause any escape analysis without access to the *Swing* code to declare it as escaping. In *Azureus*, escape analysis reports that synchronization wrappers need to be introduced for certain `ArrayList`s, but the program already performs explicit synchronizations. In principle, a more precise escape analysis could enable migration without synchronization in this case.

7. RELATED WORK

Opdyke [1992, page 27–28] identified some of the invariants that refactorings must preserve. One of these, *Compatible Signatures in Member Function Redefinition*, states that overriding methods must have corresponding argument types and return types, corresponding to our Rules (9) and (10). Opdyke writes the following about the *Type-Safe Assignments* invariant: "The type of each expression assigned to a variable must be an instance of the variable's defined type, or an instance of one of its subtypes. This applies both to assignment statements and function calls." This corresponds to our Rules (1), (5), and (8).

Fowler [1999] presents a comprehensive classification of a large number of refactorings, which includes step-by-step directions on how to perform each of these

[21] Such information could be provided in the form of *stub* implementations that approximate the behavior of selected library methods.

manually. Many of the thorny issues are not addressed. E.g., in the case of Extract Interface, Fowler only instructs one to "Adjust client type declarations to use the interface", ignoring the fact that not all declarations can be updated.

Tokuda and Batory [2001] discuss refactorings for manipulating design patterns including one called Substitute which "generalizes a relationship by replacing a subclass reference to that of its superclass". Tokuda and Batory point out that "This refactoring must be highly constrained because it does not always work". Our model can be used to add the proper precondition checking.

Halloran and Scherlis [2002] present an informal algorithm for detecting over-specific variable declarations. This algorithm is similar in spirit to our Generalize Declared Type refactoring by taking into account the members accessed from a variable, as well as the variables to which it is assigned.

The Infer Type refactoring by Steimann et al. [2006] lets a programmer select a given variable and determines or creates a minimal interface that can be used as the type for that variable. Steimann et al. only present their type inference algorithm informally, but their constraints appear similar to those presented in Section 2. In more recent work, Steimann and Mayer [2007] observe that the repeated use of Infer Type may produce suboptimal results (e.g., the creation of many similar types). Their Type Access Analyzer performs a global analysis to create a lattice that can be used as the basis for extracting supertypes, changing the types of declarations, merging structurally identical supertypes, etc.

The KABA tool [Streckenbach and Snelting 2004; Snelting and Tip 2000] generates refactoring proposals for Java applications (e.g., indications that a class can be split, or that a member can be moved). In this work, type constraints record relationships between variables and members that must be preserved. From these type constraints, a binary relation between classes and members is constructed that encodes precisely the members that must be visible in each object. Concept analysis is used to generated a concept lattice from this relation, from which refactoring proposals are generated.

Duggan's approach for parameterizing classes [Duggan 1999] predates Java generics, and his PolyJava language is incompatible with Java in several respects (e.g., the treatment of raw types and arrays, no support for wildcards). Unlike our approach, Duggan's takes a class as its input and relies on usage information to generate constraints that relate the types of otherwise unrelated declarations. If usage information is incomplete or unavailable, too many type parameters may be inferred. To our knowledge, Duggan's work was never implemented.

Donovan and Ernst [2003] present solutions to both the parameterization and the instantiation problems. For parameterization, a dataflow analysis is applied to each class to infer as many type parameters as are needed to ensure type correctness. Then, type constraints are generated to infer how to instantiate occurrences of parameterized classes. Donovan and Ernst report that "often the class is over-generalized", i.e., too many type parameters are inferred. Donovan and Ernst's work infers arrays of parameterized types, which are not allowed in Java but were permitted by the then-current proposal. Their work was only partially implemented before they turned to the work below.

Donovan et al. [2004] present a solution to the instantiation problem based on a context-sensitive pointer analysis. Their approach uses "guarded" constraints that are conditional on the rawness of a particular declaration, and that require a (limited) form of backtracking, similar to the implication constraints used in Section 5. Our solution is more scalable than Donovan's because it requires neither context-sensitive analysis nor backtracking, and more general because it is capable of inferring precise generic supertypes for subtypes of generic classes. Moreover, as Donovan's work predates Java 1.5, their refactoring tool does not consider wildcard types and supports arrays of generic types (now disallowed).

von Dincklage and Diwan [2004] present a solution to both the parameterization problem and the instantiation problem based on type constraints. Their Ilwith tool initially creates one type parameter per declaration, and then uses heuristics to merge type parameters. While the successful parameterization of several classes from the Java standard collections is reported, some of the inferred method signatures differ from those in the Java 1.5 libraries. It also appears that program behavior may be changed because constraints for overriding relationships between methods are missing. As a practical matter, Ilwith does not actually rewrite source code, but merely prints method signatures without providing details on how method *bodies* should be transformed.

8. CONCLUSIONS

An important category of refactorings is concerned with manipulating types and class hierarchies. For these refactorings, type constraints are an excellent basis for checking preconditions and computing source code modifications. We have discussed refactorings for generalization, for the introduction of generics, and for performing migrations between similar classes, using slight variations on a common type constraint formalism. All of our refactorings have been implemented in Eclipse, and several refactorings in the standard Eclipse distribution are based on our research. A detailed evaluation of the performance and effectiveness of our refactorings can be found in our earlier papers [Fuhrer et al. 2005; Balaban et al. 2005; Kieżun et al. 2007].

Acknowledgments

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004 and by the Fund for Scientific Research – Flanders (Belgium).

Dirk Bäumer, Julian Dolby, and Markus Keller made significant contributions to the refactorings described in this paper. Jan Vitek provided valuable comments on a draft of this paper.

REFERENCES

Balaban, I., Tip, F., and Fuhrer, R. 2005. Refactoring support for class library migration. In *Proc. OOPSLA*. 265–279.

Beck, K. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Bracha, G., Cohen, N., Kemper, C., Odersky, M., Stoutamire, D., Thorup, K., and Wadler, P. 2004. Adding generics to the Java programming language, final release. Tech. rep., Java Community Process JSR-000014. Sept.

De Sutter, B., Tip, F., and Dolby, J. 2004. Customization of Java library classes using type constraints and profile information. In *Proc. ECOOP'04*. 585–610.

Donovan, A. and Ernst, M. 2003. Inference of generic types in Java. Tech. Rep. MIT/LCS/TR-889, MIT. Mar.

Donovan, A., Kieżun, A., Tschantz, M., and Ernst, M. 2004. Converting Java programs to use generic libraries. In *Proc. OOPSLA*. 15–34.

Duggan, D. 1999. Modular type-based reverse engineering of parameterized types in Java code. In *Proc. OOPSLA*. 97–113.

Fowler, M. 1999. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley.

Fuhrer, R., Tip, F., Kieżun, A., Dolby, J., and Keller, M. 2005. Efficiently refactoring Java applications to use generic libraries. In *Proc. ECOOP'05*. 71–96.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The Java Language Specification*, Third ed. Addison Wesley, Boston, MA.

Griswold, W. G. 1991. Program restructuring as an aid to software maintenance. Ph.D. thesis, University of Washington. Technical Report 91-08-04.

Griswold, W. G. and Notkin, D. 1993. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol. 2,* 3, 228–269.

Halloran, T. J. and Scherlis, W. L. 2002. Models of Thumb: Assuring best practice source code in large Java software systems. Tech. Rep. Fluid Project, School of Computer Science/ISRI, Carnegie Mellon University. Sept.

Hind, M. and Pioli, A. 2001. Evaluating the effectiveness of pointer alias analyses. *Science of Comp. Programming 39,* 1, 31–55.

Kerievsky, J. 2004. *Refactoring to Patterns*. Addison-Wesley.

Kieżun, A., Ernst, M., Tip, F., and Fuhrer, R. 2007. Refactoring for parameterizing Java classes. In *Proc. ICSE*. 437–446.

Kieżun, A., Ernst, M. D., Tip, F., and Fuhrer, R. M. 2006. Refactoring for parameterizing Java classes. Tech. rep., MIT. Sept.

Mens, T. and Tourwé, T. 2004. A survey of software refactoring. *IEEE Trans. on Softw. Eng. 30,* 2 (Feb.), 126–139.

Opdyke, W. F. 1992. Refactoring object-oriented frameworks. Ph.D. thesis, University Of Illinois at Urbana-Champaign.

Opdyke, W. F. and Johnson, R. E. 1993. Creating abstract superclasses by refactoring. In *The ACM 1993 Computer Science Conf. (CSC'93)*. 66–73.

Palsberg, J. and Schwartzbach, M. 1993. *Object-Oriented Type Systems*. John Wiley & Sons.

Ryder, B. G. 2003. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. CC 2003*. Warsaw, Poland, 126–137.

Snelting, G. and Tip, F. 2000. Understanding class hierarchies using concept analysis. *ACM Trans. on Programming Languages and Systems 22,* 3 (May), 540–582.

Steimann, F. and Mayer, P. 2007. Type access analysis: Towards informed interface design. In *Proc. TOOLS Europe*. To appear.

Steimann, F., Mayer, P., and Meissner, A. 2006. Decoupling classes with inferred interfaces. In *ACM Symposium on Applied Computing (SAC)*. Dijon, France, 1404–1408.

Streckenbach, M. and Snelting, G. 2004. Refactoring class hierarchies with KABA. In *Proc. OOPSLA*. Vancouver, BC, Canada, 315–330.

Tip, F. 2007. Refactoring using type constraints. In *Proceedings of the 14th International Static Analysis Symposium (SAS'07)*. Kongens Lyngby, Denmark, 1–17.

Tip, F., Kieżun, A., and Bäumer, D. 2003. Refactoring for generalization using type constraints. In *Proc. OOPSLA*. 13–26.

Tokuda, L. and Batory, D. 2001. Evolving object-oriented designs with refactorings. *Kluwer Journal of Automated Software Engineering 8,* 1 (Jan.), 89–120.

Torgersen, M., Plesner Hansen, C., Ernst, E., von der Ahé, P., Bracha, G., and Gafter, N. 2004. Adding wildcards to the Java programming language. In *Proc. of the 2004 ACM symposium on Applied computing*. 1289–1296.

von Dincklage, D. and Diwan, A. 2004. Converting Java classes to use generics. In *Proc. OOPSLA*. 1–14.