Scalable Libraries with Safe Conditional Methods

Shan Shan Huang David Zook Yannis Smaragdakis
Georgia Institute of Technology
College of Computing
{ssh, dzook, yannis}@cc.gatech.edu

Abstract

Recent research in Programming Languages has yielded proposals for supporting the concept of "optional" or "conditional" methods in a library without sacrificing static type safety. Specifically, the cJ extension to Java and the Generalized Constraints proposal for C# follow technically different avenues, yet both enable specifying methods that are defined only under specific type conditions. Effectively, these language features enable a safe analogue of the C/C++ "`#ifdef`*" construct. Using such conditional methods, a library designer can support optional functionality, yet statically ensure that programmers only use it when doing so is safe. In this paper, we show how our cJ language enables scalable library design with optional methods. As a case study, we describe how cJ solves the safety problems of the Java Collections Framework—the standard Java data structures library—without sacrificing conciseness. This addresses a well-known issue with the Java Collections Framework.*

1. Introduction

2 Libraries with Optional Methods

2.1 cJ Background

2.2 cJ Optional Methods

2.3 "Implicit" Subtyping in cJ

3 Case Study: Java Collections Framework

The Java Collections Framework is the standard Java data structures library. The design of this library is a striking demonstration of the difficulties in maintaining both static type safety and scalability, and the inability to do so effectively with features available in Java today. The Java Collections Framework supplies four main interfaces: `Collection`, `Set`, `Map`, and `List`, and a number of more specialized interfaces and classes all extending (implementing) these main interfaces. However, not all data structures support all the methods in their corresponding interfaces. Six out of the fifteen methods of interface `Collection` in JDK 1.5 are optional and may result in run-time errors. A common optional concept, for instance, is that of "modifiability": is the collection modifiable through its public interface or not? This concept is not captured via the Java type system in the design of the Java Collections Framework. Instead, any code attempt to modify an "unmodifiable" collection passes compile-time checks, only to result in the throwing of an `UnsupportedOperationException` at *run-time*. Another similar notion is that of size variability. Some collections are modifiable, yet their size cannot change—arrays are a standard example. An array supports the operations of the `List` interface with the exception of `add` or `remove`, which throw an `UnsupportedOperationException`.

The above is a well-known issue with Java Collections. In fact, the very first "frequently asked question" in the Java Collections API Design FAQ[1] is:

> Why don't you support immutability directly in the core collection interfaces so that you can do away with optional operations (and `UnsupportedOperationException`)?

The design rationale reflected in the answer to this FAQ indirectly offers a compelling argument for cJ. The developers note:

> Clearly, static (compile time) type checking is highly desirable, and is the norm in Java. We would have supported it if we believed it were feasible. Unfortunately, attempts to achieve this goal cause an explosion in the size of the interface hierarchy ...

[1]http://java.sun.com/j2se/1.5.0/docs/guide/collections/designfaq.html

The Java Collections API developers proceed to give external testimony from other Java developers (e.g., Doug Lea's experience with a collections package) and an illustration of the kinds of "explosion in size" problems that a type-safe design would encounter, if concepts such as "modifiable", "variable-size", and "append-only" are expressed in the type system.

The "explosion in size" becomes obvious in the next section, as we show how one would implement a type-safe Java Collections Framework, in Java.

3.1 A Type-safe Java Collections Framework — In Java

Here, we explore how one would implement a type-safe Java Collections Framework, while incorporating three optional fetaures: modifiability, delete-only, and variable-size. Note that there are only two *orthogonal* variabilities: modifiability is orthogonal to both delete-only and variable-size. Modifiability indicates the collection (or map)'s ability to modify the content of a particular cell, while delete-only (or variable-size) indicates the collection (map)'s ability to remove (or both remove and add) a cell from the collection (map). Delete-only and variable-size, however, are not orthogonal properties — any variable-size collection(map) is also a delete-only collection(map).

In this section, we only rewrite the four main interfaces in the Java Collections Framework: `Collection`, `Map`, `List`, and `Set`. `Collection`, `List`, and `Set` also return `Iterator` and `ListIterator` interfaces, where `Iterator` supports only forward iteration and removal of the current element, while `ListIterator` supports bi-directional iteration, as well as addition and modification of elements. As we will show, these iterator interfaces also need to be rewritten to ensure type safety.

To support optional concepts, we first define a base set of interfaces where all methods are non-optional. The following are our base interfaces:

```
interface Collection<E> {
   boolean contains(Object o);
   boolean isEmpty();
   Iterator<E> iterator();
   ...
}
interface List<E> extends Collection<E> {
   E get(int index);
   ListIterator<E> listIterator();
   ...
}
interface Set<E> extends Collection<E> {
   int size();
   ...
}
interface Map<K,V> {
   V get(Object key);
   Set<K> keySet();
   ...
}
interface Iterator<E> {
   boolean hasNext();
   E next();
}
interface ListIterator<E>
      extends Iterator<E> {
   boolean hasPrevious();
   E previous();
   ...
}
```

In the original Java Collections Framework, `List` and `ListIterator` are the only interface with optional "modifiable" behavior. Thus, we add interfaces `ModifiableList` and `ModifiableListIterator` to our list of interfaces:

```
interface ModifiableList<E> extends List<E> {
   E set(int index, E element);
   ModifiableListIterator<E> listIterator();
   ...
}
interface ModifiableListIterator<E>
      extends ListIterator<E> {
   void set (E o);
}
```

We next define interfaces to support the optional "delete-only" behavior:

```
interface DeleteOnlyCollection<E>
       extends Collection<E> {
   boolean remove(Object o);
   void clear();
   DeleteOnlyIterator<E> iterator();
   ...
}
interface DeleteOnlyList<E>
       extends List<E> {
   boolean remove(int index);
   DeleteOnlyIterator<E> iterator();
   DeleteOnlyListIterator<E> listIterator();
   ...
}
interface DeleteOnlySet<E>
       extends Set<E> {
   boolean remove(Object o);
   DeleteOnlyIterator iterator();
   ...
}
interface DeleteOnlyMap<K,V>
       extends Map<K,V> {
   V remove(Object key);
   ...
}
interface DeleteOnlyIterator<E>
       extends Iterator<E> {
   void remove();
}
interface DeleteOnlyListIterator<E>
       etxends ListIterator<E> {
   void remove();
}
```

We have now defined interfaces to support two optional features, and we are up to 9 interfaces for the data structures, and 4 iterators.

To add the optional "variable-size" features, we need to add the following interfaces:

```
interface VariableSizeCollection<E>
      extends DeleteOnlyCollection<E> {
   boolean add(E o);
   ...
}
interface VariableSizeList<E>
      extends DeleteOnlyList<E> {
   boolean add(int index, E o);
   VariableSizeListIterator<E> listIterator();
   ...
}
interface VariableSizeSet<E>
      extends DeleteOnlySet<E> {
   boolean add(E o);
   ...
}
interface VariableSizeMap<K,V>
      extends DeleteOnlyMap<K,V> {
   V put(K key, V value);
   ...
}
interface VariableSizeListIterator<E>
      extends DeleteOnlyListIterator<E> {
   void add(E o);
}
```

We have now defined all interfaces to support three optional features (two orthogonal ones), and we are already up to 13 data structure interfaces and 5 iterators. The Java Collections Design FAQ specified even more axis of variability likely to arise in practice, e.g. an "append-only" optional feature for log-style lists. If we were to add more variability, the number of interfaces will quickly become unmaintainable.

The Java Collections Design FAQ concludes to support our observation:

> Now we're up to twenty or so interfaces and five iterators, and it's almost certain that there are still collections arising in practice that don't fit cleanly into any of the interfaces.

Thus, the design of the current Java Collections Framework circumvents the static type system in order to avoid a combinatorial explosion in the number of types specified in the library.

3.2 cJ Collections Framework — Type-safe, Concise, and Enhanced

cJ addresses fully and cleanly the above problem with the Java Collections Framework. In this section, we show the cJ implementation of the Collections Framework, which is both type-safe, and maintains the same number of interfaces for data structures and iterators. Additionally, the conditional interfaces feature of cJ allows us to enhance the Collections Framework by having a collection of `Comparable` elements also implementing the `Comparable` interface.

First, we define one "marker" interface for each optional property:

```
interface Modifiable {}
interface DeleteOnly {}
interface VariableSize extends DeleteOnly {}
```

We then redefine `Collection`, `List`, `Set`, and `Map` with one extra type parameter, `O`, used exclusively for configuring optional features:

```
interface Collection<E,O>
      <E extends Comparable<E>>?
      extends Comparable<Collection<E,O>> {
   boolean contains(Object o);
   Iterator<E,O> iterator();
   <O extends DeleteOnly>?
   <  boolean remove(Object o);
      void clear();
      ...
   >
   <O extends VariableSize>?
   <  boolean add(E o);
      ...
   >
   ...
}
interface List<E,O> extends Collection<E,O> {
   ListIterator<E,O> listIterator();
   <O extends Modifiable>?
   <  E set(int index, E element);
      ...
   >
   <O extends DeleteOnly>?
   <  boolean remove(int index);
      ...
   >
   <O extends VariableSize>?
   <  boolean add(int index, E o);
      ...
   >
   ...
}
interface Set<E,O> extends Collection<E,O> {
   int size();
   <O extends DeleteOnly>?
   <  boolean remove(Object o);
```

```
      ...
   >

   <O extends VariableSize>?
   <  boolean add(E o);
      ...
   >
   ...
}
interface Map<K,V,O> {
   V get (Object key);
   Set<K,O> keySet();
   <O extends DeleteOnly>?
   <  V remove(Object key);
      ...
   >
   <O extends VariableSize>?
   <  V put(K key, V value);
      ...
   >
   ...
}
```

Similarly, the iterator interfaces are redefined with the extra type parameter for optional feature configuration:

```
interface Iterator<E,O> {
   boolean hasNext();
   E next();
   <O extends DeleteOnly>?
   void remove();
}
interface ListIterator<E,O>
      extends Iterator<E,O> {
   <O extends DeleteOnly>?
   <  void remove();
      ...
   >
   <O extends VariableSize>?
   <  void add(E o);
      ...
   >
}
```

To use these interfaces, a user chooses the flavor of collection or map she wants, and indicate the flavor through the second type parameter. For example, `List<String,Modifiable>` is a modifiable `List`, whereas `List<String,Object>` is an unmodifiable `List` (`Object` is not a subtype of `Modifiable`). To combine orthogonal flavors, one can define a new interface which combines the flavors:

```
interface ModifiableVariableSize
     extends Modifiable,VariableSize {}
List<String,ModifiableVariableSize> l;
```

In the above code segment, `l` is a `List` that is both modifiable and variable in size.

4 Related Work

5 Conclusions