Refactoring for Generalization using Type Constraints

Frank Tip
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
tip@watson.ibm.com

Adam Kiezun*
MIT Lab for Computer Science
200 Technology Square, Cambridge, MA 02139, USA
akiezun@mit.edu

Dirk Bäumer
IBM Research OTI Labs
Oberdorfstrasse 8, CH-8001 Zürich, Switzerland
dirk_baeumer@ch.ibm.com

ABSTRACT

Refactoring is the process of applying behavior-preserving transformations (called "refactorings") in order to improve a program's design. Associated with a refactoring is a set of preconditions that must be satisfied to guarantee that program behavior is preserved, and a set of source code modifications. An important category of refactorings is concerned with generalization (e.g., EXTRACT INTERFACE for re-routing the access to a class via a newly created interface, and PULL UP MEMBERS for moving members into a superclass). For these refactorings, both the preconditions and the set of allowable source code modifications depend on interprocedural relationships between types of variables. We present an approach in which *type constraints* are used to verify the preconditions and to determine the allowable source code modifications for a number of generalization-related refactorings. This work is implemented in the standard distribution of Eclipse (see `www.eclipse.org`).

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming 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

Design, Languages, Theory

Keywords

Refactoring, type constraints, program analysis, subtyping, class hierarchy

*This work was done while the second author was at IBM Research OTI Labs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'03, October 26–30,2003,Anaheim,California,USA.
Copyright 2003 ACM 1-58113-712-5/03/0010 ...$5.00.

1. INTRODUCTION

Refactoring [6, 15] is the process of modifying a program's source code without changing its behavior, with the objective of improving the program's design. A refactoring operation is identified by a name, a set of preconditions under which it is allowed and the actual source-code transformation that is performed. Recently, code-centric development methodologies such as "Extreme Programming" [2] have embraced refactoring because it fits well with their goal of continuously improving source code quality. This has resulted in a renewed interest in tools that verify the preconditions of refactorings, and that perform the actual source code updates. Several popular development environments such as Eclipse [5] and IntelliJ IDEA [12] incorporate refactoring capabilities.

An important category of refactorings is concerned with generalization [6, Chapter 11][14], e.g., PULL UP MEMBERS for moving a member into a superclass so it can be shared by a number of subclasses, and EXTRACT INTERFACE for redirecting access to a class via a newly created interface. The latter refactoring involves updating declarations of variables, parameters, method return types, and fields to use the newly added interface. Not updating these declarations leads to overspecific variable declarations, which conflicts with the principles of object-oriented design [13].

This paper proposes the use of an existing framework of *type constraints* [16] to address various aspects of refactorings related to generalization. We show how type constraints can be used to efficiently compute the maximal set of allowable source-code modifications for EXTRACT INTERFACE, and demonstrate that this solution preserves type-correctness (we argue informally that preservation of program behavior is implied). We also show how type constraints serve to succinctly state the preconditions for PULL UP MEMBERS, and briefly discuss other refactorings that can be modeled similarly. Our work is implemented in the standard distribution of Eclipse [5], and is available from `www.eclipse.org`.

1.1 Motivating Example

Figure 1 shows a Java program P_1 that illustrates some of the challenges associated with EXTRACT INTERFACE. In Figure 1, class `List` defines array-based lists that support operations `add()` for adding an element, `addAll()` for adding the contents of another list, `sort()` for sorting a list, and `iterator()` for iterating through a list without being aware of its implementation. Also shown is a class `Client` with a

```
interface Bag {
  public java.util.Iterator iterator();
  public [List] add(Comparable element);
  public [List] addAll([List] v0);
}
class List implements Bag {
  int size = 0;
  Comparable[] elems = new Comparable[10];
  public java.util.Iterator iterator(){
    return new Iterator(this);
  }
  public [List] add(Comparable e){
    if (this.size+1 == this.elems.length){
      Comparable[] newElems = new Comparable[2 * this.size];
      System.arraycopy(this.elems, 0, newElems, 0, this.size);
      this.elems = newElems;
    }
    this.elems[this.size++] = e;
    return this;
  }
  public [List] addAll([List] v1) {
    for(java.util.Iterator i=v1.iterator(); i.hasNext(); ){
      this.add((Comparable)i.next())
    }
    return this;
  }
  public void sort() { /* insertion sort */
    for (int i = 1; i < this.size; i++) {
      Comparable e1 = this.elems[i];
      int j = i;
      while ((j > 0) && (this.elems[j-1].compareTo(e1) > 0)) {
        this.elems[j] = this.elems[j-1];
        j--;
      }
      this.elems[j] = e1;
    }
  }
}
class Iterator implements java.util.Iterator {
  private int count = 0;
  private List v2;
  Iterator(List v3){ this.v2 = v3;}
  public boolean hasNext(){ return this.count < this.v2.size; }
  public Object next(){ return this.v2.elems[this.count++]; }
  public void remove(){ throw new UnsupportedOperationException(); }
}
class Client {
  public static void main(String[] args){
    List v4 = createList();
    populate(v4); update(v4); sortList(v4); print(v4);
  }
  static List createList(){ return new List();}
  static void populate([List] v5){ v5.add("foo").add("bar");}
  static void update([List] v6) {
    [List] v7 = new List().add("zap").add("baz");
    v6.addAll(v7);
  }
  static void sortList(List v8) { v8.sort(); }
  static void print([List] v9) {
    for (java.util.Iterator iter = v9.iterator(); iter.hasNext(); )
      System.out.println("Object:  " + iter.next());
  }
}
```

Figure 1: Example program P_1 illustrating the EXTRACT INTERFACE refactoring. Declarations shown in boxes can be given type `Bag` instead of type `List`.

`main()` method that models a typical usage of `List`. Now, let us assume that we want to (further) hide the implementation details of `List`, to make it easier to switch to a different (e.g., linked) list implementation. To do so, we create an interface `Bag` that declares `add()`, `addAll()`, and `iterator()`. Then, we make `Bag` a superinterface of `List`. In Figure 1, these basic steps of EXTRACT INTERFACE have already been performed.

At this point, `Bag` is not yet *used* because P_1 contains no element whose declared type is `Bag`. As the main goal of EXTRACT INTERFACE is to re-route the access to `List` via the `Bag` interface, we want to update the declarations of variables, parameters, fields, and method return types so that they use, where possible, `Bag` instead of `List`. In program P_1, `v0`, `v1`, `v2`, `v3`, `v4`, `v5`, `v6`, `v7`, `v8` and `v9`, and the return types of `List.add()`, `List.addAll()`, `Bag.add()`, `Bag.addAll()` and `Client.createList()` are of type `List`. Which of these can be given type `Bag` without affecting program behavior? Careful examination of the program reveals that:

- Field `Iterator.v2` cannot be declared as type `Bag` because the fields `size` and `elems` are accessed from `v2`, but not declared in `Bag`.
- Parameter `v3` (declared in class `Iterator`'s constructor) is assigned to `v2`, requiring that the declared type of `v3` be equal to or a subtype of the declared type of `v2`. As the declaration of `v2` cannot be updated, the declaration of `v3` cannot be updated either.
- The type of `v8` (declared in `Client.sortList()`) must remain `List` because `sort()` is called on `v8`, and `sort()` is not declared in `Bag`.
- Variable `v4` (declared in `Client.main()`) is passed as an argument to `Client.sortList()`, implying an assignment `v8 = v4`. Hence, `v4`'s declared type must be equal to, or a subtype of `v8`'s declared type, which cannot be changed. So, `v4`'s type cannot change either.
- Finally, the return type of `Client.createList()` cannot be updated because the return value of this method is assigned to `v4`, whose declared type must remain `List`, as was discussed above.

To summarize our findings, only `v0`, `v1`, `v5`, `v6`, `v7`, `v9` and the return types of `List.add()`, `List.addAll()`, `Bag.add()`, and `Bag.addAll()` can be given type `Bag` instead of `List`. In Figure 1, these declarations are shown boxed. Clearly, care must be taken when updating declarations.

1.2 Organization of this Paper

Section 2 presents a model of type constraints for a substantial Java subset. Its use in modeling EXTRACT INTERFACE is presented in Section 3. Section 4 studies other refactorings that can be accommodated using this model. Section 5 presents implementation issues related to Java features not previously discussed. Sections 6 and 7 discuss related work and directions for future work, respectively.

1.3 Assumptions

We make the *closed-world assumption* that a refactoring tool has access to a program's full source code, and that only the behavior of this program needs to be preserved.

Furthermore, we will not consider the introduction of type casts, which can expand the applicability of refactorings (e.g., in the example of Figure 1, `v4`, `v8`, and the return type of `Client.createList()` can be given type `Bag` if a cast to `List` is inserted in `Client.sortList()`). We believe that introducing casts does not improve a program's design.

Finally, we currently disallow refactorings such as EXTRACT INTERFACE in the presence of overloading (i.e., having, in one class, two or more methods with identical names but different argument types). A call to an overloaded method is resolved to the method whose formal parameter types most closely match the types of the actual parameters [10, Section 15.12.2.2]. Refactorings that change the declared types of method parameters must take care to ensure that program behavior is not affected. Due to the static nature of overloading resolution, sufficient conditions that guarantee preservation of behavior can be designed, but their precise formulation remains as an item for future work. Section 5 discusses overloading and other language features that pose challenges for refactoring.

2. FORMAL MODEL

Palsberg and Schwarzbach [16] introduced a model of *type constraints* for the purpose of checking whether a program conforms to a language's type system. If a program satisfies all type constraints, no type violations will occur at runtime (e.g., no method $m(\cdots)$ is invoked on an object whose class does not define or inherit $m(\cdots)$). In our setting, we start with a well-typed program, and use type constraints similar to those in [16] to determine that declarations can be updated, or that members can be moved without affecting a program's well-typedness.

2.1 Notation and Terminology

We will adopt the term *declaration element* to refer 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, $All(P, C)$ denotes the set of all declaration elements of type C in program P. For program P_1 of Figure 1, using method names to represent method return types, we have:

$$All(P_1, \texttt{List}) = \{\ \texttt{v0}, \texttt{v1}, \texttt{v2}, \texttt{v3}, \texttt{v4}, \texttt{v5}, \texttt{v6}, \texttt{v7}, \texttt{v8}, \texttt{v9}, \texttt{Bag.add}(), \texttt{Bag.addAll}(), \texttt{List.add}(), \texttt{List.addAll}(), \texttt{Client.createList}()\ \}$$

In what follows, v, v' denote variables, M, M' denote methods, F, F' denote fields, C, C' denote classes, I, I' denote interfaces, and T, T' denote types[1]. It is important to note that the symbol M denotes a method together with all its signature and return type information and the reference to its declaring type. Similarly, F and C denote a field and a type, respectively, together with its name, type in which it is declared and, in the case of fields, its declared type.

Moreover, the notation E, E' will be used to denote an expression or declaration element at a specific point in the program, corresponding to a specific node in the program's abstract syntax tree. We will assume that type information about expressions and declaration elements is available from a type-checker or compiler.

A method M is *virtual* if M is not a constructor, M is not private and M is not static. Definitions 2.1 and 2.2 below

[1]In this paper, the term *type* will denote a class or an interface.

$[E]$	the type of expression or declaration element E
$[M]$	the declared return type of method M
$[F]$	the declared type of field F
$Decl(M)$	the type that contains method M
$Decl(F)$	the type that contains field F
$Param(M,i)$	the i-th formal parameter of method M
$T' \leq T$	T' is equal to T, or T' is a subtype of T
$T' < T$	T' is a proper subtype of T (i.e., $T' \leq T$ and not $T \leq T'$)
$super(C)$	the superclass of class C

Figure 2: Type constraint notation.

define concepts of *overriding*[2] and *root definitions* for virtual methods. Definition 2.3 defines a notion of *hiding* for fields that will be needed for the Pull Up Members refactoring in Section 4.

Def. 2.1 (overriding). *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. In this case, we also say that M' is overridden by M.*

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

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

Def. 2.3 (hiding). *Field F in type C hides field F' in type B if F and F' have identical names and C is a subtype of B. Then, we also say that F' is hidden by F.*

2.2 Type Constraints

Figure 2 shows the notation that will be used to express type constraints. A *constraint variable* α is one of the following: C (a type constant), $[E]$ (representing the type of an expression or declaration element E), $Decl(M)$ (representing the type in which method M is declared), or $Decl(F)$ (representing the type in which field F is declared). A *type constraint* is a relationship between two or more constraint variables that must hold in order for a program to be type-correct. In this paper, a *type constraint* has one of the following forms: (i) $\alpha_1 \triangleq \alpha_2$, indicating that α_1 is defined to be the same as α_2 (ii) $\alpha_1 \leq \alpha_2$, indicating that α_1 must be equal to or be a subtype of α_2, (iii) $\alpha_1 = \alpha_2$, indicating that $\alpha_1 \leq \alpha_2$ and $\alpha_2 \leq \alpha_1$, (iv) $\alpha_1 < \alpha_2$, indicating that $\alpha_1 \leq \alpha_2$ but not $\alpha_2 \leq \alpha_1$, (v) $\alpha_1^L \leq \alpha_1^R$ **or** $\cdots$ **or** $\alpha_k^L \leq \alpha_k^R$, indicating that $\alpha_j^L \leq \alpha_j^R$ must hold for at least one j, $1 \leq j \leq k$.

In discussions about types of expressions and subtype-relationships that occur in a specific program P, we will use the notation of Figure 2 with subscript P. For example, $[E]_P$ denotes the type of expression E in program P, $Decl_P(M)$ denotes the declared return type of method M in program P, $Decl_P(F)$ denotes the declared type of field F in program P, and $T' \leq_P T$ denotes a subtype-relationship that occurs in program P. In cases where the program under consideration is unambiguous, we will frequently omit these P-subscripts (in particular, the subscripts of the subtype operators '$\leq_P$' and '$<_P$' can often be omitted, because programs P and P' have identical class hierarchies). We can now define what it means for a program to be type-correct.

[2]Note that, according to Definition 2.1, a virtual method overrides itself.

Def. 2.4 (type correctness). *We say that a type constraint $[E] \leq [E']$ is satisfied by a program P if and only if $[E]_P \leq_P [E']_P$ (similar cases exist for the other forms of type constraints). Constraints of form $\alpha_1 \triangleq \alpha_2$ are always satisfied. We say that program P is type-correct if and only if all constraints in $TC(P)$ are satisfied by P.*

2.3 Generating Type Constraints

Figure 3 lists the type constraints implied by a number of common Java features, which were carefully designed to reflect the semantics of Java [10]. Due to space limitations, we only discuss a few of the more interesting rules in detail.

Rules (1)–(18)[3] define relationships between types of different expressions and declaration elements that must hold in order for the program to be type-correct. Rule (1) states that the type of the left-hand side of an assignment must either be the same as, or a supertype of the type of the right-hand side. For a call $E.m(\cdots)$ to a virtual method M, we have that: (i) the type of the call-expression is the same as M's return type (rule (2)[4]), (ii) the type of each actual parameter must be the same as, or a supertype of the corresponding formal parameter (rule (3)), and (iii) a method with the same signature as M must be declared in $[E]$ or one of its supertypes (rule (4)). This last constraint involves determining a set of methods $M_1, \cdots, M_k$ overridden by M using Definition 2.2, and requiring $[E]$ to be a subtype of one or more of $Decl(M_1), \cdots, Decl(M_k)$.

Rules (8)–(10) are concerned with overriding. Changing a parameter's type need not by itself affect type-correctness, but it may affect virtual dispatch (and program) behavior. Hence, we require that types of parameters (rule (8)) and return types (rule (9)) of overriding methods correspond. Rule (10) states that no single type can contain two methods with the same signature, and will be needed in Section 4 to check for cases where methods cannot be moved.

For a cast from class C to class C', an ordering relationship between C and C' in the class hierarchy is required (rule (16)). Note that this constraint does not apply if the type of the expression or the type being cast to is an interface [10, Section 5.5]. Rules (19)–(20) define the type of certain kinds of expressions (`this`-pointers and allocation sites, respectively). Finally, rules (21) and (22) define the type containing a specific method and field, respectively.

Rules (23)–(26) define the types of declaration elements by referring to their declared types. We conclude this discussion with a remark. Some of the constraints of Figure 3 (in particular, (8) and (11)) go beyond the type-checking that is routinely performed by Java compilers. These rules are needed to ensure preservation of program behavior.

2.4 Classifying Type Constraints

Definition 2.5 below defines $TC(P)$ to be the set of all type constraints generated for program P, according to the rules of Figure 3.

[3] Rules (17) and (18) in Figure 3 are only shown for completeness, and are not affected by the refactorings we consider.

[4]Rules (2), (5), (13), and (19)–(22) *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.

program construct	implied type constraint(s)	
assignment $E_1 = E_2$	$[E_2] \leq [E_1]$	(1)
method call $E.m(E_1, \cdots, E_n)$ to a virtual method M	$[E.m(E_1, \cdots, E_n)] \triangleq [M]$	(2)
	$[E_i] \leq [Param(M, i)]$	(3)
	$[E] \leq Decl(M_1)$ **or** $\cdots$ **or** $[E] \leq Decl(M_k)$ where $RootDefs(M) = \{ M_1, \cdots, M_k \}$	(4)
access $E.f$ to field F	$[E.f] \triangleq [F]$	(5)
	$[E] \leq Decl(F)$	(6)
`return` E in method M	$[E] \leq [M]$	(7)
M' overrides M, $M' \neq M$	$[Param(M', i)] = [Param(M, i)]$	(8)
	$[M'] = [M]$	(9)
	$Decl(M') < Decl(M)$	(10)
F' hides F	$Decl(F') < Decl(F)$	(11)
constructor call `new` $C(E_1, \cdots, E_n)$ to constructor M	$[E_i] \leq [Param(M, i)]$	(12)
direct call $E.m(E_1, \cdots, E_n)$ to method M	$[E.m(E_1, \cdots, E_n)] \triangleq [M]$	(13)
	$[E_i] \leq [Param(M, i)]$	(14)
	$[E] \leq Decl(M)$	(15)
cast $(C)E$	$[E] \leq [(C)E]$ **or** $[(C)E] \leq [E]$ if $[E]$ is a class	(16)
for every type T	$T \leq$ `java.lang.Object`	(17)
	`[null]` $\leq$ `T`	(18)
implicit declaration of `this` in method M	`[this]` $\triangleq Decl(M)$	(19)
expression `new` $C(E_1, \cdots, E_n)$	[`new` $C(E_1, \cdots, E_n)] \triangleq C$	(20)
declaration of method M (declared in type T)	$Decl(M) \triangleq T$	(21)
declaration of field F (declared in type T)	$Decl(F) \triangleq T$	(22)
explicit declaration of variable or method parameter $T\ v$	$[v] \triangleq T$	(23)
declaration of method M with return type T	$[M] \triangleq T$	(24)
declaration of field F with type T	$[F] \triangleq T$	(25)
cast $(T)E$	$[(T)E] \triangleq T$	(26)

Figure 3: Type constraints for a set of core Java language features. Rules (1)–(22) define the types of expressions and impose constraints between the types of expressions and declaration elements. Rules (23)–(26) define the types of declaration elements.

DEF. 2.5 ($TC_{\text{fixed}}(P)$, $TC_{\text{var}}(P)$, $TC(P)$). *Let P be a program. Then, $TC_{\text{fixed}}(P)$ denotes the set of type constraints inferred for program P according to rules (1)–(22). Further, $TC_{\text{var}}(P)$ is the set of constraints inferred for P according to rules (23)–(26). Moreover, $TC(P)$ denotes the set $TC_{\text{fixed}}(P) \cup TC_{\text{var}}(P)$.*

The partitioning of $TC(P)$ into $TC_{\text{fixed}}(P)$ and $TC_{\text{var}}(P)$ in Definition 2.5 is specific to the EXTRACT INTERFACE refactoring, and reflects the fact that EXTRACT INTERFACE should preserve some, but not all type constraints for program P. In particular, the type constraints in $TC_{\text{fixed}}(P)$ corresponding to program constructs that reappear unmodified in P' must be satisfied by program P' after applying EXTRACT INTERFACE. However, the type constraints in $TC_{\text{var}}(P)$ are inferred from declaration elements in P, and need not necessarily be satisfied by P', because EXTRACT INTERFACE updates declaration elements to refer to a newly created interface, and updating a declaration implies that a different type constraint occurs in $TC_{\text{var}}(P')$.

It is important to note that the way in which type constraints are partitioned into a "fixed" and a "variable" subset depends on the refactoring under consideration. In the case of EXTRACT INTERFACE, the types of declaration elements are variable, and all other type constraints are fixed. In the case of the PULL UP MEMBERS refactoring (discussed in Section 4), the locations of members in the hierarchy is variable, and the types of declaration elements are fixed.

2.5 Type Constraints for Program P_1

Figure 4 shows the constraints in $TC_{\text{fixed}}(P_1)$ related to types `List` and `Bag`. Here, each expression $Decl(M)$ has been reduced to the (constant) class in which M is declared using rule (21). We can perform this simplification in the context of EXTRACT INTERFACE because this refactoring does not affect the declaring classes of members. In general, we can simplify constraints using "constant definitions" (i.e., constraints from $TC_{\text{fixed}}(P)$ that use the $\triangleq$ symbol) because the same definitions will occur in the refactored program. We consider the derivation of some of constraints in $TC_{\text{fixed}}(P_1)$:

(a) For the call to `List.add()` on receiver expression `v5` in method `Client.populate()`, we find using rule (4) that [`v5`] $\leq$ *Decl*(`Bag.add()`) = `Bag` (here, we used the fact that *RootDefs*(`List.add()`) = { `Bag.add()` }).

(b) For the other call to `List.add()` in `Client.populate()` we have that [`v5.add("foo")`] $\leq$ *Decl*(`Bag.add()`) = `Bag`. We then simplify [`v5.add("foo")`] to [`List.add()`] using rule (2), which yields [`List.add()`] $\leq$ `Bag`.

(c) For each field access `this.size` in method `List.sort()`, we find using rule (6) that [`this`] $\leq$ *Decl*(`List.size`)

method(s)	constraint(s)	rule(s)
List.add(),Bag.add()	[Bag.add()] = [List.add()]	(9)
List.addAll(), Bag.addAll()	[v0] = [v1] [Bag.addAll()] = [List.addAll()]	(8) (9)
List.iterator()	List ≤ [v3]	(19), (12)
List.add()	List ≤ [List.add()]	(7), (19)
List.addAll()	[v1] ≤ Bag List ≤ [List.addAll()]	(4) (19), (7)
Iterator.Iterator()	[v3] ≤ [v2]	(1)
Iterator.hasNext()	[v2] ≤ List	(6)
Iterator.next()	[v2] ≤ List	(6)
Client.main()	[Client.createList()] ≤ [v4] [v4] ≤ [v5]; [v4] ≤ [v6] [v4] ≤ [v8]; [v4] ≤ [v9]	(1), (13) (14);(14) (14);(14)
Client.createList()	List ≤ [Client.createList()]	(20), (7)
Client.populate()	[v5] ≤ Bag [List.add()] ≤ Bag	(4) (2), (4)
Client.update()	[List.add()] ≤ [v7] [List.add()] ≤ Bag [v6] ≤ Bag; [v7] ≤ [v1]	(1) (2), (4) (4);(3)
Client.sortList()	[v8] ≤ List	(4)
Client.print()	[v9] ≤ Bag	(4)

Figure 4: Type constraints $TC_{\text{fixed}}(P_1)$ for program P_1 of Figure 1. Only nontrivial constraints related to types `List` and `Bag` are shown.

`= List`. We can simplify the left side using rule (19), which yields `List` ≤ `List`. Note that this trivial constraint does not constrain the type of any variable. Similar constraints occur for accesses to field `elems`.

(d) For method `Client.main()`, we infer using rules (1) and (13) that [`Client.createList()`] ≤ [`v4`]. Applications of rule (14) yield [`v4`] ≤ [`v5`], [`v4`] ≤ [`v6`], [`v4`] ≤ [`v8`], and [`v4`] ≤ [`v9`].

3. EXTRACT INTERFACE

We can now state the refactoring problem of Section 1 as follows: We want to identify a maximal set of declaration elements $G \subseteq All(P_1, \texttt{List})$ such that the following holds in the refactored program P_1':

$$[E] \triangleq \texttt{Bag} \in TC_{\text{var}}(P_1') \quad \textbf{if } E \in G, \text{ and}$$
$$[E] \triangleq \texttt{List} \in TC_{\text{var}}(P_1') \quad \textbf{if } E \in (All(P_1, \texttt{List}) \setminus G)$$

and such that all constraints in $TC(P_1')$ are satisfied. A naive approach to solve this problem would be to compute all possible values of G that satisfy the type constraints in $TC(P_1')$, and then select a maximal G. Assuming that $All(P_1, \texttt{List})$ contains N elements, 2^N possible values exist for G (each element can have type `Bag` or `List`). Hence, the cost of this naive approach is a prohibitive $O(2^N)$.

Observe, however, that the type constraints in $TC_{\text{fixed}}(P_1)$ already indicate which declaration elements cannot be updated. For example, from Figure 4 it can be seen that `List` ≤ [`v3`] ≤ [`v2`] ≤ `List`, which implies that `v2` and `v3` can only have type `List`. Definition 3.1 below formalizes this notion of "non-updatability".

DEF. 3.1 (NON-UPDATABLE DECLARATION ELEMENTS). *Let P be a program, let C be a class in P, 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:*

$$\begin{aligned}
&Bad(P,C,I) = \{\, E \mid E \in All(P,C),\ \textit{and}\\
&\quad (\ ([E]{\leq}T_1 \textbf{ or } \cdots \textbf{ or } [E]{\leq}T_k \in TC_{\text{fixed}}(P), \qquad \textbf{(a)}\\
&\qquad\quad I \not\leq_P T_1, \cdots, I \not\leq_P T_k)\\
&\qquad \textit{or}\\
&\qquad ([E]{\leq}[E'] \in TC_{\text{fixed}}(P), \qquad \textbf{(b)}\\
&\qquad\quad E' \notin All(P,C), I \not\leq_P [E'])\\
&\qquad \textit{or}\\
&\qquad ([E] = [E'] \in TC_{\text{fixed}}(P)\ \textit{or} \qquad \textbf{(c)}\\
&\qquad [E]{\leq}[E'] \in TC_{\text{fixed}}(P)\ \textit{or}\\
&\qquad [E]{<}[E'] \in TC_{\text{fixed}}(P),\ E' \in Bad(P,C,I))\ )\,\}
\end{aligned}$$

Part (a) of Definition 3.1 is concerned with constraints that are due to a method call $E.m(\cdots)$, and states that E cannot be given type I if a declaration of $m(\cdots)$ does not occur in (a supertype of) I. Part (b) of Definition 3.1 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 (c) handles the propagation of "badness" due to overriding, assignments, and parameter passing. For program P_1 of Figure 1, we have:

$$Bad(P_1, \texttt{List}, \texttt{Bag}) = \{\, \texttt{v2}, \texttt{v3}, \texttt{v4}, \texttt{v8}, \texttt{Client.createList()} \,\}$$

Hence, the declarations of `v0`, `v1`, `v5`, `v6`, `v7`, and `v9` and the return types of `List.add()`, `List.addAll()`, `Bag.add()`, and `Bag.addAll()` can be changed to `Bag`. Note that we reached the same conclusion via informal reasoning in Section 1.

An observant reader may have noticed that Definition 3.1 does not contain a case to deal with type constraints that arise due to casts. This is the case because rule (16) only applies to a cast expression $(T)E$ if T is a class, and changing a type cast $(C)E$ into $(I)E$ has the effect of *removing* a type constraint. Nevertheless, the type of a cast-expression may be constrained by assignments to other variables (and parameter-passing), as Figure 5 illustrates.

```
private static void sortAndPrintAllLists(List allLists){
  for (Iterator iter0 = allLists.iterator(); iter0.hasNext();) {
    List list0 = (List) iter0.next();
    list0.sort();
  }
  for (Iterator iter1 = allLists.iterator(); iter1.hasNext();) {
    List list1 = (List) iter1.next();
    System.out.println(list1.toString());
  }
}
```

Figure 5: Example method that uses type casts (this method is assumed to occur in class `Client` of Figure 1).

For the code in the first loop of Figure 5, we infer [`(List)iter0.next()`] $\leq$ [`list0`] (rule (1)) and [`list0`] $\leq$ *Decl*(`List.sort()) = List` (rule (4)). Hence, we have that: [`(List)iter0.next()`] $\leq$ `List`. Since the cast expression must have a type that is equal to, or a subtype of `List`, we cannot update it to type `Bag`. For the second loop, we find that [`(List)iter1.next()`] $\leq$ [`list1`] (rule (1)) and [`list1`] $\leq$ *Decl*(`Object.toString()) = Object` (rule (4)). Hence, it is only required that [`(List)iter1.next()`] $\leq$ `Object` which always holds (rule (17)). Hence, the cast in the second loop can be updated to `(Bag)iter1.next()`.

3.1 Justification

Theorem 3.2 states that updating the declaration elements in $All(P, C)$ that do not occur in $Bad(P, C, I)$ produces a program that is type-correct.

THEOREM 3.2 (TYPE-CORRECTNESS). *Let P be a program that is type-correct, let C and I be a class and an interface in P, respectively, such that C is the only class that implements I, and assume that I does not have any supertypes other than* `Object`*. Let P' be a program obtained from P by giving type I to all declaration elements in $All(P, C) \setminus Bad(P, C, I)$. Then, P' is type-correct.*

PROOF. In P', only declared types of variables and fields, method return types, and types referred to in casts are modified. Consequently, P and P' have identical class hierarchies, and there exists a one-to-one mapping between method, fields, types and expressions in program P and their counterparts in P'. In fact, the statements and expressions in P and P' are *exactly the same* except for: (i) casts that refer to type C in P, and to type I in P', and (ii) virtual method calls $E.m(\cdots)$ that refer to a method $C.m(\cdots)$ in P but to a method $I.m(\cdots)$ in P'. The latter situation occurs if the static type of receiver expression E is changed from C to I as a result of updating a declaration. In the proof, we use the mapping between statements/expressions in P and their counterparts in P' implicitly in checking whether or not P' is type-correct.

According to Definition 2.5, we have that $TC(P') = TC_{\text{fixed}}(P') \cup TC_{\text{var}}(P')$. Of these, the constraints in $TC_{\text{var}}(P')$ are constructed from declaration elements that occur in program P' and are always satisfied (because they *define* rather than *constrain*). Therefore, we only need to demonstrate that the constraints in $TC_{\text{fixed}}(P')$ are satisfied. Examination of rules (1)—(22) in Figure 3 reveals that for each constraint $t \in TC_{\text{fixed}}(P)$, one of the following situations applies:

1. Type constraint $t \in TC_{\text{fixed}}(P)$ was generated for a program construct that occurs unmodified in P'. Then, $t \in TC_{\text{fixed}}(P')$.

2. Type constraint $t \in TC_{\text{fixed}}(P)$ was generated for a cast $(C)E$, for which $[(C)E]_P = C$, but $[(C)E]_{P'} = I$. In this case, rule (16) does not apply in P', and $t \notin TC_{\text{fixed}}(P')$. Observe that no type constraint occurs in $TC_{\text{fixed}}(P')$ that "replaces" t.

3. Type constraint $t \in TC_{\text{fixed}}(P)$ was generated for a virtual method call $E.m(\cdots)$ (rule (4)), where the type of E is changed from C to I, and where $RootDefs(m(\cdots)) = \{ I.m(\cdots) \}$ or $RootDefs(m(\cdots)) = \{$ `Object`$.m(\cdots) \}$. Then, $t \in TC_{\text{fixed}}(P')$.

4. Type constraint $t \in TC_{\text{fixed}}(P)$ was generated for a virtual method call $E.m(\cdots)$ (rule (4)), where the type of E is changed from C to I, and where $RootDefs(m(\cdots))$ is not a singleton set (i.e., not case 3 above). In this case, it follows from Definitions 2.2 and 3.1 that $t \equiv t_1 \mathbf{or} \cdots \mathbf{or} t_k$ for some $k \geq 2$, and that $t_j = I$, for some $1 \leq j \leq k$. Moreover, for program P' we have that $RootDefs(I.m(\cdots)) = \{ I.m(\cdots) \}$ and hence that $t_j \equiv [E] \leq I \in TC_{\text{fixed}}(P')$. To summarize, for modified virtual method calls a type constraint $t \equiv t_1 \mathbf{or} \cdots \mathbf{or} t_k$ is replaced with one of its components t_j.

It should be noted that $TC_{\text{fixed}}(P')$ contains no constraint t that is not in $TC_{\text{fixed}}(P)$ (i.e., constraints inferred by rules (1)–(22)) except for the case where t is a replacement for another constraint in $TC_{\text{fixed}}(P)$ (see item 4 above). This is the case because P and P' have identical class hierarchies, and there exists a one-to-one mapping between program constructs in P and program constructs in P'.
We will demonstrate the type-correctness of P' by showing that, for every type constraint $t \in TC_{\text{fixed}}(P)$, the corresponding type constraint in $TC_{\text{fixed}}(P')$, if it exists, is satisfied by P'. The following cases exist[5]:

1. $t \equiv [E] \triangleq C$. This constraint is generated due to an application of one of the rules (2), (5), (13), and (19)–(22) These constraints merely define the type of an expression and cannot be violated.

2. $t \equiv [E] = [E']$. If $[E]_P = [E']_P \neq C$, we have $[E]_{P'} = [E]_P = [E']_P = [E']_{P'}$ because the transformation

[5]We will not discuss a number of constraints in $TC_{\text{fixed}}(P)$ that involve constant types (such as those involving the declaring classes of members). These remaining cases are similar to the ones presented, and are in most cases trivial.

only affects expressions and declaration elements of type C. Otherwise, $[E]_P = [E']_P = C$, and it follows from Definition 3.1 that either: (i) $E \in Bad(P, C, I)$ and $E' \in Bad(P, C, I)$, or (ii) $E \notin Bad(P, C, I)$ and $E' \notin Bad(P, C, I)$. In either case ($[E]_{P'} = C$, $[E']_{P'} = C$ or $[E]_{P'} = I$, $[E']_{P'} = I$), we have that: $[E]_{P'} = [E']_{P'}$.

3. $t \equiv [E] \leq D$ **or** $D \leq [E]$. Then, t occurs due to a cast $(D)E$ in P (rule (16)). There are two cases:
   (a) $[(D)E]_P \neq C$ or $(D)E \in Bad(P, C, I)$. Then, the cast cannot be updated, and $[(D)E]_{P'} = [(D)E]_P = D$. Two sub-cases exist: (i) If $E \notin All(P, C)$, then $[E]_{P'} = [E]_P$, and t holds because P is type-correct; (ii) $[E]_P = C$. If $E \in Bad(P, C, I)$, we have $[E]_{P'} = [E]_P$, and t holds because P is type-correct. If $E \notin Bad(P, C, I)$, we have that $[E]_{P'} = I$, and $t \notin TC_{\text{fixed}}(P')$ because rule (16) does not apply.
   (b) $[(D)E]_P = C$ and $(D)E \notin Bad(P, C, I)$. The cast occurs as $(I)E$ in P', and $t \notin TC_{\text{fixed}}(P')$ because rule (16) does not apply.

4. $t \equiv [E] \leq [E']$. We distinguish the following cases:
   (a) $E \notin All(P, C)$ and $E' \notin All(P, C)$. Then, $[E]_P = [E]_{P'}$ and $[E']_P = [E']_{P'}$. Hence, $[E]_{P'} \leq [E']_{P'}$.
   (b) $E \notin All(P, C)$ and $E' \in All(P, C)$. Then, $[E]_P = [E]_{P'}$. Two sub-cases exist: (i) if $E' \in Bad(P, C, I)$, then $[E']_{P'} = [E']_P$ and $[E]_{P'} \leq [E']_{P'}$; (ii) if $E' \notin Bad(P, C, I)$, then $C = [E']_P \leq [E']_{P'} = I$. Hence, $[E]_{P'} \leq [E']_{P'}$.
   (c) $E \in All(P, C)$ and $E' \notin All(P, C)$. Then, $[E']_{P'} = [E']_P$. Two sub-cases exist: (i) If $I \not\leq [E']_P$, then from Definition 3.1 it follows that $E \in Bad(P, C, I)$, so we have that $[E]_{P'} = [E]_P$, and therefore $[E]_{P'} \leq [E']_{P'}$; (ii) Otherwise, we have that $I \leq [E']_P$, so $[E]_{P'} \leq \text{I} \leq [E']_{P'}$.
   (d) $E \in All(P, C)$ and $E' \in All(P, C)$. Two sub-cases exist: (i) If $E' \in Bad(P, C, I)$, Definition 3.1 implies that $E \in Bad(P, C, I)$. Hence, $[E]_P = [E]_{P'} = [E']_P = [E']_{P'} = C$, and $[E]_{P'} \leq [E']_{P'}$; (ii) If $E' \notin Bad(P, C, I)$, then $[E'] = I$ in P'. Depending on whether or not $E \in Bad(P, C, I)$, we have $[E]_{P'} = C$ or $[E]_{P'} = I$, but in either case we have that $[E]_{P'} \leq [E']_{P'}$.

5. $t \equiv [E] \leq T$. Then, t is due to a virtual call $E.m(\cdots)$ to a method M, and T defines a method M_T that is overridden by M (rule (4)). Two cases exist:
   (a) $E \notin All(P, C)$ or $E \in Bad(P, C, I)$. Then, $[E]_{P'} = [E]_P$ and from rule (4) and Definition 2.2 it follows that $t \in TC_{\text{fixed}}(P')$. The fact that program P satisfies t implies that $[E]_P \leq T$, and from $[E]_{P'} = [E]_P \leq T$ it follows that P' satisfies t.
   (b) $E \in All(P, C)$ and $E \notin Bad(P, C, I)$. Then, $[E]_{P'} = I$. From Definition 2.2, it follows that $T =$ `Object` or $T = I$ (otherwise, Definition 2.2 would not have computed a singleton set), so we have 2 subcases: (i) if $T =$ `Object`, it follows from Definition 2.2 and rule (4) that $t \in TC_{\text{fixed}}(P')$, and P' trivially satisfies t because $[E]_{P'} = I \leq$ `Object` $= T$; (ii) if $T = I$, then it follows from Definition 2.2 and rule (4) that $t \in TC_{\text{fixed}}(P')$, and P' trivially satisfies t because $[E]_{P'} = I = T$.

6. $t \equiv [E] \leq T_1$ **or** $\cdots$ **or** $[E] \leq T_k$, $k \geq 2$. Then, t is due to a virtual call $E.m(\cdots)$ to a method M, and each type T_i ($1 \leq i \leq k$) defines a method M_i that is overridden by M (rule (4)). From the Definition 2.2 and the fact that $k \geq 2$, it can be seen that no method $m(\cdots)$ occurs in class `Object`. Two cases exist:
   (a) $E \notin All(P, C)$ or $E \in Bad(P, C, I)$. Then, $[E]_{P'} = [E]_P$ and from rule (4) and Definition 2.2 it follows that $t \in TC_{\text{fixed}}(P')$. The fact that program P satisfies t implies that $[E]_P \leq T_h$, for at least one h such that $1 \leq h \leq k$, and from $[E]_{P'} = [E]_P \leq T_h$, it follows that P' satisfies t.
   (b) $E \in All(P, C)$ and $E \notin Bad(P, C, I)$. Then, $[E]_{P'} = I$. From Definition 2.2, the fact that `Object` is the only supertype of I, and the fact that method $m(\cdots)$ is not defined in class `Object`, we find that a constraint $t' \equiv [E] \leq I \in TC_{\text{fixed}}(P')$ is generated for method call $E.m(\cdots)$ in program P' (this constraint $t' \in TC_{\text{fixed}}(P')$ replaces constraint $t \in TC_{\text{fixed}}(P)$). P' satisfies this constraint because $[E]_{P'} = I$.

Since all type constraints in $TC(P')$ are satisfied, P' is type-correct. □

Example. Let us consider type constraint $t \equiv$ `[v4]`$\leq$`[v5]` derived from the call to `Client.populate()` in `Client.main()`. Constraint t is satisfied in P_1 because $[\texttt{v4}]_{P_1} = [\texttt{v5}]_{P_1} =$ `List`. We also note that `v4` $\in Bad(P_1, List, Bag)$ and `v5` $\notin Bad(P_1, List, Bag)$. Thus, case 4d(ii) of the proof applies and we see that because $[\texttt{v4}]_{P_1'} =$ `List` and $[\texttt{v5}]_{P_1'} =$ `Bag` we have that $[\texttt{v4}]_{P_1'} \leq [\texttt{v5}]_{P_1'}$, so t is satisfied in P_1' as well.

Conjecture 3.3 (preservation of behavior). *Let P be a type-correct program, let C and I be a class and an interface in P, respectively, such that C is the only class that implements I and such that I does not have any supertypes other than* `Object`*. Let P' be a program obtained from P by giving type I to all declaration elements in $All(P, C) \setminus Bad(P, C, I)$. Then, P and P' have corresponding program behaviors.*

We plan to prove Conjecture 3.3 using the following arguments: (a) For a given expression E with run-time type T, a virtual call $E.m(\cdots)$ dispatches to the same method $B.m(\cdots)$ in P and P', even if $[E]_P = C$ and $[E]_{P'} = I$; (b) For a given expression E with run-time type T, a cast $(D)E$ succeeds/fails in exactly the same cases in P and P', even if $[(D)E]_P = C$ and $[(D)E]_{P'} = I$; (c) P and P' contain exactly the same statements and expressions. Together with (a) and (b), this implies that the same points-to relationships arise in P and P'.

Theorem 3.4 (minimality of $Bad(P, C, I)$). *Let P be a type-correct program, let C and I be a class and an interface in P, respectively, such that C is the only class that implements I and such that I does not have any supertypes other than* `Object`*. Let A be any set of declaration elements such that $A \subset Bad(P, C, I)$. Then, the program P' obtained from P by giving type I to all declaration elements in $(All(P, C) \setminus A)$ is type-incorrect.*

PROOF. The proof depends on the following auxiliary definition:

DEF. 3.5 (LAYER). *Let* $Layer(E) : Bad(P,C,I) \rightarrow \mathbb{N}$ *be defined as follows:*

$$Layer(E) = \begin{cases} 0 & \textbf{if } [E]{\leq}T_1 \textbf{ or } \cdots \textbf{ or } [E]{\leq}T_k \in \\ & TC_{\text{fixed}}(P),\ I \not\leq_P T_1, \cdots, I \not\leq_P T_k \\ 0 & \textbf{if } [E]{\leq}[E'] \in TC_{\text{fixed}}(P), \\ & E' \notin All(P,C), I \not\leq_P [E'] \\ n+1 & \textbf{if } Layer(E) \neq 0 \textit{ and} \\ & n = min(\{\, m = Layer(E') \mid E' \neq E, \\ & [E] = [E'] \in TC_{\text{fixed}}(P) \textit{ or} \\ & [E]{\leq}[E'] \in TC_{\text{fixed}}(P) \textit{ or} \\ & [E]{<}[E'] \in TC_{\text{fixed}}(P)\ \}) \end{cases}$$

Let $B = Bad(P,C,I) \setminus A$. Note that $B \neq \emptyset$ because $A \subset Bad(P,C,I) \subseteq All(P,C)$. We begin by selecting a "minimal" element $E \in B$ for which there exists no $E' \in B$ such that $Layer(E') < Layer(E)$. Note that, in cases where there is no unique minimal element, one may be chosen arbitrarily. We are assuming that all elements in B are given type I in P', hence $[E]_{P'} = I$. Two cases exist:

1. $Layer(E) = 0$. Then, from Definition 3.5, it follows that there are two sub-cases: (i) $[E]{\leq}T_1$ **or** $\cdots$ **or** $[E]{\leq}T_k \in TC_{\text{fixed}}(P)$, $I \not\leq T_1, \cdots, I \not\leq T_k$, and (ii) $[E]{\leq}[E'] \in TC_{\text{fixed}}(P)$, $E' \notin All(P,C)$, $I \not\leq [E']$. In each case, P' does not satisfy the constraint because of $[E]_{P'} = I$ and is therefore not type-correct.

2. $Layer(E) = n+1$, where $n \geq 0$. Then, there exists an $E' \neq E$ such that $Layer(E') = n$ and a type constraint $t \in TC_{\text{fixed}}(P)$ exists such that $t \equiv [E]{=}[E']$, or $t \equiv [E]{\leq}[E']$, or $t \equiv [E]{<}[E']$. We observe that $E' \notin B$ because $Layer(E') < Layer(E)$, and E was selected as one of B's elements with minimal *Layer*-value. From $E' \in A$, it follows that $[E']_{P'} = C$. From $[E]_{P'} = I$ and $[E']_{P'} = C$ it follows that program P' does not satisfy type constraint t, rendering P' type-incorrect.

□

4. OTHER REFACTORINGS RELATED TO GENERALIZATION

4.1 PULL UP MEMBERS

The purpose of PULL UP MEMBERS is to move member(s) from a given class into its superclass. We will use program P_2 of Figure 6 to illustrate the various issues associated with PULL UP MEMBERS. P_2 defines an abstract class `List` with two subclasses, `CList` for representing constant-length lists, and `FList` for variable-length lists. Methods are provided for retrieving the `size()` of a list, adding elements to `FList`s, sorting `FList`s, getting/setting a specific element, determining whether or not an `FList` `isEmpty()`, and for printing out the contents of a `CList` (`CList.toString()`). Careful examination of program P_2 reveals that:

1. Methods `get()`, `set()`, and `isEmpty()` can each be pulled up (by itself) from `FList` into `List` without affecting type-correctness and program behavior.

2. Method `size()` cannot be pulled up from `FList` into `List` because another[6] method with the same signature is already defined in `List`.

3. Method `FList.add()` can only be pulled up to `List` if `FList.set()` is pulled up as well, because no method `set()` is declared in class `List`.

4. Note that the body of `FList.sort()` contains a statement `return this`, and that the return type of `sort()` is `FList`. If `sort()` is pulled up into `List`, the type of `this` becomes `List`, and the resulting program becomes type-incorrect because the return expression is no longer (a subtype of) `FList`.

5. Pulling up method `CList.toString()` does not result in any compiler errors. However, program behavior has changed because the call `w1.toString()` now dispatches to a different definition of the `toString()` method.

4.2 Checking the Preconditions of PULL UP MEMBERS

We will reuse the type constraints of Figure 3 to detect when pulling up a (set of) method(s) would affect type-correctness or program behavior. However, there is a subtle difference in the way we use these constraints. In the case of EXTRACT INTERFACE, we were solving a constraint system in which the types of declaration elements were "variable", and the declaring classes of methods were "fixed". In the case of PULL UP MEMBERS, a different partitioning is required because the types of declaration elements are fixed (they are not affected by this particular refactoring), but the declaring classes of (pulled-up) members may change. Hence, in the case of PULL UP MEMBERS, we define $TC'_{\text{var}}(P)$ to be the set of constraints inferred using rule (21), and $TC'_{\text{fixed}}(P)$ to be the set of constraints inferred using any of the other rules.

Figure 7 shows the type constraints for P_2. Similar to what we did in Section 2.5, we simplify constraints by applying the "constant definition" constraints in $TC'_{\text{fixed}}(P)$ that use the $\triangleq$ symbol. Specifically, the type of a declaration element E is shown as the constant value $[E]_P$. However, the declaring class of a member M is shown in unsimplified form (i.e., as $Decl(M)$) because the PULL UP MEMBERS refactoring may change the location of members in the class hierarchy. Observe that:

- Pulling up `FList.size()` into `List` implies that *Decl*(`FList.size()`) = `List`. This would violate constraint *Decl*(`FList.size()`) < *Decl*(`List.size()`).

- It is obvious from constraint *Decl*(`FList.add()`) ≤ *Decl*(`FList.set()`) that `FList.add()` cannot be pulled up without also pulling up `FList.set()`.

- Pulling up method `FList.sort()` means that *Decl*(`FList.sort()`) = `List`, which violates the type constraint *Decl*(`FList.sort()`) ≤ `FList`.

- The remaining problem case—pulling up `CList.toString()`—does not raise any type-correctness issues. This illustrates that type constraints by

[6]This example contains two gratuitously different `size()` methods solely for the purpose of illustrating the issues raised by method overriding.

```
abstract class List {
  int size(){
    return this.size;
  }
  void setSize(int i){
    this.size = i;
  }
  Comparable[] elems;
  int size;
}
class CList extends List {
  CList(Comparable[] objects){
    this.elems = objects;
    this.size = objects.length;
  }
  public String toString() {
    return java.util.Arrays.asList(this.elems).toString();
  }
}
class FList extends List {
  FList(){
    this.elems = new Comparable[10];
    this.size = 0;
  }
  void add(Comparable e) {
    if (this.size() + 1 == this.elems.length){
      Comparable[] newObjects =
        new Comparable[2 * this.size()];
      System.arraycopy(elems,0,newObjects,0,this.size());
      this.elems = newObjects;
    }
    this.set(this.size(), e);
    this.setSize(this.size() + 1);
  }
  ···
```

```
  /* class FList continued */
  int size(){
    int n = this.size;
    return n;
  }
  boolean isEmpty(){
    return this.size() == 0;
  }
  public FList sort() { /* insertion sort */
    for (int i = 1; i < this.size; i++) {
      Comparable e1 = this.elems[i];
      int j = i;
      while ((j > 0) &&
          (this.elems[j-1].compareTo(e1) > 0)) {
        this.elems[j] = this.elems[j-1];
        j--;
      }
      this.elems[j] = e1;
    }
    return this;
  }
  Comparable get(int index){
    return this.elems[index];
  }
  void set(int index, Comparable o){
    this.elems[index] = o;
  }
}
class Client {
  public static void main(String[] args) {
    FList w1 = new FList();
    w1.add("foo"); w1.add("bar"); w1.sort();
    System.out.println(w1.toString());
    List w2 = new CList(new Comparable[]{ "zip", "zap" });
    System.out.println(w2.toString());
  }
}
```

Figure 6: Example program P_2. Class `List` represents abstract lists, and has two concrete subclasses: `CList` for constant lists, and `FList` for mutable lists.

themselves are not always sufficient to express the preconditions of refactorings.

Definition 4.1 below introduces a predicate $\mathit{CanPullUp}(P, M)$. If this predicate holds, virtual method M in program P can be pulled up into the superclass of M's declaring class without affecting type-correctness and program behavior. Referring to the labels (a)–(e) in Definition 4.1, we have that:

(a) Type constraints of the form $\mathit{Decl}(M) \leq [E]$ are due to assignments 'E = `this`' within M's body (as well as parameter-passing and return-expressions involving `this`). To ensure that these constraints still hold after pulling up M, we require that $\mathit{super}(\mathit{Decl}_P(M)) \leq [E]_P$.

(b) A constraint $\mathit{Decl}(M) \leq \mathit{Decl}(M')$ is due to (i) a member M' that is not a virtual method is accessed from method M's `this`-pointer, or (ii) a virtual method M'' is called on M's `this`-pointer, and $\mathit{RootDefs}(M'') = \{ M' \}$. To preserve such constraints, we require that $\mathit{super}(\mathit{Decl}_P(M)) \leq \mathit{Decl}_P(M')$.

(c) A constraint of the form $\mathit{Decl}(M) \leq \mathit{Decl}(M_1)$ **or** $\cdots$ **or** $\mathit{Decl}(M) \leq \mathit{Decl}(M_k)$ arises due to a virtual method call to M' on the `this` pointer of method M, where $\mathit{RootDefs}(M') = \{ M_1, \cdots, M_k \}$. After pulling up method M, the constraint still holds if $\mathit{super}(\mathit{Decl}_P(M)) \leq \mathit{Decl}_P(M_j)$, for some $1 < j \leq k$.

(d) Constraints of the form $\mathit{Decl}(M) < \mathit{Decl}(M')$ occur when a method M overrides method M'. Requiring $\mathit{super}(\mathit{Decl}_P(M)) < \mathit{Decl}_P(M')$ ensures that no class contains methods with identical signatures after the pull-up.

(e) The final condition in Definition 4.1 suffices to preserve dispatch behavior of calls to methods with the same signature as M, and states that, if a method with the same signature as M is called on a subtype of $\mathit{super}(\mathit{Decl}_P(M))$, the dispatch should resolve to a proper subtype of $\mathit{super}(\mathit{Decl}_P(M))$. Here, $\mathit{staticLookup}(P, C, S)$ is a function that, for a given program P and class C, determines the nearest superclass of C that declares a method with signature S.

It should be stated that (d) and (e) are *sufficient* conditions that, in some cases, may prohibit pulling up methods when it is safe to do so. In case of (e) this is unavoidable, because a precise condition would requires full knowledge about program execution behavior. Work on an exact version of (d) is still in progress.

method(s)	constraint(s)	rule(s)
`List.size()`, `FList.size()`	*Decl*(`FList.size()`) < *Decl*(`List.size()`)	(10)
`List.setSize()`	*Decl*(`List.setSize()`) ≤ *Decl*(`List.size`)	(6),(19)
`List.size()`	*Decl*(`List.size()`) ≤ *Decl*(`List.size`)	(6),(19)
`CList.CList()`	*Decl*(`CList.CList()`) ≤*Decl*(`List.elems`)	(6),(19)
	Decl(`CList.CList()`) ≤*Decl*(`List.size`)	(6),(19)
`CList.toString()`	*Decl*(`CList.toString()`) ≤*Decl*(`List.elems`)	(6),(19)
`FList.FList()`	*Decl*(`FList.FList()`) ≤ *Decl*(`List.size`)	(6),(19)
	Decl(`FList.FList()`) ≤ *Decl*(`List.elems`)	(6),(19)
`FList.add()`	*Decl*(`FList.add()`) ≤ *Decl*(`List.size()`)	(4),(19)
	Decl(`FList.add()`) ≤ *Decl*(`List.elems`)	(6),(19)
	Decl(`FList.add()`) ≤ *Decl*(`List.setSize()`)	(4),(19)
	Decl(`FList.add()`) ≤ *Decl*(`FList.set()`)	(4),(19)
`FList.size()`	*Decl*(`FList.size()`) ≤ *Decl*(`List.size`)	(6),(19)
`FList.isEmpty()`	*Decl*(`FList.isEmpty()`) ≤ *Decl*(`List.size()`)	(4),(19)
`FList.sort()`	*Decl*(`FList.sort()`) ≤ *Decl*(`List.size()`)	(4),(19)
	Decl(`FList.sort()`) ≤ *Decl*(`List.elems`)	(6),(19)
	Decl(`FList.sort()`) ≤ `FList`	(7),(19)
`FList.get()`	*Decl*(`FList.get()`) ≤ *Decl*(`List.elems`)	(6),(19)
`FList.set()`	*Decl*(`FList.set()`) ≤ *Decl*(`List.elems`)	(6),(19)
`Client.main()`	`FList` ≤ *Decl*(`FList.add()`)	(4),(23)
	`FList` ≤ *Decl*(`FList.sort()`)	(4),(23)

Figure 7: Type constraints $TC_{\text{fixed}}(P_2)$ for program P_2 of Figure 6. Only nontrivial constraints related to types `List`, `CList`, and `FList` are shown.

DEF. 4.1. *Let P be a program, let M be a virtual method in P such that $Decl(M) \neq$ `Object`, and let $Decl(M)$ be a class. Define:*

$$
\begin{aligned}
&\mathit{CanPullUp}(P, M) \Leftrightarrow \\
&\quad \forall \mathit{Decl}(M) \leq [E] \in TC'_{\text{fixed}}(P) : \\
&\qquad \mathit{super}(\mathit{Decl}_P(M)) \leq_P [E]_P, \mathit{and} && (a) \\
&\quad \forall \mathit{Decl}(M) \leq \mathit{Decl}(M') \in TC'_{\text{fixed}}(P),\ M \neq M' : \\
&\qquad \mathit{super}(\mathit{Decl}_P(M)) \leq_P \mathit{Decl}_P(M'), \mathit{and} && (b) \\
&\quad \forall \mathit{Decl}(M) \leq \mathit{Decl}(M_1) \textbf{ or } \cdots \textbf{ or} \\
&\qquad \mathit{Decl}(M) \leq \mathit{Decl}(M_k) \in TC'_{\text{fixed}}(P), k > 1 : \\
&\qquad \mathit{super}(\mathit{Decl}_P(M)) \leq_P \mathit{Decl}_P(M_j), \\
&\qquad \mathit{for\ some}\ j, 1 < j \leq k, \mathit{and} && (c) \\
&\quad \forall \mathit{Decl}(M) < \mathit{Decl}(M') \in TC'_{\text{fixed}}(P) : \\
&\qquad \mathit{super}(\mathit{Decl}_P(M)) <_P \mathit{Decl}_P(M'), \mathit{and} && (d) \\
&\quad \forall\, C \leq \mathit{super}(\mathit{Decl}_P(M)) : \\
&\qquad \mathit{staticLookup}(P, C, \mathit{Sig}(M)) <_P \mathit{super}(\mathit{Decl}_P(M)) && (e)
\end{aligned}
$$

It is straightforward to extend *CanPullUp* to nonvirtual methods, member types, fields and to sets of members (methods, types and fields). In addition, type constraints can be used to determine, for a given method M that cannot be pulled up in isolation, if there exists a set of methods containing M that can be pulled up. Space limitations prevent us from providing additional details.

4.3 Other Refactorings

Other refactorings that can be modeled using our approach include:

GENERALIZE TYPE. This refactoring replaces the type of a *single* declaration element E with its supertype. The preconditions for this refactoring can be stated as a predicate on the type constraints that involve E. In some cases, GENERALIZE TYPE can enable PULL UP MEMBERS refactorings that are otherwise impossible. If, in program P_2 of Figure 6, the return type of `FList.sort()` is generalized to `List`, `FList.sort()` can be pulled up into class `List`.

EXTRACT SUBCLASS [6, page 330]. This is a refactoring for "splitting" a class, to address situations where a class has members that are used in some instances of the class and not in others. This raises issues related to the updating of declaration elements similar to those discussed for EXTRACT INTERFACE.

PUSH DOWN MEMBERS [6, page 328]. This is the inverse operation of PULL UP MEMBERS, and although technically a refactoring for *specialization*, it raises very similar issues.

5. IMPLEMENTATION AND PRAGMATIC ISSUES

We have implemented EXTRACT INTERFACE in Eclipse [5]. Our implementation follows Definition 3.1 to find declaration elements that can be updated, and uses the standard model and GUI support for performing refactorings in Eclipse [1]. Work on the other refactorings of Section 4 is in progress.

Thus far, we have only shown type constraints for a core set of Java features. A number of Java's other features are discussed below. Some of the more interesting corresponding type constraints are shown in Figure 8.

Arrays. Arrays and array initializers introduce some rather straightforward constraints, as shown in Figure 8. Additionally, concrete refactorings need to take array types into account as well. E.g., in the case of EXTRACT INTERFACE, the computation of $Bad(P, C, I)$ must take into account expressions and declaration elements of type $C[]$, $C[][]$, $\cdots$, etc. Moreover, it would be desirable to determine those declaration elements of type $C[]$, $C[][]$, $\cdots$ that can be updated to $I[]$, $I[][]$, $\cdots$, respectively. A precise solution to this problem is a topic for future work.

Casts and `instanceof` expressions. Type constraints implied by `instanceof` expressions (rule (38)) are anal-

program construct	implied type constraint(s)	
`new` $T[E]$	$[\texttt{new}\ T[E]] \triangleq T[\,]$	(27)
	$[E] = \textsc{Int}$	(28)
expression $T[E]$	$[T[E]] \triangleq T$	(29)
	$[E] = \textsc{Int}$	(30)
for any $T \leq T'$	$T[\,] \leq T'[\,]$	(31)
initialized declaration $T[\,]\ c = \{E_1, \ldots, E_n\}$	$[E_i] \leq T$	(32)
initialized creation expression `new` $T[E]\{E_1, \ldots, E_n\}$	$[E_i] \leq T$	(33)
`try` $\{\ldots\}$ `catch`$(E\ e)\{\ldots\}$	$E \leq$ `java.lang.Throwable`	(34)
`try`$\{\ldots\}$ `catch`$(E_1\ e_1)\{\ldots\}$ … `catch`$(E_n\ e_n)\{\ldots\}$	$\forall i, j : 1 \leq i < j \leq n$ $E_j \not\leq E_i$	(35)
M overrides M'	$\forall E \in \mathit{Excptns}(M) \cap \mathit{CheckedExcptns}$ $\exists E' \in \mathit{Excptns}(M')$ $E \leq E'$	(36)
expression E `instanceof` T	$[\text{E } \texttt{instanceof } \text{T}] \triangleq \textsc{Bool}$	(37)
	$[E] \leq T$ **or** $T \leq [E]$ if T is a class and $[E]$ is a class	(38)

$\mathit{CheckedExcptns}$ $\{E : \text{E} \leq$ `java.lang.Throwable` $\wedge\ \text{E} \not\leq$ `java.lang.Error` $\wedge$ $E \not\leq$ `java.lang.RuntimeException`$\}$

$\mathit{Excptns}(M)$ set of exceptions included in `throws` clause of M's declaration

Figure 8: Additional type constraints.

ogous to those for casts (rule (16)) [10, Section 15.20.2]. In addition to rule (16), various other constraints on the correctness of casts exist. For a detailed discussion, we refer the reader to [10, Section 5].

Member types. Member types [10, Section 8.5] have access to fields, methods, types and variables declared in their enclosing scopes and supertypes. A member type is type-incorrect if it uses an identifier that is declared in both a supertype and an enclosing scope. Care must be taken when applying refactorings like Pull Up Members to avoid introducing such ambiguities.

Exceptions. Several additional type constraints related to exceptions are listed in Figure 8 as rules (34)–(36). Replacing the type referred to in a **catch**-clause with its supertype may change program behavior in various ways (e.g., more exceptions being caught, or the program becoming type-incorrect). In our implementation, we exclude `java.lang.Throwable` and its subclasses from consideration, similarly to [11].

Overloading. Overloading (i.e., having methods with identical names but different argument types in a class) raises interesting issues for the refactorings under consideration. An invocation of $m(T_1, \cdots, T_k)$ results in selection of a method $m(\cdots)$ with the most specific signature that matches $m(T_1, \cdots, T_k)$ [10, Section 15.12.2.2]. Changing parameter types, or pulling up methods may affect this specificity ordering, and change program behavior. Currently, we disallow refactorings when such problems occur.

Visibility/Accessibility issues. Care must be taken to preserve the appropriate visibility relationships. E.g., extracting an interface may require adding `import` statements, to ensure the visibility of parameter types. A refactoring tool must either increase visibility of these members, or disallow refactoring.

6. RELATED WORK

In his pioneering work on refactoring, Opdyke [15] identified and informally specified invariants that any refactoring must preserve [15, page 27–28]. One of these invariants, *Compatible Signatures in Member Function Redefinition*, states that overriding methods must have corresponding argument types and return types, and is reflected by 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 is expressed by our constraints (1), (3), (12), and (14). Opdyke states the preconditions of refactorings as requirements on source-level constructs, using a number of auxiliary predicates that represent structural properties of programs. Opdyke does not address the issue of using invariants to compute a set of allowable source code modifications, as we do in the case of Extract Interface.

Fowler [6] 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 related to generalization-related refactorings are not addressed. For example, 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.

The development environment *IntelliJ IDEA* [12] by JetBrains, Inc. supports refactorings that deal with generalization such as Extract Interface, and automatically determines declaration elements that can be updated. We are unfamiliar with the details of their implementation, but the results obtained with their tool appear similar to ours.

Halloran and Scherlis [11] present an algorithm for detecting overspecific variable declarations. In contrast to our work, every variable declaration is analyzed in isolation, and relationships between declarations—all-important in our approach—are not considered. Hence, several run/modify iterations may be required to discover all possible declaration updates.

Duggan [4] studies the problem of reverse engineering parameterized types into existing Java programs. In this work, type constraints are used to infer bounds on type parameters of parameterized classes, and to determine situations in which downcasts are guaranteed to succeed. Duggan's formal model of type constraints is similar in spirit to ours, although notationally quite different. Some important differences include our treatment of virtual method calls and overriding, and the fact that in our constraints, the location of methods and fields is variable.

Tokuda and Batory [22] discuss the use of refactorings to introduce new (or evolve existing) design patterns [8] in applications. Several refactorings are presented to support this evolution of program designs, 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 proper precondition checking for SUBSTITUTE.

Seguin [18] analyzes, using PULL UP FIELD as an example, challenges for refactoring in strongly typed languages such as Java. The issues encountered by Seguin are similar to those described in this paper. Seguin, however, advocates the use of type casts to preserve the program's type-correctness, a solution that we consider inappropriate in a refactoring tool (see Section 1.3).

Snelting and Tip [19, 20] present an approach for generating refactoring proposals for Java applications (e.g., indications that a class can be split, or that a member can be moved). This work is based on earlier work by Tip and Sweeney [21] in which 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.

Much previous work on generating and solving type constraints exists (see, e.g., [17, 7]). Recently, Glew and Palsberg [9] designed an algorithm for type-safe method inlining that makes use of type constraints to refine type-annotations. Carlos [3] describes an algorithm that analyzes a program's type constraints to identify all declaration elements that need to be updated in response to a request for updating a given declaration element. This algorithm is only described informally, and several important language features are not addressed (e.g., casts). No proof of correctness or optimality of the computed solution is given.

7. FUTURE WORK

Plans for future work include a complexity analysis of our algorithms, and a detailed study of other generalization-related refactorings (see Section 4.3). With respect to EXTRACT INTERFACE, we plan to extend the declaration-updating process to include declaration elements whose type is an array, and declaration elements whose type is a subtype of the class from which the interface is extracted. A complete formalization of the type constraints associated with overloading is also future work.

Acknowledgments

We are grateful to Jens Palsberg and V.T. Rajan for comments on drafts of this paper.

8. REFERENCES

[1] BÄUMER, D., GAMMA, E., AND KIEŻUN, A. Integrating refactoring support into a Java development tool. In *OOPSLA'01 Companion* (October 2001).

[2] BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[3] CARLOS, C. S. The elimination of overheads due to type annotations and the identification of candidate refactorings. Master's thesis, North Carolina State University, 2002.

[4] DUGGAN, D. Modular type-based reverse engineering of parameterized types in java code. In *Proceedings of the 14th Annual Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA'99)* (Denver, CO, November 1999), pp. 97–113.

[5] ECLIPSE.ORG. Eclipse. On-line at http://www.eclipse.org.

[6] FOWLER, M. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] GAGNON, E. M., HENDREN, L. J., AND MARCEAU, G. Efficient inference of static types for Java bytecode. In *Proceedings of SAS'00, International Static Analysis Symposium* (2000), Springer-Verlag (*LNCS* 1824), pp. 199–219.

[8] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] GLEW, N., AND PALSBERG, J. Type-safe method inlining. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)* (Málaga, Spain, June 2002), pp. 525–544. Springer-Verlag LNCS 2374.

[10] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification (Second Edition)*. Addison-Wesley, 2000.

[11] HALLORAN, T. J., AND 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, Sept. 2002.

[12] JETBRAINS, INC. Intellij IDEA. On-line at http://www.intellij.com/jetbrains.

[13] MEYER, B. *Object-Oriented Software Construction*. Prentice Hall, Inc., 1997.

[14] OPDYKE, W. F., AND JOHNSON, R. E. Creating abstract superclasses by refactoring. In *The ACM 1993 Computer Science Conf. (CSC'93)* (February 1993), pp. 66–73.

[15] OPDYKE, W. F. *Refactoring Object-Oriented Frameworks*. PhD thesis, University Of Illinois at Urbana-Champaign, 1992.

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

[17] Palsberg, J. Efficient inference of object types. *Information and Computation 123*, 2 (1995), 198–209.

[18] Seguin, C. Refactoring tool challanges in a strongly typed language. In *OOPSLA'00 Companion* (October 2000), pp. 101–102.

[19] Snelting, G., and Tip, F. Reengineering class hierarchies using concept analysis. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Orlando, FL, November 1998), pp. 99–110.

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

[21] Tip, F., and Sweeney, P. Class hierarchy specialization. *Acta Informatica 36* (2000), 927–982.

[22] Tokuda, L., and Batory, D. Evolving object-oriented designs with refactorings. *Kluwer Journal of Automated Software Engineering* (August 2001), 89–120.