RC22662 (W0212-027) December 4, 2002
Computer Science

IBM Research Report

Refactoring for Generalization using Type Constraints

Frank Tip
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Adam Kiezun, Dirk Baeumer
2 Object Technology International AG
Oberdorfstrasse 8,CH-8001 Z urich,Switzerland

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

Refactoring for Generalization using Type Constraints

Frank Tip[1] and Adam Kieżun[2] and Dirk Bäumer[2]

[1] IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
tip@watson.ibm.com
[2] Object Technology International AG
Oberdorfstrasse 8, CH-8001 Zürich, Switzerland
{adam_kiezun,dirk_baeumer}@oti.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 depends 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 (available from www.eclipse.org).

1 Introduction

Refactoring [5, 13] 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 [4] and IntelliJ [10] incorporate refactoring capabilities.

An important category of refactorings is concerned with generalization [5, Chapter 11][12], 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 involves updating declarations of variables, parameters, return types, and fields to use

the newly added interface. Not updating these declarations leads to overspecific variables, which conflicts with the principles of object-oriented design [11].

This paper proposes the use of an existing framework of *type constraints* [14] 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 and program behavior. 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 [4], 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 `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 references to it. 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, `l0`, `l1`, `l2`, `l3`, `l4`, `l5`, `l6`, `l7`, `l8` and `l9`, 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 `l2` cannot be declared as type `Bag` because the fields `size` and `elems` are accessed from `l2`, but not declared in `Bag`.
- Variable `l3` is assigned to `l2`, requiring that the declared type of `l3` be equal to or a subtype of the declared type of `l2`. As the declaration of `l2` cannot be updated, the declaration of `l3` cannot be updated either.
- The declared type of `l8` must remain `List` because `sort()` is called on `l8`, and `sort()` is not declared in `Bag`.
- `l4` is passed as an argument to `Client.sortList()`, implying an assignment `l8 = l4`. Hence, `l4`'s declared type must be equal to, or a subtype of `l8`'s declared type, which cannot be changed. So, `l4`'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 `l4`, whose declared type must remain `List`, as was discussed above.

```
interface Bag {
  public java.util.Iterator iterator();
  public [List] add(Object element);
  public [List] addAll([List] l0);
}
class List implements Bag {
  int size = 0; Object[] elems = new Object[10];
  public java.util.Iterator iterator(){ return new Iterator(this); }
  public [List] add(Object e){
    if (this.size+1 == this.elems.length){
      Object[] newObjects = new Object[2 * this.size];
      System.arraycopy(this.elems, 0, newObjects, 0, this.size);
      this.elems = newObjects;
    }
    this.elems[this.size++] = e; return this;
  }
  public [List] addAll([List] l1) { java.util.Iterator i=l1.iterator();
    for( ; i.hasNext(); this.add(i.next())); return this;
    }
    public void sort(){
      for (int t = 0; t < this.size; t++){
        for (int u = t+1; u < this.size; u++){
          Object e1=this.elems[t];Comparable e2=(Comparable)this.elems[u];
          if (e2.compareTo(e1)<0){ this.elems[t]=e2; this.elems[u]=e1;}
  } } } }
  class Iterator implements java.util.Iterator {
    private int count = 0; private List l2;
    Iterator(List l3){ this.l2 = l3;}
    public boolean hasNext(){ return this.count < this.l2.size; }
    public Object next(){ return this.l2.elems[this.count++]; }
    public void remove(){ throw new UnsupportedOperationException(); }
  }
  class Client {
    public static void main(String[] args){
      List l4 = createList();
      populate(l4); update(l4); sortList(l4); print(l4);
    }
    static List createList(){ return new List();}
    static void populate([List] l5){ l5.add("foo").add("bar");}
    static void update([List] l6) {
      [List] l7 = new List().add("zap").add("baz"); l6.addAll(l7);
    }
    static void sortList(List l8) { l8.sort(); }
    static void print([List] l9) {
      for (java.util.Iterator iter = l9.iterator(); iter.hasNext(); )
        System.out.println("Object: " + iter.next());
  } }
```

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

To summarize our findings, only `l0`, `l1`, `l5`, `l6`, `l7`, `l9` 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, and its use in modeling Extract Interface. Section 3 studies other refactorings that can be accommodated using this model. Section 4 presents implementation issues related to Java features not previously discussed. Sections 5 and 6 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, `l4`, `l8`, 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.

2 Formal Model

Palsberg and Schwarzbach [14] 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 run-time (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 [14] 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 use 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, we have, using method names to represent method return types:

$$All(P_1, \texttt{List}) = \{\ \texttt{l0}, \texttt{l1}, \texttt{l2}, \texttt{l3}, \texttt{l4}, \texttt{l5}, \texttt{l6}, \texttt{l7}, \texttt{l8}, \texttt{l9}, \texttt{Bag.add}(), \texttt{Bag.addAll}(), \texttt{List.add}(), \texttt{List.addAll}(), \texttt{Client.createList}()\ \}$$

In what follows, v, v' denote variables, M, M' denote methods, F, F' denote fields, C, C' denote types[1], and E, E' denote expressions and declaration elements. The following notation will be used to express type constraints:

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

$[E]_P$	the type of expression E in program P
$[M]_P$	the declared return type of method M in program P
$[F]_P$	the declared type of field F in program P
$Decl_P(M)$	the type that contains method M in program P
$Decl_P(F)$	the type that contains field F in program P
$Param(M, i)$	the i-th formal parameter of method M
$C' \leq C$	C' is equal to C, or C' is a subtype of C
$C' < C$	C' is a proper subtype of C (i.e., $C' \leq C$ and $C' \neq C$)
$super(C)$	the superclass of class C

We will frequently omit the P-subscript of $[E]_P$, $Decl_P(M)$, and $Decl_P(F)$ where P is unambiguous, and simply write $[E]$, $Decl(M)$, and $Decl(F)$, respectively.

A method M is *virtual* if M is not a constructor, M is not private and M is not static. Definitions 1 and 2 below define concepts of *overriding*[2] and *root definitions* for virtual methods. Definition 3 defines a notion of *hiding* for fields that will be needed for the Pull Up Members refactoring in Section 3.

Definition 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 $C \leq B$. In this case, we also say that M' is overridden by M.*

Definition 2 (RootDefs). *Let M be a method. Define:*

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

Definition 3 (hiding). *Field F in type C hides field F' in type B if F and F' have identical names and $C < B$. Then, we also say that F' is hidden by F.*

2.2 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), or $Decl(M)$ (representing the type in which member M is declared). In this paper, a *type constraint* has one of the following forms: (i) $\alpha_1 = \alpha_2$, indicating that α_1 and α_2 must be the same type, (ii) $\alpha_1 \leq \alpha_2$, indicating that α_1 must be equal to or a subtype of α_2, (iii) $\alpha_1 < \alpha_2$, indicating that $\alpha_1 \leq \alpha_2$ but not $\alpha_1 = \alpha_2$, (iv) $\alpha_1^L \leq \alpha_1^R$ **or** $\cdots$ **or** $\alpha_1^L \leq \alpha_k^R$, indicating that $\alpha_j^L \leq \alpha_j^R$ must hold for at least one j, $1 \leq j \leq k$.

Figure 2 lists the type constraints implied by a number of common Java features, which were carefully designed to reflect the semantics of Java [8]. 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

[2] Note that, according to Definition 1, a virtual method overrides itself.

[3] Rules (17) and (18) in Figure 2 are only shown for completeness, and are not affected by the refactorings we consider.

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	$[E.m(E_1, \cdots, E_n)] = [M]$	(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	$[E.f] = [F]$	(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	$[E.m(E_1, \cdots, E_n)] = [M]$	(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 C is a class and $[E]$ is a class	(16)
for every type C	$C \leq$ `java.lang.Object`	(17)
	`[null]` $\leq$ `C`	(18)
implicit declaration of `this` in method M	`[this]` $= Decl(M)$	(19)
expression `new` $C(E_1, \cdots, E_n)$	`[new` $C(E_1, \cdots, E_n)] = C$	(20)
type-name expression C	$[C] = C$	(21)
explicit declaration $C\ v$	$[v] = C$	(22)
declaration of method M with return type C	$[M] = C$	(23)
declaration of field F with type C	$[F] = C$	(24)
cast $(C)E$	$[(C)E] = C$	(25)

Fig. 2. Type constraints for a set of core Java language features. Rules (1)–(21) define the types of expressions and impose constraints between the types of expressions and declaration elements. Rules (22)–(25) define the types of declaration elements.

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, and requiring $[E]$ to be a subtype of one or more of $\mathit{Decl}(M_1), \cdots, \mathit{Decl}(M_k)$.

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 3 to check for cases where methods cannot be moved.

For a cast from type C to type D, an ordering relationship between C and D in the class hierarchy is required (rule (16)). This constraint only applies if both C and D are classes [8, Section 5.5]. Rules (22)–(25) 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 2 (in particular, (8)–(11)) go beyond the type-checking that is routinely performed by Java compilers. These rules are needed to ensure preservation of program behavior.

Definition 4 ($TC_{\text{fixed}}(P)$, $TC_{\text{decl}}(P)$, $TC(P)$). *Let P be a program. Then, $TC_{\text{fixed}}(P)$ denotes the set of type constraints inferred for program P according to rules (1)–(21). Further, $TC_{\text{decl}}(P)$ is the set of constraints inferred for P according to rules (22)–(25). Moreover, $TC(P)$ denotes the set $TC_{\text{fixed}}(P) \cup TC_{\text{decl}}(P)$.*

Our use of type constraints deserves a few additional comments. First, a type constraint $[E] \leq [E'] \in TC(P)$ states that the type-correctness of program P requires that $[E]_P \leq [E']_P$ (similar statements can be made about the other forms of type constraints). We say that a program P *satisfies* a type constraint $[E] \leq [E']$ if $[E]_P \leq [E']_P$ (again, similar cases exist for the other forms of type constraints), and that P is *type-correct* if all constraints in $TC(P)$ are satisfied.

2.3 Type constraints for program P_1

Figure 3 shows the type constraints in $TC_{\text{fixed}}(P_1)$ related to types `List` and `Bag`. Here, each expression $\mathit{Decl}(M)$ was reduced to the (constant) class in which M is declared. We can perform this simplification when considering Extract Interface because this refactoring does not affect the classes in which members are declared. Let us consider the derivation of some of these constraints:

(a) For the call to `List.add()` on receiver expression `l5` in method `Client.populate()`, we find according to rule (2) that [`l5.add("foo")`] = [

[4] Rules (2), (5), (13), (20), and (21) *define* the type of certain kinds of expressions. While these rules are not very interesting by themselves, they are essential for defining the relationships between the types of expressions and declaration elements.

`List.add()`], and using rule (4) that [`l5`] $\leq$ *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: [`l5.add("foo").add("bar")`] = [`List.add()`] and [`l5.add("foo")`] $\leq$ *Decl*(`Bag.add()) = Bag`. Combining this with (a) yields [`List.add()`] $\leq$ `Bag`.

(c) For each field access `this.size` in method `List.sort()`, we find using rules (19) and (6) that `List` = *Decl*(`List.sort()`) $\leq$ *Decl*(`List.size) = 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()`] $\leq$ [`l4`]. Applications of rule (14) yield [`l4`] $\leq$ [`l5`], [`l4`] $\leq$ [`l6`], [`l4`] $\leq$ [`l8`], and [`l4`] $\leq$ [`l9`].

method(s)	**constraint(s)**	**rule(s)**
`List.add(),Bag.add()`	[`Bag.add()`] = [`List.add()`]	(9)
`List.addAll()`, `Bag.addAll()`	[`l0`] = [`l1`] [`Bag.addAll()`] = [`List.addAll()`]	(8) (9)
`List.iterator()`	`List` $\leq$ [`l3`]	(19), (12)
`List.add()`	`List` $\leq$ [`List.add()`]	(7), (19)
`List.addAll()`	[`l1`] $\leq$ `Bag` `List` $\leq$ [`List.addAll()`]	(4) (19), (7)
`Iterator.Iterator()`	[`l3`] $\leq$ [`l2`]	(1)
`Iterator.hasNext()`	[`l2`] $\leq$ `List`	(6)
`Iterator.next()`	[`l2`] $\leq$ `List`	(6)
`Client.main()`	[`Client.createList()`] $\leq$ [`l4`] [`l4`] $\leq$ [`l5`]; [`l4`] $\leq$ [`l6`] [`l4`] $\leq$ [`l8`]; [`l4`] $\leq$ [`l9`]	(1), (13) (14);(14) (14);(14)
`Client.createList()`	`List` $\leq$ [`Client.createList()`]	(20), (7)
`Client.populate()`	[`l5`] $\leq$ `Bag` [`List.add()`] $\leq$ `Bag`	(4) (2), (4)
`Client.update()`	[`List.add()`] $\leq$ [`l7`] [`List.add()`] $\leq$ `Bag` [`l6`] $\leq$ `Bag`; [`l7`] $\leq$ [`l1`]	(1) (2), (4) (4);(3)
`Client.sortList()`	[`l8`] $\leq$ `List`	(4)
`Client.print()`	[`l9`] $\leq$ `Bag`	(4)

Fig. 3. 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.

2.4 Determining declarations that can be changed

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

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

and such that all constraints in $TC(P_1')$ are satisfied. A naive approach to solve this problem would be to check for each possible value of G, if it satisfies 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 can be updated. For example, from Figure 3 it can be seen that `List` $\leq$ [`l3`] $\leq$ [`l2`] $\leq$ `List`, which implies that `l2` and `l3` can only have type `List`. Definition 5 below formalizes this notion of "non-updatability".

Definition 5 (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:*

$$
\begin{aligned}
&Bad(P, C, I) = \\
&\quad \{\, E \mid E \in All(P, C),\ [E] \leq C_1 \textbf{ or } \cdots \textbf{ or } [E] \leq C_k \in TC_{\text{fixed}}(P), \\
&\qquad I \not\leq C_1, \cdots, I \not\leq C_k \,\} \cup \\
&\quad \{\, E \mid E \in All(P, C),\ [E] \leq [E'] \in TC_{\text{fixed}}(P),\ E' \notin All(P, C), I \not\leq [E'] \,\} \cup \\
&\quad \{\, E \mid E \in All(P, C),\ [E] = [E'] \in TC_{\text{fixed}}(P) \textit{ or } [E] \leq [E'] \in TC_{\text{fixed}}(P) \textit{ or } \\
&\qquad [E] < [E'] \in TC_{\text{fixed}}(P),\ E' \in Bad(P, C, I) \,\}
\end{aligned}
$$

The first part of Definition 5 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. The second part of Definition 5 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`). The third part handles the propagation of "badness" due to overriding, assignments, and parameter passing. For program P_1 of Figure 1, we have:

$$Bad(P_1, \texttt{List}, \texttt{Bag}) = \{\, \texttt{l2}, \texttt{l3}, \texttt{l4}, \texttt{l8}, \texttt{Client.createList}() \,\}$$

Hence, the declarations of `l0`, `l1`, `l5`, `l6`, `l7`, and `l9` 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 5 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 4 illustrates.

For the code in the first loop of Figure 4, 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()`.

```
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());
  }
}
```

Fig. 4. Example method that makes use of type casts (this method is assumed to occur in class `Client` of Figure 1).

2.5 Justification

Theorem 1 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 1 (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. P and P' have identical class hierarchies, and contain the same expressions and statements. Examination of rules (1)—(20) in Figure 2 reveals that $TC(P) \supseteq TC(P')$. (The only case where a constraint $t \in TC(P)$ does not occur in $TC(P')$ is when t is due to a cast $(C)E$, for which $[(C)E)]_P = C$, and $[(C)E)]_{P'} = I$). We need to show for each constraint $t \in TC(P)$ that either t is satisfied by P' or $t \notin TC(P')$. The following cases exist[5]:

1. $t \equiv [E] = C$. This constraint is generated due to an application of rule (19), (20), or (21) to define the type of a `this`-expression, a `new`-expression, or a static type expression, respectively. These constraints merely define the type of an expression and cannot be violated.

[5] We will only examine the cases that involve type constraints in which each "side" represents the type of a variable, parameter, field, or method return type. Constraints that involve constant types (such as those involving the declaring classes of members) can be dealt with similarly, and are in most cases trivial.

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 only affects expressions and declaration elements of type C. Otherwise, $[E]_P = [E']_P = C$, and it follows from Definition 5 that either: (i) $E \in \mathit{Bad}(P, C, I)$ and $E' \in \mathit{Bad}(P, C, I)$, or (ii) $E \notin \mathit{Bad}(P, C, I)$ and $E' \notin \mathit{Bad}(P, C, I)$. 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 $(D)E \in \mathit{Bad}(P, C, I)$. Then, the cast cannot be updated, and $[(D)E]_{P'} = [(D)E]_P = D$. Two sub-cases exist: (i) If $E \notin \mathit{All}(P, C)$, then $[E]_{P'} = [E]_P$, and t holds because P is type-correct; (ii) $[E]_P = C$. If $E \in \mathit{Bad}(P, C, I)$, we have $[E]_{P'} = [E]_P$, and t holds because P is type-correct. If $E \notin \mathit{Bad}(P, C, I)$, we have that $[E]_{P'} = I$, and $t \notin \mathit{TC}(P')$ because rule (16) does not apply.
   (b) $[(D)E]_P = C$ and $(D)E \notin \mathit{Bad}(P, C, I)$. The cast occurs as $(I)E$ in P', and $t \notin \mathit{TC}(P')$ because rule (16) does not apply.
4. $t \equiv [E] \leq [E']$. We distinguish the following cases:
   (a) $E \notin \mathit{All}(P, C)$ and $E' \notin \mathit{All}(P, C)$. Then, $[E]_P = [E]_{P'}$ and $[E']_P = [E']_{P'}$. Hence, $[E]_{P'} \leq [E']_{P'}$.
   (b) $E \notin \mathit{All}(P, C)$ and $E' \in \mathit{All}(P, C)$. Then, $[E]_P = [E]_{P'}$. Two sub-cases exist: (i) if $E' \in \mathit{Bad}(P, C, I)$, then $[E']_{P'} = [E']_P$ and $[E]_{P'} \leq [E']_{P'}$. (ii) if $E' \notin \mathit{Bad}(P, C, I)$, then $C = [E']_P \leq [E']_{P'} = I$. Hence, $[E]_{P'} \leq [E']_{P'}$.
   (c) $E \in \mathit{All}(P, C)$ and $E' \notin \mathit{All}(P, C)$. Then, $[E']_{P'} = [E']_P$. Two sub-cases exist: (i) If $I \not\leq [E']_P$, then from Definition 5 it follows that $E \in \mathit{Bad}(P, C, I)$, 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 $[E]_{P'} \leq \mathrm{I} \leq [E']_{P'}$.
   (d) $E \in \mathit{All}(P, C)$ and $E' \in \mathit{All}(P, C)$. Two sub-cases exist: (i) If $E' \in \mathit{Bad}(P, C, I)$, Definition 5 implies that $E \in \mathit{Bad}(P, C, I)$. Hence, $[E]_P = [E]_{P'} = [E']_P = [E']_{P'} = C$, and $[E]_{P'} \leq [E']_{P'}$. (ii) If $E' \notin \mathit{Bad}(P, C, I)$, then $[E'] = I$ in P'. Depending on whether or not $E \in \mathit{Bad}(P, C, I)$, we have $[E]_{P'} = C$ or $[E]_{P'} = I$, but in either case $[E]_{P'} \leq [E']_{P'}$.
5. $t \equiv [E] \leq C_1$ **or** $\cdots$ **or** $[E] \leq C_k$, $k \geq 1$. Then, t is due to a virtual call $E.m(\cdots)$ to a method M (rule (4)). Definition 2 implies $[E]_P \leq C_h$, for some $1 \leq h \leq k$, where M overrides a method M_h in C_h. Two cases exist:
   (a) $I \leq C_j$ for some $1 \leq j \leq k$. (we must use '$\leq$' instead of '$=$' as C_j may be `Object`.) Then, $[E]_P \leq C_j$. Two sub-cases exist: (i) $E \in \mathit{All}(P, C)$ and $E \notin \mathit{Bad}(P, C, I)$. Then, $[E]_{P'} = I \leq C_j$. (ii) $E \notin \mathit{All}(P, C)$ or $E \in \mathit{Bad}(P, C, I)$. Then, $[E]_{P'} = C \leq I \leq C_j$.
   (b) $I \not\leq C_j$ for all $1 \leq j \leq k$. From Definition 5, it follows that $E \in \mathit{Bad}(P, C, I)$, and hence that $[E]_{P'} = [E]_P \leq C_h$.

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

Conjecture 1 (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 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 $\mathit{All}(P, C) \setminus \mathit{Bad}(P, C, I)$. Then, P and P' have corresponding program behaviors.

We plan to prove Conjecture 1 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 2 (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 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:

Definition 6 (Layer). *Let $Layer(E) : Bad(P, C, I) \to \mathbb{N}$ be defined as follows:*

$$Layer(E) = \begin{cases} 0 & \textbf{if } [E] \leq C_1 \textbf{ or } \cdots \textbf{ or } [E] \leq C_k \in TC_{\text{fixed}}(P), \\ & \quad I \not\leq C_1, \cdots, I \not\leq C_k \\ 0 & \textbf{if } [E] \leq [E'] \in TC_{\text{fixed}}(P),\ E' \notin All(P, C), I \not\leq [E'] \\ n+1 & \textbf{if } Layer(E) \neq 0 \textit{ and } n = min(\{\ m = Layer(E') \mid \\ & \quad E' \neq E, [E] = [E'] \in TC_{\text{fixed}}(P) \textit{ or} \\ & \quad [E] \leq [E'] \in TC_{\text{fixed}}(P) \textit{ or } [E] < [E'] \in TC_{\text{fixed}}(P)\ \}) \end{cases}$$

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 6, it follows that there are two subcases: (i) $[E] \leq C_1$ **or** $\cdots$ **or** $[E] \leq C_k \in TC_{\text{fixed}}(P)$, $I \not\leq C_1, \cdots, I \not\leq C_k$, 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 $E' \in Bad(P, C, I)$, 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. □

3 Other Refactorings Related to Generalization

3.1 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 5 to illustrate the various

```
abstract class List {
  int size(){ return this.size; }
  void setSize(int i){ this.size = i; }
  Object[] elems; int size;
}
class CList extends List {
  CList(Object[] 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 Object[10]; this.size = 0; }
  void add(Object e) {
    if (this.size() + 1 == this.elems.length){
      Object[] newObjects = new Object[2 * this.size()];
      System.arraycopy(elems, 0, newObjects, 0, this.size());
      this.elems = newObjects;
    }
    this.set(this.size(), e);
    this.setSize(this.size() + 1);
  }
  int size(){ int n = this.size; return n; }
  boolean isEmpty(){ return this.size() == 0; }
  FList sort(){
    for (int t = 0; t < this.size(); t++){
      for (int u = t + 1; u < this.size(); u++){
        Object e1=this.elems[t]; Comparable e2=(Comparable)this.elems[u];
        if (e2.compareTo(e1) < 0){ this.elems[t]=e2; this.elems[u]=e1; }
      }
    }
    return this;
  }
  Object get(int index){ return this.elems[index]; }
  void set(int index, Object o){ this.elems[index] = o; }
}
class Client {
  public static void main(String[] args) {
    FList l1 = new FList();
    l1.add("foo"); l1.add("bar"); l1.sort();
    System.out.println(l1.toString());
    List l2 = new CList(new Object[]{ "zip", "zap" });
    System.out.println(l2.toString());
  }
}
```

Fig. 5. Example program P_2.

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` is `empty()`, 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 `l1.toString()` now dispatches to a different definition of the `toString()` method.

3.2 Using type constraints for PULL UP MEMBERS

We will reuse the type constraints of Figure 2 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 "variables", and the declaring classes of methods were "constants". In the case of PULL UP MEMBERS, the opposite is true: 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.

Figure 6 shows the type constraints for P_2. Here, the declaring class of a member M is shown in unsimplified form, as *Decl*(M), and the type of a declaration element E as the constant value $[E]_P$. 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()`) $\leq$ *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()`) $\leq$ `FList`.
- The remaining problem case—pulling up `CList.toString()`—does not raise any type-correctness issues. This illustrates that type constraints by themselves are not always sufficient to express the preconditions of refactorings.

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

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),(22)
	`FList` ≤ *Decl*(`FList.sort()`)	(4),(22)

Fig. 6. Type constraints $TC_{\text{fixed}}(P_2)$ for program P_2 of Figure 5. Only nontrivial constraints related to types `List`, `CList`, and `FList` are shown.

Definition 7 below introduces a predicate *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 7, 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 $\mathit{Decl}(M) \leq \mathit{Decl}(M_1)$ **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'. By requiring that $\mathit{super}(\mathit{Decl}_P(M)) < \mathit{Decl}_P(M')$, we ensure that no class contains methods with identical signatures after the pull-up.

(e) The final condition in Definition 7 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 $super(Decl_P(M))$, the dispatch should resolve[7] to a proper subtype of $super(Decl_P(M))$.

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 program execution behavior. Work on an exact version of (d) is still in progress.

Definition 7. *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:*

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

It is straightforward to extend *CanPullUp* to nonvirtual methods, and to sets of methods. 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.

3.3 Other refactorings

Other refactorings that can be modeled using our approach include:

Generalize Type. This refactoring replaces the type of a 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 5, the return type of `FList.sort()` is generalized to `List`, `FList.sort()` can be pulled up into class `List`.

Extract Subclass [5, 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 [5, page 328]. This is the inverse refactoring of Pull Up Members, and although it is technically a refactoring for *specialization*, the issues that arise are very similar to those for Pull Up Members.

[7] Here, $staticLookup(P, C, S)$ is a function that, for a given program P and class C, selects the definition of a method with signature S in the nearest superclass of C.

program construct	implied type constraint(s)	
`new` $C[E]$	$[\texttt{new } C[E]] = C[\,]$	(26)
	$[E] = \textsc{Int}$	(27)
expression $C[E]$	$[C[E]] = C$	(28)
	$[E] = \textsc{Int}$	(29)
for any $C \leq C'$	$C[\,] \leq C'[\,]$	(30)
initialized declaration C[ ] c = $\{E_1, \ldots, E_n\}$	$[E_i] \leq C$	(31)
initialized creation expression `new` $C[E]\{E_1, \ldots, E_n\}$	$[E_i] \leq C$	(32)
`try {...} catch(E e){...}`	$E \leq$ `java.lang.Throwable`	(33)
`try{...} catch(`$\texttt{E}_1$ $\texttt{e}_1$`){...}` ... `catch(`$\texttt{E}_n$ $\texttt{e}_n$`){...}`	$\forall i, j : 1 \leq i < j \leq n$ $E_j \not\leq E_i$	(34)
M overrides M'	$\forall E \in \mathit{Excptns}(M) \cap \mathit{CheckedExcptns}$ $\exists E' \in \mathit{Excptns}(M')$ $E \leq E'$	(35)
expression E `instanceof` C	[E `instanceof` C] = Bool	(36)
	$[E] \leq C$ **or** $C \leq [E]$ if C is a class and $[E]$ is a class	(37)

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

Excptns(M) set of exceptions included in `throws` clause of M's declaration

Fig. 7. Additional type constraints.

4 Implementation and Pragmatic Issues

We have implemented Extract Interface in Eclipse [4]. Our implementation closely follows Definition 5 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 3 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 7.

Arrays. Arrays and array initializers introduce some rather straightforward constraints, as shown in Figure 7.

Casts and `instanceof` expressions. Type constraints implied by `instanceof` expressions (rule (37)) are analogous to those for casts (rule (16)) [8, 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 [8, Section 5].

Member types. Member types [8, 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 7 as rules (33)–(35). 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 [9].

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 $m(C_1, \cdots, C_N)$ results in selection of a method $m(\cdots)$ with the most specific signature that matches $m(C_1, \cdots, C_N)$ [8, 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.

5 Related Work

In his pioneering work on refactoring, Opdyke [13] identified and informally specified invariants that any refactoring must preserve [13, 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 [5] 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* [10] 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 [9] 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.

Tokuda and Batory [20] discuss the use of refactorings to introduce new (or evolve existing) design patterns [7] 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 [16] 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 [17, 18] 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 [19] 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., [15, 6]). 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.

6 Future Work

Plans for future work include a complexity analysis of our algorithms, and a detailed study of other generalization-related refactorings (see Section 3.3). With respect to EXTRACT INTERFACE, we plan to extend the declaration-updating process to include declaration elements whose type is a subtype of the class from which the interface is extracted. Furthermore, we are interested in providing more interactive feedback in refactoring tools using type constraints. For example, one could inform the user of declarations that can be generalized when a given method is pulled up, or declarations that need to be generalized in order to enable a PULL UP operation that is otherwise prohibited.

Acknowledgments

We are grateful to Jens Palsberg for comments on a draft of this paper.

References

1. Dirk Bäumer, Erich Gamma, and Adam Kieżun. Integrating refactoring support into a Java development tool. In *OOPSLA'01 Companion*, October 2001.
2. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
3. Cohan Sujay Carlos. The elimination of overheads due to type annotations and the identification of candidate refactorings. Master's thesis, North Carolina State University, 2002.
4. Eclipse.org. Eclipse. On-line at http://www.eclipse.org.
5. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.
6. Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Proceedings of SAS'00, International Static Analysis Symposium*, pages 199–219. Springer-Verlag (*LNCS* 1824), 2000.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (Second Edition)*. Addison-Wesley, 2000.
9. T. J. Halloran and William L. Scherlis. Models of Thumb: Assuring best practice source code in large Java software systems. Technical Report Fluid Project, School of Computer Science/ISRI, Carnegie Mellon University, September 2002.
10. JetBrains, Inc. Intellij IDEA. On-line at http://www.intellij.com/jetbrains.
11. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Inc., 1997.
12. William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *The ACM 1993 Computer Science Conf. (CSC'93)*, pages 66–73, February 1993.
13. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University Of Illinois at Urbana-Champaign, 1992.
14. Jens Palsberg and Michael Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.
15. Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.
16. Christopher Seguin. Refactoring tool challanges in a strongly typed language. In *OOPSLA'00 Companion*, pages 101–102, October 2000.
17. Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 99–110, Orlando, FL, November 1998.
18. G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Trans. on Programming Languages and Systems*, pages 540–582, May 2000.
19. Frank Tip and Peter Sweeney. Class hierarchy specialization. *Acta Informatica*, 36:927–982, 2000.
20. Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Kluwer Journal of Automated Software Engineering*, pages 89–120, August 2001.