

cJ: Enhancing Java with Safe Type Conditions

Shan Shan Huang^{1,2} David Zook¹ Yannis Smaragdakis²

¹College of Computing
Georgia Institute of Technology

²Department of Computer and Information Science
University of Oregon

6th International Conference on Aspect Oriented Software
Development



The Problem, and the Non-Solution

- Problem: Specialize reusable code.

```
template <class E>
class List {

    bool add(const E &e) { ... }

};
```



The Problem, and the Non-Solution

- Problem: Specialize reusable code.

```
template <class E>
class List {

    bool add(const E &e) { ... }

};
```



The Problem, and the Non-Solution

- Problem: Specialize reusable code.
- Solution in C++: `#ifdef`

```
template <class E>
class List {

    bool add(const E &e) { ... }

};
```



The Problem, and the Non-Solution

- Problem: Specialize reusable code.
- Solution in C++: `#ifdef`

```
template <class E>
class List {
    #ifdef VariableSize
        bool add(const E &e) { ... }
    #endif
};
```



The Problem, and the Non-Solution

- Problem: Specialize reusable code.
- Solution in C++: `#ifdef`

```
template <class E>
class List {
    #ifdef VariableSize
        bool add(const E &e) { ... }
    #endif
};
class Client {
    void meth() {
        List<string> ls;
        ls.add("John");
    }
};
```

The Problem, and the Non-Solution

- Problem: Specialize reusable code.
- Solution in C++: `#ifdef`

```
template <class E>
class List {
    #ifdef VariableSize
        bool add(const E &e) { ... }
    #endif
};
class Client {
    void meth() {
        List<string> ls;
        ls.add("John");
    }
};
```

The Problem, and the Non-Solution

- Problem: Specialize reusable code.
- **NON**-Solution in C++: #ifdef

```
template <class E>
class List {
    #ifdef VariableSize
        bool add(const E &e) { ... }
    #endif
};
class Client {
    void meth() {
        List<string> ls;
        ls.add("John");
    }
};
```

#ifdef Is Like a Vegas Wedding!

Sure, it's flexible...



Outline

- 1 cJ in a Nutshell
 - Extension of Java with static-ifs
 - Statically safe library specialization (like `#ifdef` done right!)
- 2 Type Hierarchies as Concerns
 - Case Study: Java Collections Framework
- 3 Using cJ to Separate Concerns
 - cJ Collections Framework

Outline

- 1 cJ in a Nutshell
 - Extension of Java with static-ifs
 - Statically safe library specialization (like `#ifdef` done right!)
- 2 Type Hierarchies as Concerns
 - Case Study: Java Collections Framework
- 3 Using cJ to Separate Concerns
 - cJ Collections Framework

Structure of Class Vary Based on Type Conditions!

- Methods and fields can be **conditionally declared** within a cJ class.

Structure of Class Vary Based on Type Conditions!

- Methods and fields can be **conditionally declared** within a cJ class.

```
class Foo<T> {  
    ...  
    <T extends Bar>?  
    int i;  
}
```

Structure of Class Vary Based on Type Conditions!

- Methods and fields can be **conditionally declared** within a cJ class.

```
class Foo<T> {  
    ...  
    <T extends Bar>?  
    int i;  
}
```

```
Foo<Bar> foobar = new Foo<Bar>();  
... foobar.i ... // OK
```

Structure of Class Vary Based on Type Conditions!

- Methods and fields can be **conditionally declared** within a cJ class.

```
class Foo<T> {  
    ...  
    <T extends Bar>?  
    int i;  
}
```

```
Foo<Object> foob = new Foo<Object>();  
... foob.i ... // Compile-time error!
```

More on Safe Type Conditions

- Even superclasses and interfaces can be conditional!

```
interface Collection<E>
  <E extends Serializable>? extends Serializable
{
  ...
}
```

More on Safe Type Conditions

- Even superclasses and interfaces can be conditional!

```
interface Collection<E>
  <E extends Serializable>? extends Serializable
{
  ...
}
```

Maintaining Separate Checkability: I

- Code type-safe under current type conditions:

```
class Farm<T> {
  <T extends Dog>?
  void makeItBark(T d) {
    ...d.bark()...
  }
}

public class Dog {
  public void bark() { ... }
}
```

Maintaining Separate Checkability: I

- Code type-safe under current type conditions:

```
class Farm<T> {  
    <T extends Dog>?  
    void makeItBark(T d) {  
        ...d.bark()...  
    }  
}  
  
public class Dog {  
    public void bark() { ... }  
}
```

Maintaining Separate Checkability: I

- Code type-safe under current type conditions:

```
class Farm<T> {  
    <T extends Dog>?  
    void makeItBark(T d) {  
        ...d.bark()...  
    }  
}  
  
public class Dog {  
    public void bark() { ... }  
}
```

Maintaining Separate Checkability: I

- Code type-safe under current type conditions:

```
class Farm<T> {  
    <T extends Dog>?  
    void makeItBark(T d) {  
        ...d.bark()...  
    }  
}  
  
public class Dog {  
    public void bark() { ... }  
}
```

Maintaining Separate Checkability: II

- Type conditions for references must be equivalent or stronger than type conditions for declarations.

```
class Foo<T> {  
    <T extends Bar>? int i;  
  
    <T extends Bar & Baz>?  
    void incI () { i++; }  
  
    void decI () { i--; }  
}
```

Maintaining Separate Checkability: II

- Type conditions for references must be equivalent or stronger than type conditions for declarations.

```
class Foo<T> {  
    <T extends Bar>? int i;  
  
    <T extends Bar & Baz>?  
    void incI () { i++; }  
  
    void decI () { i--; }  
}
```

Maintaining Separate Checkability: II

- Type conditions for references must be equivalent or stronger than type conditions for declarations.

```
class Foo<T> {  
    <T extends Bar>? int i;  
  
    <T extends Bar & Baz>?  
    void incI () { i++; }  
  
    void decI () { i--; }  
}
```

Maintaining Separate Checkability: II

- Type conditions for references must be equivalent or stronger than type conditions for declarations.

```
class Foo<T> {  
    <T extends Bar>? int i;  
  
    <T extends Bar & Baz>?  
    void incI () { i++; }  
  
    void decI () { i--; }  
}
```

Maintaining Separate Checkability: II

- Type conditions for references must be equivalent or stronger than type conditions for declarations.

```
class Foo<T> {  
    <T extends Bar>? int i;  
  
    <T extends Bar & Baz>?  
    void incI () { i++; } // OK  
  
    void decI () { i--; }  
}
```

Maintaining Separate Checkability: II

- Type conditions for references must be equivalent or stronger than type conditions for declarations.

```
class Foo<T> {  
    <T extends Bar>? int i;  
  
    <T extends Bar & Baz>?  
    void incI () { i++; } // OK  
  
    void decI () { i--; }  
}
```

Maintaining Separate Checkability: II

- Type conditions for references must be equivalent or stronger than type conditions for declarations.

```
class Foo<T> {  
    <T extends Bar>? int i;  
  
    <T extends Bar & Baz>?  
    void incI () { i++; } // OK  
  
    void decI () { i--; } // Compile-time error!  
}
```

Outline

- 1 cJ in a Nutshell
 - Extension of Java with static-ifs
 - Statically safe library specialization (like `#ifdef` done right!)
- 2 Type Hierarchies as Concerns
 - Case Study: Java Collections Framework
- 3 Using cJ to Separate Concerns
 - cJ Collections Framework

Java Collections Framework

- THE Java data structures library.
- Data structure classes/interfaces, such as Collection, List, Set, Map, etc.

Java Collections Framework

- THE Java data structures library.
- Data structure classes/interfaces, such as Collection, List, Set, Map, etc.



The “Concerns” of a Data Structure

Concerns intrinsic to the data structure:

- Storage: linked-list, array, hash table, etc.
- Size of the structure.

Cross-cutting concerns:

- Can its size change?
 - If size can change, in what manner? Deletion-only?
Append-only?
- Can its contents be modified?
- ...

JCF's Solution: Mangled Concerns!

```
public interface Collection<E> {  
    public int size();  
    public boolean contains(Object o);  
    ...  
  
    public boolean add(E e);  
    public boolean remove(Object o);  
}
```

JCF's Solution: Mangled Concerns!

```
public interface Collection<E> {  
    public int size();  
    public boolean contains(Object o);  
    ...  
  
    // Size modifying operations:  OPTIONAL METHODS!!  
    public boolean add(E e);  
    public boolean remove(Object o);  
}
```

JCF's Solution: Mangled Concerns!

```
public interface Collection<E> {  
    public int size();  
    public boolean contains(Object o);  
    ...  
  
    // Size modifying operations:  OPTIONAL METHODS!!  
    public boolean add(E e);  
    public boolean remove(Object o);  
}
```

- Throws **UnsupportedOperationException** at **RUNTIME!**

JCF's Solution: Mangled Concerns!

```
public interface Collection<E> {  
    public int size();  
    public boolean contains(Object o);  
    ...  
  
    // Size modifying operations:  OPTIONAL METHODS!!  
    public boolean add(E e);  
    public boolean remove(Object o);  
}
```

- Throws **UnsupportedOperationException** at **RUNTIME!**
- **6 out of 15** methods in `Collection` are *optional*.

Even More Optional Methods:

```
public interface List<E> extends Collection<E> {  
    public E get(int index);  
    ...  
  
    public E add(int index, E element);  
  
    public E set(int index, E element);  
}
```

Even More Optional Methods:

```
public interface List<E> extends Collection<E> {  
    public E get(int index);  
    ...  
  
    // Size modifying operations: OPTIONAL METHODS  
    public E add(int index, E element);  
  
    public E set(int index, E element);  
}
```

Even More Optional Methods:

```
public interface List<E> extends Collection<E> {  
    public E get(int index);  
    ...  
  
    // Size modifying operations: OPTIONAL METHODS  
    public E add(int index, E element);  
  
    // Content modifying operations: OPTIONAL METHODS  
    public E set(int index, E element);  
}
```

A Real Piece of Code from JCF:

```
class UnmodifiableCollection<E>
    implements Collection<E> ... {
    ...

    public boolean add(E o) {
        throw new UnsupportedOperationException();
    }
    public boolean remove(Object o) {
        throw new UnsupportedOperationException();
    }
}
```

A Real Piece of Code from JCF:

```
class UnmodifiableCollection<E>
    implements Collection<E> ... {
    ...

    public boolean add(E o) {
        throw new UnsupportedOperationException();
    }
    public boolean remove(Object o) {
        throw new UnsupportedOperationException();
    }
}
```

The Official Rationale:

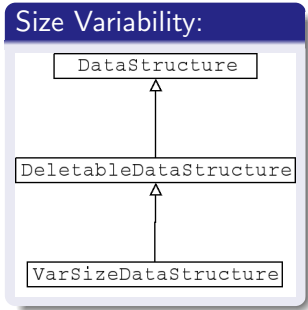
The first question on FAQ for JCF:

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

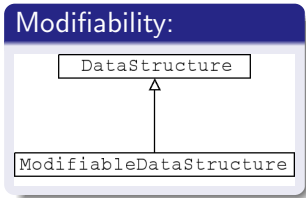
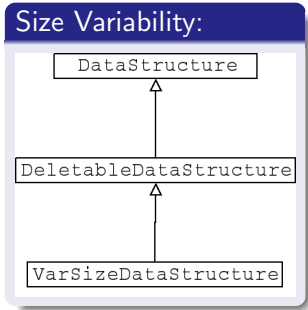
The answer (much abbreviated):

“Clearly, static type checking is highly desirable... Unfortunately, attempts to achieve this goal cause an explosion in the size of the interface hierarchy.”

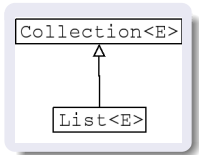
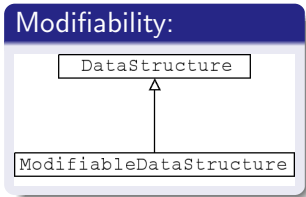
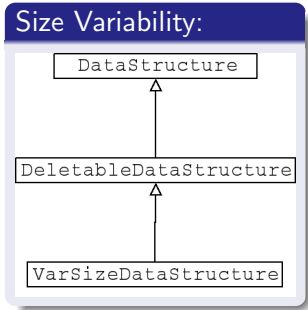
Weaving Type Hierarchies (Concerns)



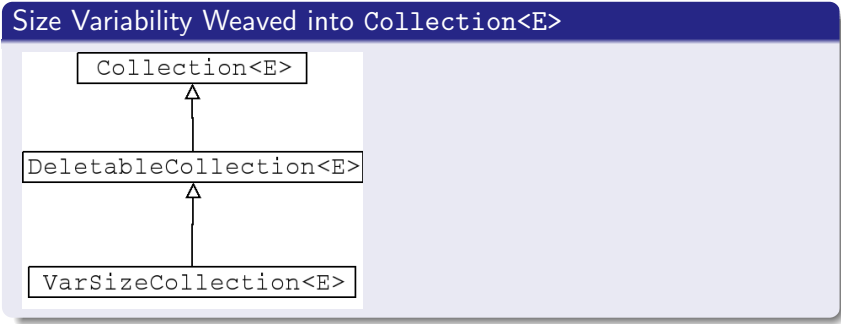
Weaving Type Hierarchies (Concerns)



Weaving Type Hierarchies (Concerns)

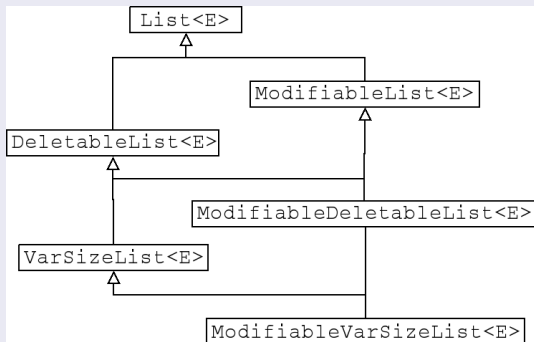


Type Hierarchy Concerns Weaved into Collection:



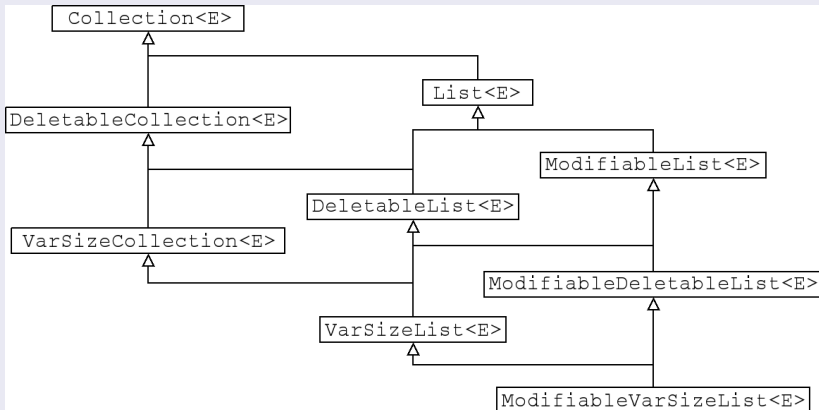
Type Hierarchy Concerns Weaved into List:

Size Variability and Modifiability Weaved into List<E>



Put It All Together:

Both Concerns Weaved into Collection and List



Just A Reminder...

What it looked like before:

Collection<E>



List<E>



And That Was Only Two Concerns and Two interfaces...

- JCF has 16 main interfaces.
- More concerns could arise in practice: persistent vs. not, etc.

Other Designers' Testimony:

“Much as it pains me to say it, strong static typing does not work for collection interfaces in Java.” – Doug Lea

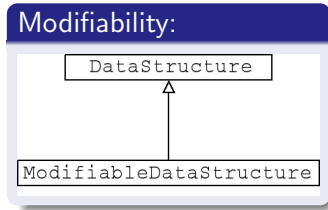
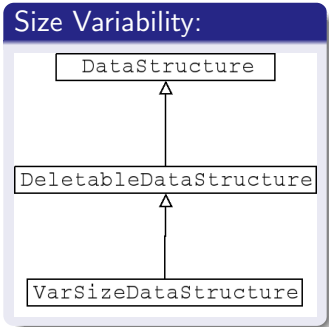
And That Was Only Two Concerns and Two interfaces...

But that was before cJ!!!

Outline

- 1 cJ in a Nutshell
 - Extension of Java with static-ifs
 - Statically safe library specialization (like `#ifdef` done right!)
- 2 Type Hierarchies as Concerns
 - Case Study: Java Collections Framework
- 3 Using cJ to Separate Concerns
 - cJ Collections Framework

Review: Type Hierarchy Concerns



Collections: the cJ Way

- Interfaces to represent concerns type hierarchy:

Size variability:

```
public interface Deletable {}  
public interface VariableSize extends Deletable {}
```

Modifiability:

```
public interface Modifiable {}
```

Weaving concerns into Collection:

```
public interface Collection<E> {  
    public int size();  
    public boolean contains(Object o);  
    ...  
  
    public boolean add(E e);  
  
    public boolean remove(Object o);  
}
```

Weaving concerns into Collection:

```
public interface Collection<E,M> {  
    public int size();  
    public boolean contains(Object o);  
    ...  
  
    public boolean add(E e);  
  
    public boolean remove(Object o);  
}
```

Weaving concerns into Collection:

```
public interface Collection<E,M> {  
    public int size();  
    public boolean contains(Object o);  
    ...  
  
    <M extends VariableSize>?  
    public boolean add(E e);  
  
    public boolean remove(Object o);  
}
```

Weaving concerns into Collection:

```
public interface Collection<E,M> {  
    public int size();  
    public boolean contains(Object o);  
    ...  
  
    <M extends VariableSize>?  
    public boolean add(E e);  
  
    <M extends Deletable>?  
    public boolean remove(Object o);  
}
```

Weaving concerns into List:

```
public interface List<E> extends Collection<E> {  
    public E get(int index);  
    ...  
  
    public E add(int index, E element);  
  
    public E set(int index, E element);  
}
```

Weaving concerns into List:

```
public interface List<E,M> extends Collection<E,M> {  
    public E get(int index);  
    ...  
  
    public E add(int index, E element);  
  
    public E set(int index, E element);  
}
```

Weaving concerns into List:

```
public interface List<E,M> extends Collection<E,M> {  
    public E get(int index);  
    ...  
  
    <M extends VariableSize>?  
    public E add(int index, E element);  
  
    public E set(int index, E element);  
}
```

Weaving concerns into List:

```
public interface List<E,M> extends Collection<E,M> {  
    public E get(int index);  
    ...  
  
    <M extends VariableSize>?  
    public E add(int index, E element);  
  
    <M extends Modifiable>?  
    public E set(int index, E element);  
}
```

cJ Collections: Type Safe

- Runtime `UnsupportedOperationException` replaced by compile-time error!

```
List<Dog, Modifiable> dogs =  
    new ArrayList<Dog, Modifiable>();
```

cJ Collections: Type Safe

- Runtime UnsupportedOperationException replaced by compile-time error!

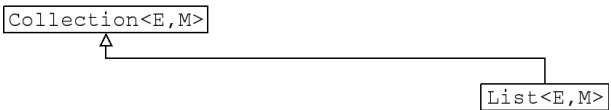
```
List<Dog, Modifiable> dogs =  
    new ArrayList<Dog, Modifiable>();  
  
dogs.set(0, new Dog("Spotty")); //OK
```

cJ Collections: Type Safe

- Runtime `UnsupportedOperationException` replaced by compile-time error!

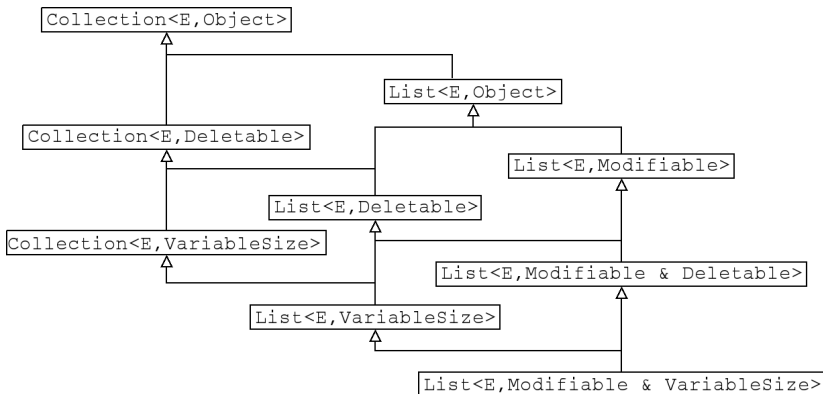
```
List<Dog, Modifiable> dogs =  
    new ArrayList<Dog, Modifiable>();  
  
dogs.set(0, new Dog("Spotty")); //OK  
  
dogs.add(new Dog("Sparky")); //Compile-time error!
```

cJ Collections: Concise Type Hierarchy!



Implicitly Exponential Type Hierarchy

- Cross-cutting type hierarchies “weaved” into explicit type hierarchy



Abstracting Over One Concern

How do we abstract over one concern?

e.g. a List that is variable size, but may or may not be modifiable?

Abstracting Over One Concern

How do we abstract over one concern?

e.g. a `List` that is variable size, but may or may not be modifiable?

Wildcards!

```
List<Dog, ? extends VariableSize> dogs;  
...  
dogs.add(new Dog("Spotty")); // OK  
  
dogs.set(1, new Dog("Sparky")); // NO!
```

Abstracting Over One Concern

How do we abstract over one concern?

e.g. a List that is variable size, but may or may not be modifiable?

Wildcards!

```
List<Dog, ? extends VariableSize & Modifiable> dogs;  
...  
dogs.add(new Dog("Spotty")); // OK  
  
dogs.set(1, new Dog("Sparky")); // OK too!
```

Homogeneous Translation – by Erasure

```
public interface Collection<E,M> {  
    public int size();  
    ...  
  
    <M extends VariableSize>?  
    public boolean add(E e);  
  
    <M extends Deletable>?  
    public boolean remove(Object o);  
  
}
```

Homogeneous Translation – by Erasure

```
public interface Collection<E,M> {  
    public int size();  
    ...  
  
    public boolean add(E e);  
  
    <M extends Deletable>?  
    public boolean remove(Object o);  
  
}
```

Homogeneous Translation – by Erasure

```
public interface Collection<E,M> {  
    public int size();  
    ...  
  
    public boolean add(E e)  
        throws UnsupportedOperationException;  
  
    <M extends Deletable>?  
    public boolean remove(Object o);  
  
}
```

Homogeneous Translation – by Erasure

```
public interface Collection<E,M> {  
    public int size();  
    ...  
  
    public boolean add(E e)  
        throws UnsupportedOperationException;  
  
    public boolean remove(Object o);  
  
}
```

Homogeneous Translation – by Erasure

```
public interface Collection<E,M> {  
    public int size();  
    ...  
  
    public boolean add(E e)  
        throws UnsupportedOperationException;  
  
    public boolean remove(Object o)  
        throws UnsupportedOperationException;  
}
```

Homogeneous Translation – by Erasure

```
public interface Collection<E> {  
    public int size();  
    ...  
  
    public boolean add(E e)  
        throws UnsupportedOperationException;  
  
    public boolean remove(Object o)  
        throws UnsupportedOperationException;  
}
```

Conclusion

- cJ: Enhances Java Generics with statically-safe type conditions
 - Like `#ifdef` done right!!!

Conclusion

- cJ: Enhances Java Generics with statically-safe type conditions
 - Like `#ifdef` done right!!!
- cJ: Separates cross-cutting concerns at the type hierarchy level!
 - Solves JCF-like problems, when multiple cross-cutting concerns cause exponential growth in number of interfaces.

Conclusion

- cJ: Enhances Java Generics with statically-safe type conditions
 - Like `#ifdef` done right!!!
- cJ: Separates cross-cutting concerns at the type hierarchy level!
 - Solves JCF-like problems, when multiple cross-cutting concerns cause exponential growth in number of interfaces.
- cJ: Fits seamlessly into current Java object model.
 - Homogeneous translation
 - Integration with current wildcard mechanism.

Finally...

cJ Availability:
<http://www-static.cc.gatech.edu/~ssh/cj>

QUESTIONS?

