Refactoring Using Type Constraints*

Frank Tip

IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
ftip@us.ibm.com

Abstract. Type constraints express subtype-relationships between the types of program expressions that are required for type-correctness, and were originally proposed as a convenient framework for solving type checking and type inference problems. In this paper, we show 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 results.

1 Introduction

Refactoring is the process of applying behavior-preserving transformations (called "refactorings") to a program's source code with the objective of improving that program's design. Common reasons for refactoring include the elimination of undesirable program characteristics such as duplicated code, making existing program components reusable in new contexts, and breaking up monolithic systems into components. Pioneered in the early 1990s by Opdyke et al. [15,16] and by Griswold et al. [9,10], the field of refactoring received a major boost with the emergence of code-centric design methodologies such as extreme programming [2] that advocate continuous improvement of code quality. Fowler [7] and Kerievsky [12] authored popular books that classify many widely used refactorings, and Mens and Tourwé [14] presented a survey of the field.

Refactoring is usually presented as an interactive process where the programmer takes the initiative by indicating a point in the program where a specific transformation should be applied. Then, the programmer must verify if a number of specified preconditions hold, and, assuming this is the case, apply a number of prescribed editing steps. However, 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`).

* This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 1–17, 2007.

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 as a system of type constraints. Type constraints [17] 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, and were originally proposed as a means for expressing type checking and type inference problems. In our work, a system of type constraints is derived from a program to reason about the correctness of refactorings. Specifically, 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.

We show 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. Several refactorings[1] in the Eclipse 3.2 distribution are based on the research presented in this paper. Our previous papers [22,3,8,1,13], presented these refactorings in detail, along with experimental evaluations. This paper presents an informal overview of the work and uses a running example to show how different refactorings require slight variations on the basic type constraints model.

2 Type Constraints

Type constraints are a formalism for expressing subtype relationships between the types of declarations and expressions, and were originally proposed as a means for stating type-checking and type inference problems [17]. In the basic model, a *type constraint* has of one of the following forms:

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

Here, α, α', ... are *constraint variables* that represent the types associated with program constructs. In this paper, M denotes a method (with associated signature and type information), F denotes a field, C denotes a class, I denotes an interface, T denotes a class or an interface, and E denotes an expression. Constraint variables are of one of the following forms:

T	a type constant	$[F]$	the declared type of field F
$[E]$	the type of an expression E	$Decl(M)$	the type in which method M is declared
$[M]$	the declared return type of method M	$Decl(F)$	the type in which field F is declared

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

program construct	implied type constraint(s)	
assignment $E_1 = E_2$	$[E_2] \leq [E_1]$	(1)
method call $E.m(E_1, \cdots, E_n)$	$[E.m(E_1, \cdots, E_n)] = [M]$	(2)
to a virtual method M	$[E_i] \leq [Param(M, i)]$	(3)
where $RootDefs(M) = \{ M_1, \cdots, M_k \}$	$[E] \leq Decl(M_1)$ **or** $\cdots$ **or** $[E] \leq Decl(M_k)$	(4)
access $E.f$ to field F	$[E.f] = [F]$	(5)
	$[E] \leq Decl(F)$	(6)
`return` E in method M	$[E] \leq [M]$	(7)
M' overrides M,	$[Param(M', i)] = [Param(M, i)]$	(8)
$M' \neq M$	$[M'] \leq [M]$	(9)
F' hides F	$Decl(F') < Decl(F)$	(10)
constructor call `new` $C(E_1, \cdots, E_n)$	$[\texttt{new } C(E_1, \cdots, E_n)] = C$	(11)
to constructor M	$[E_i] \leq [Param(M, i)]$	(12)
direct call	$[E.m(E_1, \cdots, E_n)] = [M]$	(13)
$E.m(E_1, \cdots, E_n)$	$[E_i] \leq [Param(M, i)]$	(14)
to method M	$[E] \leq Decl(M)$	(15)
implicit declaration of `this` in method M	$[\texttt{this}] = Decl(M)$	(16)

Fig. 1. Type constraints for a set of core Java language features

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 1 shows rules that generate constraints from a representative set of program constructs.

For example, 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 otherwise the assignment would not be type correct. In the rules discussed below, $Param(M, i)$ denotes the i-th formal parameter of method M. For a call $E.m(\cdots)$ to a virtual method M, we have that: the type of the call-expression is the same as M's return type (rule (2)[2]), the type of each actual parameter must be the same as, or a subtype of the corresponding formal parameter (rule (3)), and a method with the same signature as M must be declared in $[E]$ or one of its supertypes (rule (4)). Rule (4) determines a set of methods $M_1, \cdots, M_k$ overridden by M using Definition 1 below, and requires $[E]$ to be a subtype of one or more[3] of $Decl(M_1), \cdots, Decl(M_k)$. In this definition, a virtual method M in type C overrides a virtual 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 1 (RootDefs). *Let M be a method. Define:*
$RootDefs(M) = \{ M' |\, M$ *overrides* M'*, and there exists no* M'' $(M'' \neq M')$ *such that* M' *overrides* $M'' \}$

[2] Rules (2), (5), (13), (11), and (16) *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.

[3] In cases where a referenced method does not occur in a supertype of $[E]$, the *RootDefs*-set defined in Definition 1 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.

Changing a parameter's type need not affect type-correctness, but may affect virtual dispatch (and program) behavior. Hence, we require that types of corresponding parameters of overriding methods be identical (rule (8)). As of Java 5.0, return types in overriding methods may be covariant (rule (9)). Rule (16) defines the type of a `this` expression to be the class that declares the associated method. The constraint rules for several features (e.g., casts) have been omitted due to space limitations and can be found in our earlier papers.

3 Refactorings for Generalization

Figure 2 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()`, and `isEmpty()` 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 integer `1` onto it, then creates another stack onto which it pushes the values `2.2` and `3.3`. The elements of the second stack are then moved to the first, the contents of one of the stacks is printed, and the elements of the first stack are transferred into a `Vector` whose contents are displayed in a tree. Executing the program creates a graphical representation of a tree containing, from top to bottom, nodes 2.2, 3.3, and 1.

3.1 Extract Interface

One possible criticism about the code in Figure 2 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 aims to address this issue by introducing an interface that declares a subset of the methods in a class, and updating references in client code to refer to the interface instead of the class wherever possible. 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, as shown in Figure 3 (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 45.

Using type constraints, it is straightforward to compute the declarations that can be updated to refer to `IStack` instead of `Stack`. Figure 4(a) shows some of the type constraints generated for declarations and expressions of type `Stack` in the program of Figure 2, according to the the rules of Figure 1. It is important to note that the constraints were generated *after* adding interface `IStack` to the class hierarchy. Now, from the constraints of Figure 4(a), it is easy to see that `Stack`$\leq$[`s2`]$\leq$[`s5`]$\leq$`Stack` and hence that the types of `s2` and `s5` have to remain

```
[1]  class Client {
[2]    public static void main(String[] args){
[3]      Stack s1 = new Stack();
[4]      s1.push(new Integer(1));
[5]      Stack s2 = new Stack();
[6]      s2.push(new Float(2.2));
[7]      s2.push(new Float(3.3));
[8]      s1.moveFrom(s2);
[9]      s2.moveTo(s1);
[10]     Stack.print(s2);
[11]     Vector v1 = new Vector(); /* A1 */
[12]     while (!s1.isEmpty()){
[13]       Number n = (Number)s1.pop();
[14]       v1.add(n);
[15]     }
[16]     JFrame frame = new JFrame();
[17]     frame.setTitle("Example");
[18]     frame.setSize(300, 100);
[19]     JTree tree = new JTree(v1);
[20]     frame.add(tree, BorderLayout.CENTER);
[21]     frame.setVisible(true);
[22]   }
[23] }
```

```
[24] class Stack {
[25]   private Vector v2;
[26]   public Stack(){
[27]     v2 = new Vector(); /* A2 */
[28]   }
[29]   public void push(Object o){
[30]     v2.addElement(o);
[31]   }
[32]   public Object pop(){
[33]     return v2.remove(v2.size()-1);
[34]   }
[35]   public void moveFrom(Stack s3){
[36]     this.push(s3.pop());
[37]   }
[38]   public void moveTo(Stack s4){
[39]     s4.push(this.pop());
[40]   }
[41]   public boolean isEmpty(){
[42]     return v2.isEmpty();
[43]   }
[44]   public static void print(Stack s5){
[45]     Enumeration e = s5.v2.elements();
[46]     while (e.hasMoreElements())
[47]       System.out.println(e.nextElement());
[48]   }
[49] }
```

Fig. 2. An example program. The allocation sites for the two `Vector` objects 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));
    Stack s2 = new Stack();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Number n = (Number)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 o){
    v2.addElement(o);
  }
  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 static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    while (e.hasMoreElements())
      System.out.println(e.nextElement());
  }
}
```

Fig. 3. The example program of Figure 2 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)

line(s)	constraint(s)	rule(s)
3	Stack≤[s1]	(11),(1)
4, 8, 12, 13	[s1]≤IStack	(4)
5	Stack≤[s2]	(11),(1)
6, 7, 9	[s2]≤IStack	(4)
8,35	[s2]≤[s3]	(3)
9,38	[s1]≤[s4]	(3)
10,44	[s2]≤[s5]	(14)
36	[s3]≤IStack	(4)
39	[s4]≤IStack	(4)
45	[s5]≤Stack	(6)

(a)

line(s)	constraint(s)	rule applied
19	JTree≤[tree]	(11),(1)
20	[tree]≤Component	(12)
11	Vector≤[v1]	(11),(1)
14	[v1]≤Collection	(4)
19	[v1]≤Vector	(12)
27	Vector≤[v2]	(11),(1)
30	[v2]≤Vector	(4)
33, 42	[v2]≤Collection	(4)

(b)

Fig. 4. (a) Type constraints generated for the application of the Extract Interface refactoring to the program of Figure 2 (only nontrivial constraints related to variables `s1`–`s5` are shown). (b) Type constraints used for the application of Generalize Declared Type (only nontrivial constraints related to variables `tree`, `v1`, and `v2` are shown). Line numbers refer to Figure 2, and rule numbers to rules of Figure 1.

`Stack`. However, the types of `s1` and `s4` are less constrained ([`s1`]≤[`s4`]≤`IStack`) implying that type `IStack` may be used for these variables. In general, the types of variables may not be changed independently. For example, changing `s1`'s type to `IStack` but leaving `s4`'s type unchanged results in a type-incorrect program. In a previous paper [22], we presented an algorithm for computing the maximal set of variables whose type can be updated to refer to a newly extracted interface.

3.2 Generalize Declared Type

Another possible criticism of the program of Figure 2 is the fact that the types of some variable declarations in the program of Figure 2 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 `Component` instead of `JTree` without affecting type-correctness or program behavior, as is indicated by a box in Figure 3. This, in turn, would enable one to vary the implementation to use, say, a `JList` instead of a `JTree` in `Client.main()`. 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 the method `addElement()`, which is not declared in any supertype of `Vector`, is invoked on `v2` on line 30. Furthermore, the type of `v1` cannot be generalized because, on line 19, `v1` is passed 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 4(b) shows the constraints generated from the example program of Figure 2 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: `JTree`≤[`tree`]≤`Component`, `Vector`≤[`v1`]≤`Vector`, and `Vector`≤[`v2`]≤`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 `Component`.

3.3 Other Refactorings for Generalization

Several other refactorings related to generalization can be modeled similarly. For example, the Pull Up Members refactoring is concerned with moving methods and fields from a class to one of its superclasses. For this refactoring, we leave the types of variables constant by including constraints that require variables to have their originally declared type while allowing the locations of methods and fields to vary by leaving constraint variables of the form *Decl*(.) unconstrained.

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, the signatures of access methods such as `Vector.get()` refer to type `Object`, which prevents the compiler from ensuring the type-safety of vector operations, and therefore down-casts 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.

Donovan et al. [5] 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. 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. The former problem subsumes the latter because the introduction of type parameters often requires the instantiation of generic classes.

The Introduce Type Parameter refactoring developed recently by Kieżun et al. [13] provides a solution to the parameterization problem in which the programmer selects a declaration for which the 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 [24]). Fuhrer et al. [8] proposed a solution to the instantiation problem that forms the basis for the Infer Generic Type Arguments refactoring in Eclipse.

The right column of Figure 5 shows class `Stack` after applying Introduce Type Parameter to the formal parameter of method `Stack.push()` (for the purposes of this example, it is assumed that class `Stack` is analyzed in isolation). Underlining is used to indicate changes w.r.t. the version of `Stack` in Figure 2.

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 the parameter of `Stack.push()`, for the return type of `Stack.pop()`, and for the type of field `v2`. A more interesting change can be seen in the `moveFrom()`, `moveTo()`, and `print()` methods. Here, the parameters now have wildcard types `Stack<? extends T1>`, `Stack<? super T1>`, and `Stack<?>`, respectively. As we shall see shortly, 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 their declared types.

The left column of Figure 5 shows the result of applying Infer Generic Type Arguments to the example program after the parameterization of `Stack`. Observe that the types of `s1` and `s2` are now `Stack<Number>` and `Stack<Float>`, and that the downcast on line 13 that was present originally has been removed. This result was enabled directly 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 `s1` and `s2`, making it impossible to remove the downcast.

```
class Client {
  public static void main(String[] args){
    Stack<Number> s1 = new Stack<Number>();
    s1.push(new Integer(1));
    Stack<Float> s2 = new Stack<Float>();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector<Number> v1 = new Vector<Number>();
    while (!s1.isEmpty()){
      Number n = 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 o){
    v2.addElement(o);
  }
  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 static void print(Stack<?> s5){
    Enumeration<?> e = s5.v2.elements();
    while (e.hasMoreElements())
      System.out.println(e.nextElement());
  }
}
```

Fig. 5. The example program after the application of Introduce Type Parameter to the formal parameter of `Stack.push()`, followed by an application of Infer Generic Type Arguments to the entire application

4.1 Infer Generic Type Arguments

The Infer Generic Type Arguments refactoring requires a minor extension of the type constraint formalism of Section 2, which we illustrate by way of our running example. Some technical details are not discussed due to space limitations, and can be found in a previous paper [8].

In order to reason about type parameters, we introduce a new kind of constraint variable. These constraint variables are of the form $T(x)$, representing 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 `E`(v) = `String`. We also need additional rules for generating type constraints to ensure that the appropriate values are inferred for the new constraint variables. We now give a few examples to illustrate how these rules are *inferred* from method signatures in parameterized classes. In giving these examples, we assume that class `Stack` has already been parameterized as in the right column of Figure 5 (either manually, or using the INTRODUCE TYPE PARAMETER refactoring presented in Section 4.2).

Example 1. Consider the method call `s1.push(new Integer(1))` on line 4 in Figure 2. 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`$\leq\alpha$ and this constraint is generated from rule (17) in Figure 6(a).

Example 2. Similarly, the call `s1.pop()` on line 13 refers to method `void Stack<T1>.pop()`. If `s1` is of some parametric type, say `Stack<`α`>`, then [`s1.pop()`] $= \alpha$ and this constraint can be generated from rule (18).

Example 3. Consider the call `s1.moveFrom(s2)` on line 8. 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 $\alpha_2 \leq \alpha_1$ and this constraint is generated from rule (19).

As can be seen from Figure 6(a), 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. In the examples we have seen, type parameters occur as types of formal parameters, as return types, and as actual type parameters in the type of a formal parameter. Several other forms exist [8].

Figure 6(b) shows the constraints generated for the example. From these constraints, it follows that: `Integer`$\leq$`T1(s1)`, `Float`$\leq$`T1(s2)`, and `T1(s2)`$\leq$`T1(s1)`, and hence that `Float`$\leq$`T1(s1)`. Since `Number` is a common supertype of `Integer` and `Float`, a possible solution to this constraint system is:

$$\texttt{T1(s1)} \leftarrow \texttt{Number}, \texttt{T1(s2)} \leftarrow \texttt{Float}$$

However, several other solutions exist, such as the following uninteresting one:

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

Our current constraint solver relies on heuristics to guide it towards preferred solutions. The most significant of these heuristics are preferring more specific types over less specific ones, and avoiding marker interfaces such as `Serializable`.

Generating the refactored source code is now straightforward. The type of variable `s1` in the example program, for which we inferred `T1(s1)` = `Number`, is rewritten to `Stack<Number>`. Similarly, the types of `s2` and `v1` are rewritten to `Stack<Float>` and `Vector<Number>`, respectively. Furthermore, all downcasts are removed for which the type of the expression being cast is a subtype of the

program construct	constraint(s)	
method call E_1.push(E_2) to void Stack<T1>.push(T1)	[E_2]≤T1(E_1)	(17)
method call E.pop() to void Stack<T1>.pop()	[E.pop()]=T1(E)	(18)
method call E_1.moveFrom(E_2) to void Stack<T1>. moveFrom(Stack<? extends T1>)	T1(E_2)≤T1(E_1)	(19)
method call E_1.moveTo(E_2) to void Stack<T1>. moveTo(Stack<? super T1>)	T1(E_1)≤T1(E_2)	(20)
method call E_1.add(E_2) to boolean Vector<E>.add(E)	[E_2]≤E(E_1)	(21)

(a)

line(s)	constraint(s)	rule(s)
4	Integer≤T1(s1)	(11),(17)
6,7	Float≤T1(s2)	(11),(17)
8	T1(s2)≤T1(s1)	(19)
9	T1(s2)≤T1(s1)	(20)
13	[s1.pop()]=T(s1)	(18)
13	Number≤E(v1)	(21)

(b)

Fig. 6. (a) Additional constraint generation rules needed for the Infer Generic Type Arguments refactoring, automatically derived from method signatures (only constraints for methods used in the example program are shown). (b) 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 2, and rule numbers refer to Figures 6(a) and 1.

target type. For example, for the downcast `(Number)s1.pop()` on line 13, we inferred [`s1.pop()`] = `Number` enabling us to remove the cast.

4.2 Introduce Type Parameter

Consider a scenario where a programmer wants to apply Introduce Type Parameter to replace the type of the formal parameter `o` of `Stack.push()` with a new type parameter. Our solution requires a new form of constraint variable called context variable[4]. 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 a constraint variable α'. As an example, consider the type `Stack<T1>`. In the context of an instance `Stack<Number>`, the interpretation of `T1` is `Number`. Now, if we have a variable `x` of type `Stack<Number>`, then the interpretation of `T1` in the context of the type of `x` is `Number` and we will denote this fact by $\mathcal{I}_{[\texttt{x}]}(T1) = \texttt{Number}$. Here, $\mathcal{I}_{[\texttt{x}]}$ is an *interpretation function* that maps the formal type parameter[5] `T1` of `Stack` to the type with which it is instantiated in type [`x`]. For a more interesting example, consider the call `s1.push(new Integer(1))` on line 4 of Figure 2. For this call to be type-correct, the type `Integer` of actual parameter `new Integer(1)` must be a subtype of the formal parameter `o` of `Stack.push()` *in the context of the type of* `s1`. This can be expressed by a constraint $\texttt{Integer} \leq \mathcal{I}_{[\texttt{s1}]}([\texttt{o}])$. Note that we cannot simply require that `Integer`≤[`o`] because if `Stack` becomes a parameterized class `Stack<T1>`, and the type of `o` becomes `T1`, then `T1` is out of scope on line 4 (in addition, `Integer` is not a subtype of `T1`).

[4] Also required are *wildcard variables* to model cases where Java's typing rules *require* the introduction of wildcard types due to method overriding [13].

[5] For parameterized types with multiple type parameters such as `HashMap`, the interpretation function provides a binding for each of them [13].

line(s)	constraint(s)	
30	[o] ≤ E(v2)	(i)
33	E(v2) ≤ [Stack.pop()]	(ii)
36	$\mathcal{I}_{[s3]}$([Stack.pop()]) ≤ [o]	(iii)
39	[Stack.pop()] ≤ $\mathcal{I}_{[s4]}$([o])	(iv)

(a)

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

(b)

Fig. 7. (a) Type constraints generated for class `Stack` of Figure 2 when applying Introduce Type Parameter. (b) Solution to the constraints computed by our algorithm.

Figure 7(a) shows some of the constraints generated for class `Stack` of Figure 2. For these constraints, the algorithm by Kieżun et al. [13] computes the solution shown in Figure 7(b). This solution can be understood as follows. The type of `o` has become a new type parameter `T1` because this declaration was selected by the user. From constraints (i) and (ii) in Figure 7, it follows that `E(v2)` and [`Stack.pop()`] must each be a supertype of `T1`, and from constraint (iii) it can be seen 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, and `T1` is selected because the choice of `Object` would lead to a violation of constraint (iii).

Taking into account constraint (ii), it follows that `E(v2)` = `T1`. Now, for $\mathcal{I}_{s3}$([`Stack.pop`()]), the algorithm may choose any subtype of `T1`, and it heuristically[6] chooses `? extends T1`. Likewise, the type `? super T1` is selected for $\mathcal{I}_{[s4]}$([o]).

At this point, determining how the rewrite the source code is straightforward. From Figure 7(b), it can be seen that type of `o` 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 `? extends T1` was inferred for $\mathcal{I}_{[s3]}$([`Stack.pop`()]). By a similar argument, the type of `s4` is rewritten to `Stack<? super T1>`. The right column of Figure 5 shows the result.

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*.

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

In the program of Figure 2, `Vector`s are used in two places (variable `v1` declared on line 11 and field `v2` declared on line 25). 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`:

- Some methods in `Vector` are not supported by `ArrayList`. E.g, the example program calls `Vector.addElement()` on line 30, a method not declared in `ArrayList`. In this case, the call can be replaced with a call to `ArrayList.add()`, but other cases require the introduction of more complex expressions, or preclude migration altogether.
- Opportunities for migration may be limited when applications interact with libraries. For example, variable `v1` declared on line 11 serves as the actual parameter in a call to a constructor `JTree(Vector)` on line 19. Changing the type of `v1` to any supertype of `Vector` would render this call type-incorrect. Hence, the allocation site labeled `A1` cannot be migrated to `ArrayList`.
- Migrating one class may require migrating another. Consider the call on line 45 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[7]. 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 46 and 47 with calls to `Iterator.hasNext()` and `Iterator.next()`.
- If a `Vector` is accessed concurrently, then preservation of synchronization behavior is important. This is accomplished by introducing *synchronization wrappers*. This issue does not arise in the program of Figure 2 because it is single-threaded; the paper by Balaban et al. [1] presents an example.

We have developed a Replace Class refactoring that addresses these migration problems. This refactoring relies on a *migration specification* that specifies for each method in the source class how it is to be rewritten. Figure 8 shows the fragments of the specification for performing the migration from `Vector` to `ArrayList` and from `Enumeration` to `Iterator` needed for the example program (the complete specification can be found in [1]). Migration specifications only have to be written *once* for each pair of (source,target) classes.

We adapt the type constraints formalism of Section 2 as follows to implement Replace Class. For each source class S and target class T in a migration, the type system is extended with types $S^{\top}$ and $S_{\perp}$, such that $S \leq S^{\top}$, $T \leq S^{\top}$, $S_{\perp} \leq S$,

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

```
(1)  new Vector(), unsynchronized                  → new ArrayList()
(2)  new Vector(), synchronized                    → Collections.synchronizedList(
                                                         new ArrayList())
(3)  boolean Vector:receiver.add(Object:v)         → boolean receiver.add(v)
(4)  void Vector:receiver.addElement(Object:v)     → boolean receiver.add(v)
(5)  Object Vector:receiver.remove(int:i)          → Object receiver.remove(i)
(6)  int Vector:receiver.size()                    → int receiver.size()
(7)  boolean Vector:receiver.isEmpty()             → boolean receiver.isEmpty()
(8)  Enumeration Vector:receiver.elements()        → Iterator receiver.iterator()
(9)  boolean Enumeration:receiver.hasMoreElements() → boolean receiver.hasNext()
(10) Object Enumeration:receiver.nextElement()     → Object receiver.next()
```

Fig. 8. Specification used for migrating the example program

line(s)	constraint(s)	
11	[A1]≤[v1], [A1]≤Vector$^{\top}$, Vector$_{\perp}$≤[A1]	(i),(ii),(iii)
19	[v1]≤Vector	(iv)
27	[A2]≤[v2], [A2]≤Vector$^{\top}$, Vector$_{\perp}$≤[A2]	(v),(vi),(vii)
30	[o]≤Object	(viii)
33,42	[v2]≤Collection	(ix)
45	[s5.v2]=Vector → [s5.v2.elements()]=Enumeration	(x)
45	[s5.v2]=ArrayList → [s5.v2.elements()]=Iterator	(xi)

Fig. 9. Some of the type constraints generated for the application of the Replace Class refactoring to the program of Figure 2

$S_{\perp} \leq T$[8]. Moreover, rule (11) of Figure 1 is adapted to generate constraints for allocation sites that permit the migration from source types to target types. For example, constraints (ii) and (iii) in Figure 9 are generated for the allocation site labeled `A1` on line 11 in Figure 2.

For a migration from a class S to a class T, a call to a method in S gives rise to *implication constraints* of the form $\alpha = K \rightarrow c$. Here, α is a constraint variable, K is a type, and c is an unconditional constraint that must be satisfied if the condition holds. For example, consider the call `s5.v2.elements()` on line 45, which can be rewritten to an expression `s5.v2.iterator()` (see Figure 8). The implication constraints (x) and (xi) in Figure 9 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()`.

Solving systems of implication constraints may require backtracking. However, it is often possible to perform simplifications that eliminate the need for implications. As an example, consider the call `v2.addElement(o)` on line 30. If the type of `v2` remains `Vector`, we must constrain `o` to be a subtype of the formal parameter of `Vector.addElement()`, which can be expressed by the constraint: [`v2`]=`Vector` → [`o`]≤[*Param*(`0,Vector:addElement(Object)`)]. Similarly, for the case where the type of `v2` becomes `ArrayList`, we have: [`v2`]=`ArrayList` → [`o`]≤[*Param*(`0,ArrayList:add(Object)`)]. These constraints can be combined into a single unconditional constraint [`o`]≤`Object` (constraint (viii)) because both *Param*-expressions evaluate to `Object`.

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

```
class Client {
  public static void main(String[] args){
    Stack s1 = new Stack();
    s1.push(new Integer(1));
    Stack s2 = new Stack();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Number n = (Number)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 ArrayList v2;
  public Stack(){
    v2 = new ArrayList();
  }
  public void push(Object o){
    v2.add(o);
  }
  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 static void print(Stack s5){
    Iterator e = s5.v2.iterator();
    while (e.hasNext())
      System.out.println(e.next());
  }
}
```

Fig. 10. The example program after the application of Replace Class refactoring

From constraints (i) and (iv) in Figure 9, it follows that [`A1`]≤[`v1`]≤`Vector`, implying that the type `A1` must remain `Vector`. However, the typing [`A2`] ← `ArrayList`, [`v2`] ← `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 10.

A few additional complicating factors exist. In order to preserve synchronization behavior, we 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 `Vector`s, we attempt a translation that introduces a synchronization wrapper (rule (2) of Figure 8). Hence, there are three alternatives for each `Vector` allocation site: it can remain a `Vector`, become an unwrapped `ArrayList`, or a wrapped `ArrayList`. Preserving the behavior of downcasts requires additional constraints [1].

6 Related Work

Opdyke [15, 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 constraints (8) and (9). 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 constraints (1), (3), (12), and (14).

Fowler [7] presents a comprehensive classification of a large number of refactorings, which includes step-by-step directions on how to perform each of these 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 [23] 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 [11] present an informal algorithm for detecting overspecific 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.* [20] 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 [19] 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 [21,18] 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 [6] 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 fully implemented.

Donovan and Ernst [4] 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 predates Java generics (arrays of parameterized types are inferred, which are not allowed in Java) and was never fully implemented.

Donovan et al. [5] 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 [25] 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.

7 Conclusion

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 [8,1,13].

Acknowledgments

The contents of this paper are based on the author's joint research with Ittai Balaban, Dirk Bäumer, Julian Dolby, Michael Ernst, Robert Fuhrer, Markus Keller, Adam Kieżun, and Bjorn De Sutter. Ittai Balaban, Adam Kieżun, and Jan Vitek provided valuable comments on drafts of this paper.

References

1. Balaban, I., Tip, F., Fuhrer, R.: Refactoring support for class library migration. In: Proc. OOPSLA, pp. 265–279 (2005)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, London, UK (2000)

3. De Sutter, B., Tip, F., Dolby, J.: Customization of Java library classes using type constraints and profile information. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 585–610. Springer, Heidelberg (2004)
4. Donovan, A., Ernst, M.: Inference of generic types in Java. Tech. Rep. MIT/LCS/TR-889, MIT (March 2003)
5. Donovan, A., Kieżun, A., Tschantz, M., Ernst, M.: Converting Java programs to use generic libraries. In: Proc. OOPSLA, pp. 15–34 (2004)
6. Duggan, D.: Modular type-based reverse engineering of parameterized types in Java code. In: Proc. OOPSLA, pp. 97–113 (1999)
7. Fowler, M.: Refactoring. In: Improving the Design of Existing Code, Addison-Wesley, London, UK (1999)
8. Fuhrer, R., Tip, F., Kieżun, A., Dolby, J., Keller, M.: Efficiently refactoring Java applications to use generic libraries. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 71–96. Springer, Heidelberg (2005)
9. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington, Technical Report 91-08-04 (1991)
10. Griswold, W.G., Notkin, D.: Automated assistance for program restructuring. ACM Trans. Softw. Eng. Methodol. 2(3), 228–269 (1993)
11. Halloran, T.J., Scherlis, W.L.: 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 (September 2002)
12. Kerievsky, J.: Refactoring to Patterns. Addison-Wesley (2004)
13. Kieżun, A., Ernst, M., Tip, F., Fuhrer, R.: Refactoring for parameterizing Java classes. In: Proc. ICSE, pp. 437–446 (2007)
14. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. on Softw. Eng. 30(2), 126–139 (2004)
15. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, University Of Illinois at Urbana-Champaign (1992)
16. Opdyke, W.F., Johnson, R.E.: Creating abstract superclasses by refactoring. In: The ACM 1993 Computer Science Conf. (CSC'93), February 1993, pp. 66–73 (1993)
17. Palsberg, J., Schwartzbach, M.: Object-Oriented Type Systems. John Wiley & Sons, West Sussex, England (1993)
18. Snelting, G., Tip, F.: Understanding class hierarchies using concept analysis. In: ACM Trans. on Programming Languages and Systems, May 2000, pp. 540–582 (2000)
19. Steimann, F., Mayer, P.: Type access analysis: Towards informed interface design. In: Proc. TOOLS Europe (to appear, 2007)
20. Steimann, F., Mayer, P., Meißner, A.: Decoupling classes with inferred interfaces. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 1404–1408. Springer, Heidelberg (2006)
21. Streckenbach, M., Snelting, G.: Refactoring class hierarchies with KABA. In: Proc. OOPSLA, pp. 315–330 (2004)
22. Tip, F., Kieżun, A., Bäumer, D.: Refactoring for generalization using type constraints. In: Proc. OOPSLA, pp. 13–26 (2003)
23. Tokuda, L., Batory, D.: Evolving object-oriented designs with refactorings. Kluwer Journal of Automated Software Engineering, 89–120 (August 2001)
24. Torgersen, M., Plesner Hansen, C., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: Proc. of the 2004 ACM symposium on Applied computing, pp. 1289–1296 (2004)
25. von Dincklage, D., Diwan, A.: Converting Java classes to use generics. In: Proc. OOPSLA, pp. 1–14 (2004)