Compiler-Enforced Immutability for the Java Language

Adrian Birka

MIT Laboratory for Computer Science
Technical Report MIT-LCS-TR-908

Revision of Master of Engineering Thesis
MIT Deparment Electrical Engineering and Computer Science

June, 2003

Contents

List of Figures

Abstract

This thesis presents the design, implementation, and evaluation of an extension to the Java language, ConstJava, that is capable of expressing immutability constraints and verifying them at compile time. The specific constraint expressed in ConstJava is that the transitive state of the object to which a given reference refers cannot be modified using that reference.

In addition to the ability to specify and enforce this basic constraint, ConstJava includes several other features, such as mutable fields, immutable classes, templates, and the `const_cast` operator, that make ConstJava a more useful language.

The thesis evaluates the utility of ConstJava via experiments involving writing ConstJava code and converting Java code to ConstJava code. The evaluation of ConstJava shows that the language provides tangible benefits in early detection and correction of bugs that would otherwise be difficult to catch. There are also costs associated with the use of ConstJava. These are minimized by ConstJava's backward compatibility with Java, and by the high degree of inter-operability of the two languages, which allows for a less painful transition from Java to ConstJava.

This technical report is a revision of the author's Master's thesis, which was advised by Prof. Michael D. Ernst.

Chapter 1

Introduction

An important goal in language design is making it easier for the programmer to specify constraints on code. Such constraints can ease or accelerate the detection of errors and hence reduce the time spent debugging. An example of such a constraint is static type-checking, which is now found in most languages. Another such constraint is the ability to specify that an object is immutable, or another version of such constraint, that it cannot be changed through a given reference.

The Java language [GJSB00] lacks the ability to specify immutability constraints. This paper describes ConstJava, an extension to the Java language that permits the specification and compile-time verification of immutability constraints. ConstJava specifies immutability constraints using the keyword `const`, which is modeled after C++. The language is backwards compatible with Java. In addition, ConstJava code is inter-operable with Java code, and runs on an unmodified Java Virtual Machine.

ConstJava permits the specification of the following constraint: the transitive state of the object to which a given reference refers cannot be modified using that reference. ConstJava does not place any guarantee on *object* immutability. However, if only constant references to a given object exist (a constant reference is one through which an object cannot be mutated), then the object is immutable. In particular, if at instantiation an object is assigned to a constant reference, the object is immutable.

Unlike other proposals for immutability specification, ConstJava provides useful guarantees even about code that manipulates mutable objects. For example, a method that receives a constant reference as a parameter will not modify that parameter, unless the parameter is aliased in a global variable or another parameter. This allows one to specify compiler-verified constraints on behavior of methods, and eases reasoning about and optimization of programs.

I obtained experience with ConstJava by writing code in it, as well as by annotating Java code with `const` to convert it to ConstJava. This experience helped me to design language features for ConstJava to make it more useful and easier to use. In addition, the experience helped clarify the costs and benefits of using ConstJava.

This technical report is organized as follows. Chapter 2 further motivates immutability constraints. Chapter 3 describes the design goals used during the design and implementation of ConstJava. Chapter 4 describes the ConstJava language, chapter 5 discusses the design of ConstJava, and chapter 6 gives its type-checking rules. Then chapter 7 describes the experiments that were performed in order to evaluate ConstJava, while chapter 8 discusses the results of those experiments and evaluates ConstJava in view of both the experiments and the design goals mentioned in chapter 3. Finally, chapter 9 considers related work, chapter 10 discusses possible future research directions, and chapter 11 concludes. Appendix A provides the full language definition for

ConstJava.

Chapter 2

Motivation

Compiler-enforced immutability constraints offer many benefits. As one example, they permit optimizations that can reduce run time by permitting caching of values in registers that would otherwise need to be reloaded from memory. The task of alias analysis is also simplified by immutability constraints.

This technical report focuses on the software engineering benefits of compiler-enforced immutability constraints. The constraints provide an explicit, machine-checked way to express intended abstractions, which eases understanding and reasoning about code by both humans and machines. They also indicate errors that would otherwise be very difficult to track down. This chapter uses a class representing a set of integers (figure 2.1, p. 4) to explain three examples of such benefits, showing enforcement of interface contracts, prevention of representation exposure, and granting clients read-only access to internal data. In addition, this chapter discusses some of the potential costs of using a language extension such as ConstJava.

2.1 Enforcement of contracts

Method specifications describe both what a method must do and what it must not do. For instance, a method contract may state that the method will not modify some of its arguments, as is the case with `IntSet.intersect()`. Compiler enforcement of this contract guarantees that implementers do not inadvertently violate the contract and permits clients to depend on this property with confidence. ConstJava allows the designer of `IntSet` to write

```
public void intersect(const IntSet set) {
```

and the compiler ensures that the method's specification about not modifying `set` is followed.

2.2 Representation exposure

Users of a well-designed module should not be affected by, nor be able to affect, the details of its implementation. Representation exposure occurs when implementation details are accessible to the outside world. Java's access control mechanisms, for example, the `private` keyword, partly address this problem. However, due to aliasing, representation exposures can still happen even if all of the implementation fields are made private.

In the `IntSet` example, the content of the private data member `ints` is externally accessible through the reference passed to the constructor `IntSet(int[])`. The outside code can directly

```
/** This class represents a set of integers. **/
public class IntSet {
  private int[] ints; // the integers in the set with no duplications

  /**
   * This method removes all elements from this that
   *  are not in set, without modifying set.
   **/
  public void intersect(IntSet set) {
    . . . .
  }

  /**
   * Make an IntSet initialized from an int[].
   * Throws a BadArgumentException if there are duplicate
   *   elements in the argument ints.
   **/
  public IntSet(int[] ints) {
    if (hasDuplicates(ints))
      throw new BadArgumentException();
    this.ints = ints;
  }

  public int size() {
    return ints.length;
  }

  public int[] toArray() {
    return this.ints;
  }
}
```

Figure 2.1: A partial implementation of a set of integers.

change the state of `IntSet` objects, which is undesirable. Even worse, outside code can violate the representation invariant and put an `IntSet` object into an inconsistent state, which would cause methods of this object to behave incorrectly. For example, the outside code could put a duplicate integer into the array `ints`, which would cause the method `IntSet.size()` to return an incorrect value.

Representation exposure is a well known problem and there is no good solution in Java to this. The programmer has to do a deep copy of the data passed to the constructor, and if he forgets to do this, subtle and unexpected errors often arise.

In ConstJava, this case of representation exposure would be caught at compile-time. Since the constructor of `IntSet` is not intended to change the argument `ints`, a programmer using ConstJava would write

```
  public IntSet(const int[] ints) {
```

and the compiler would issue an error at the attempt to assign `ints` to `this.ints`, forcing the programmer to do an array copy.

2.3 Read-only access to internal data

Accessors often return some data that already exists as part of the representation of the module. For example, consider the `toArray` method of the `IntSet` class. It is simple and efficient, but it exposes the representation, just as was the case for the constructor. A Java solution would be to copy the array `ints` to a temporary array and return that. In ConstJava, there is a better solution:

```
public const int[] toArray() {
```

The `const` keyword ensures that the caller of `IntSet.toArray()` is unable to modify the returned array, thus permitting the simple and efficient implementation of the method to remain in place without exposing the representation.

2.4 Costs of using ConstJava

Despite the usefulness of the proposed extension, there are some costs associated with using it. As with any type-checking system, ConstJava's type-checking rules (see chapter 6, p. 17) will likely reject some programs that the programmer knows to be safe, but the ConstJava compiler cannot prove safe and hence must reject. This gives a tradeoff between the ability to detect more errors and the flexibility of the programmer's code. In addition, the amount of type information that needs to be carried in the code increases, which is a disadvantage as the code becomes more cluttered. This parallels the usual tradeoff between benefits of static type-checking and the additional clutter resulting from the necessary type declaration or casts.

Another potential problem is interfacing with existing Java code. While Java code is easy to call from ConstJava code, in general it may not be possible for Java code to call ConstJava code. This may be a problem if some part of a given program is automatically generated by a tool such as JavaCC (a Java parser generator, see [VS]) because such automatically generated code will not interface well with the rest of the program, written in ConstJava. This will be a problem for ConstJava users at least until automated tools that generate ConstJava are comparable to those available for Java.

Chapter 3

Design goals

The goal of this project is to design, implement, and evaluate an extension to the Java language for specifying immutability constraints. This extension should be able to resolve the issues discussed in chapter 2. In addition, there were several major design goals for the design and implementation of ConstJava. These are described here; section 8.1 (p. 32) evaluates the ConstJava design based on these goals.

1. The syntax and semantics of ConstJava should be backward compatible with Java, so that every Java program that does not use a ConstJava keyword as an identifier works in ConstJava.
2. The new syntax should fit naturally within the Java language framework. It would also help if the new syntax were similar to that of a syntax of some other existing language (such as C++) that already has immutability constraints, since this familiarity would make it easier for programmers to use ConstJava.
3. The semantics of ConstJava should be a simple extension of those of Java. There should not be many special cases to remember, and the semantics should be easy to understand without needing to read the formal descriptions of the type-checking rules. This requirement should also simplify use of ConstJava.
4. The system should detect as many violations of the immutability constraints as possible at compile time, preferably all of them.
5. The ConstJava compiler should be usable by programmers. It should give reasonable error messages on inputs that violate the ConstJava type-checking rules. Otherwise, there will be no advantage to finding bugs using this system over usual debugging techniques.
6. The above list of design goals omits mention of compile-time efficiency. I decided that efficiency issues are not as important as the main goal of creating a prototype of ConstJava useful for error detection and development of good programming techniques. Making the system efficient before it is even known whether it is useful to achieving these goals is counterproductive.

Chapter 4

The ConstJava Language

The ConstJava language extends Java with explicit mechanisms for specifying immutability constraints and compile-time type-checking to guarantee those constraints. The syntax of the extensions is based on that of C++ (see section 9.1, p. 34 for a detailed comparison).

ConstJava adds four new keywords to Java: `const`[1], `mutable`, `template`, and `const_cast`. The first of these is the keyword used to specify immutability constraints. The other three are additional language features that, while not essential, make ConstJava a more useful language. The keywords are used as follows:

- `const` is used in three different ways:
  1. As a type modifier: For every Java reference type `T`, `const T` is a valid type in ConstJava, and a variable of such a type is known as a constant reference. Constant references cannot be used to change the state of the object or array to which they refer. A constant reference type can be used in a declaration of any variable, field, parameter, or method return type. A constant reference type can also appear in a type-cast. See section 4.1.
  2. As a method/constructor modifier: `const` can be used after the parameter list of a non-static method declaration, to declare that method as a constant method. Constant methods cannot change the state of the object on which they are called. Only constant methods may be called through a constant reference. `const` can also be used immediately after the parameter list of a constructor of an inner class. Such a constructor is called a constant constructor. Non-constant constructors cannot be invoked when the enclosing instance is given by a constant reference; for constant constructors no such restriction exists. See section 4.2.
  3. As a class modifier: `const` can be used as a modifier in a class or an interface declaration. It specifies that instances of that class or interface are immutable. A class or interface whose declaration contains the `const` modifier is referred to as an immutable class or interface. See section 4.3.
- `mutable` is used in a non-static field declaration to specify that the fields declared by this declaration are not part of the abstract state of the object. Such fields are called mutable fields. Mutable fields can be modified by constant methods and through constant references, while non-mutable fields cannot. See section 4.4.
- `template` can be used in a declaration of a method, constructor, class, or interface, to parameterize the declaration based on constness of some type. This can shorten code and remove error-prone duplication. See section 4.5.
- `const_cast` can be used in an expression to convert a constant reference to a non-constant reference. Such casts permit constant references to be used in non-constant contexts that do not actually modify

[1]`const` is already a Java keyword, but is not presently used by Java.

the object. The `const_cast` operator introduces a loophole into the type system; however, as explained in section 10.1, constness of the reference can be enforced at run time, closing this loophole. See section 4.6.

ConstJava is backward compatible with Java: any Java program that uses none of ConstJava's keywords is a valid ConstJava program. Also, ConstJava is inter-operable with Java. As described in section 5.1, any Java code can be called from ConstJava code. ConstJava comes with standard Java APIs, but with field types and method and constructor signatures modified to include information about whether reference types are constant. See section 5.1 (p. 13) for details.

In addition, a special comment syntax allows every ConstJava program to remain a valid Java program. Any comment that begins with "`/*=`" is considered as part of the code by ConstJava. This feature allows the programmer to annotate an existing Java program with ConstJava syntax without losing the ability to use standard Java development tools.

4.1 Constant references

A constant reference is a reference that cannot be used to modify the object to which it refers. A constant reference to an object of type `T` has type `const T`. For example, suppose a variable `cvar` is declared as `const StringBuffer cvar`. Then `cvar` is a constant reference to a `StringBuffer` object; it can be used only to perform actions on the `StringBuffer` object that do not modify it. For example, `cvar.charAt(0)` is valid, but `cvar.reverse()` causes a compile-time error, because it attempts to modify the `StringBuffer` object.

When a return type of a method is a constant reference, the code that calls the method cannot use the return value to modify the object to which that value refers.

Note that `final` and `const` are orthogonal notions in a variable declaration: `final` makes the variable not assignable, but the object it references is mutable, while `const` makes the referenced object immutable (through that reference), but the variable remains assignable. Using both keywords gives variables whose transitive state cannot be changed except through a non-constant aliasing reference.

The following are the rules for usage of constant references (see chapter 6 (p. 17) for further detail). These rules ensure that any code which only has access to constant references to a given object cannot modify that object.

- A constant reference cannot be copied, either through assignment or by parameter passing, to a non-constant reference. In the above example, a statement such as `StringBuffer var = cvar;` would cause a compile-time error.
- If `a` is a constant reference, and `b` is a field of an object referred to by `a`, then `a.b` cannot be assigned to and is a constant reference.
- Only constant methods (section 4.2) can be called on constant references.

ConstJava also allows declarations of arrays of constant references. For example, `(const StringBuffer)[]` means an array of constant references to `StringBuffer` objects. For such an array, assignments into the array are allowed, while modifications of objects stored in the array are not. This is in contrast to `const StringBuffer[]`, which specifies a constant reference to an array of `StringBuffers`, and means that neither array element assignment nor modification of objects stored in the array are allowed through a reference of this type.

A non-constant reference is implicitly converted to a constant one during assignments, including implicit assignment to parameters during method or constructor invocations. A non-constant reference can also be explicitly cast to a constant one by using a type-cast with a type of the form `(const T)`. Here is an example:

```
    const StringBuffer cvar = new StringBuffer();
    StringBuffer var = new StringBuffer();
    cvar = var;                       // OK; implicit cast to const
    cvar = (const StringBuffer) var;  // OK; explicit cast to const
    var = cvar;                       // compile-time error
    var = (StringBuffer) cvar;        // compile-time error
```

4.2 Constant methods and constructors

Constant methods are methods that can be called through constant references. They are declared with the keyword `const` immediately following the parameter list of the method. It is a compile-time error for a constant method to change the state of the object on which it is called. For example, an appropriate declaration for the `StringBuffer.charAt()` method in ConstJava is:

```
    public char charAt(int index) const
```

Constant constructors are constructors that can be called with enclosing instance given through a constant reference. They are declared with the keyword `const` immediately following the parameter list of the constructor. It is a compile-time error for a constant constructor to change the state of the enclosing instance.

Methods and constructors can be overloaded based on whether they are constant. The following two methods are distinct:

```
public void foo() {
}
public void foo() const {
}
```

Similarly, methods and constructors can be overloaded based on whether a parameter is declared as constant. Overloading resolution works much like it does in Java (see [GJSB00]), except that the ConstJava type hierarchy is used in determining method applicability and specificity instead of the Java type hierarchy, and that constant methods or constructors are considered less specific then non-constant ones. For full detail, see the language definition in appendix A.

4.3 Immutable classes

A class or an interface can be declared to be immutable. This means that all of its non-mutable non-static fields are implicitly constant and final, and all of its non-static methods are implicitly constant. In addition, if the class is an inner class, all of its constructors are also implicitly constant. To declare a class or an interface as immutable, `const` is used as a modifier in its declaration.

For an immutable class or interface `T`, constant and non-constant references to objects of type `T` are equivalent, and in particular constant references can be copied to non-constant references, something that is normally disallowed (see end of section 4.1). Subclasses or sub-interfaces of immutable classes and interfaces must be declared immutable.

4.4 Mutable fields

Mutable fields are fields that are not considered to be part of the abstract state of an object by ConstJava. A mutable field of an object O can be changed through a constant reference to O. The programmer declares a given field as mutable by putting the modifier `mutable` in the declaration of the field.

The primary use of mutable fields is to cache results of some computations by constant methods. For example, this situation arises in the ConstJava compiler, where a name resolution method `resolve()` needs to cache the result of its computation. The solution looks somewhat like the following

```
class ASTName {
...
  private mutable Resolution res = null;
  public Resolution resolve() const {
    if (res == null)
      res = doResolve(); // OK only because res is mutable
    return res;
  }
}
```

Without mutable fields, constant methods are unable to cache the results of their work, and consequently ConstJava would force the programmer to either not label their methods as constant, or to take a significant efficiency penalty.

4.5 Templates

ConstJava allows method definitions to be parametrized over the constness of the parameters or of the method itself. This allows the programmer to avoid code duplication. For example, the following two definitions:

```
public static Object identity(Object obj) {
  return obj;
}
public static const Object identity(const Object obj) {
  return obj;
}
```

can be collapsed into one definition using templates (the syntax is defined later in this section):

```
template<o> public static const?o Object identity(const?o Object obj) {
  return obj;
}
```

In addition to defining polymorphic methods, templates are used in class and interface declarations to create parametrized types. A basic example of this is the container class libraries in `java.util`. ConstJava needs two types of container classes, those that contain `Objects`, and those that contain `const Objects`. This is because a `Vector` of `Objects` cannot contain a `const Object`, since its `add` method has the signature

```
public void add(int index, Object obj)
```

and so cannot be called on a `const Object`. On the other hand, a `Vector` of `const Objects`, while capable of containing both constant and non-constant `Objects`, will not permit modification of any `Objects` extracted from the vector. Its `get` method has the signature

```
public const Object get(int index) const
```

Instead of a single `Vector` class, therefore, ConstJava has two classes, and it is much easier to define them using templates. ConstJava's libraries define classes `Vector<>` and `Vector<const>`, to contain non-constant references and constant references respectively, and similarly for other container classes.[2] Templates allow the programmer to write only one version of the `Vector` class, parameterized as follows:

```
template<o>
public class Vector extends AbstractList<const?o>
       implements Cloneable, Serializable {
  ....
}
```

and then use both `Vector<>` and `Vector<const>` in his code.

Because any code that uses templates can be rewritten without templates, templates are a convenience rather than a necessity in ConstJava.

The syntax and semantics of templates are as follows. The keyword `template` is followed by a comma-separated list of distinct variables, called *polymorphic* variables, enclosed in angle brackets. This is followed by a method, constructor, class, or interface declaration. Within such declaration, anywhere where `const` may normally appear, `const?a` can be used for any polymorphic variable `a`. When a template declaration is expanded, a separate declaration is created for each boolean assignment to polymorphic variables. If the declaration declares a class or interface, the name of the class or interface has `<const?v1,const?v2...>` appended to it, where `v1`, `v2`, etc. are respectively the first, second, etc. of the polymorphic variables in the template declaration. Finally, within each generated declaration any occurrence `const?a`, where `a` is a polymorphic variable, is replaced by `const` if `a` is assigned `true` and by empty token sequence if `a` is assigned `false`.

4.6 Casting away of const

The keyword `const_cast` permits casting away `const` from a type. Its introduction into ConstJava is motivated by the fact that sometimes safe ConstJava code gets rejected by the type-checking rules. For example, it is possible that a method is logically constant (i.e., it does not change the state of `this`), yet the compiler cannot prove this fact, and hence the method cannot be declared as constant. For an example of this, see section 7.3.2 (p. 25).

Rather than force the programmer to rewrite such code, ConstJava includes `const_cast`. It allows the programmer to override type-checking rules in any given instance, and so it should be used sparingly.

Formally, the syntax for `const_cast` is

```
const_cast<EXPRESSION>
```

[2]The syntax used in ConstJava may need to be changed for compatibility with GJ [BOSW98] when Java 1.5 comes out.

`const_cast` has no run-time effect, and at compile time it simply converts the type of `EXPRESSION` from `const T` to `T`.

The ConstJava compiler ordinarily guarantees that no mutation can occur through a `const` reference. The presence of `const_cast` enables code to violate that restriction. Section 10.1 (p. 37) anticipates future research in retaining soundness even in the presence of `const_cast`, by inserting run-time checking code to guarantee that even after a `const_cast` operation, the resulting (non-constant) reference is never used to modify the object.

Chapter 5

Language design

This chapter describes three immutability checking issues that have not been handled by previous research, along with how ConstJava addresses them.

5.1 Inter-operability with Java

A major goal during the design of ConstJava was ensuring that ConstJava is inter-operable with Java. The language treats any Java method as a ConstJava method with no `const` in the parameters, return type, or on the method, and similarly with constructors and fields. In other words, since ConstJava compiler does not know what a Java method can modify, it assumes that the method may modify anything.

This approach allows ConstJava to call any Java code safely, without any immutability guarantees being violated. However, in many cases this analysis is over-conservative. For example, `Object.toString()` can safely be assumed to be a constant method. Therefore the ConstJava compiler permits the user to specify alternative signatures for methods and constructors, and alternative types for fields in Java libraries. Ideally, ConstJava would come with all Java APIs annotated with their correct ConstJava signatures. At this time, however, only the following libraries annotated: `java.lang`, `java.io`, `java.util`, `java.util.zip`, `java.awt` and sub-packages, `java.applet`, and `javax.swing` and sub-packages. Library annotation is a time-consuming process, and for that reason future research in automation of this process is anticipated, as described in section 10.2 (p. 38).

The ConstJava compiler reads a special signature file containing these alternative signatures and types. For example, this file contains

```
public String java.lang.Object.toString() const;
```

telling the compiler that the `toString` method is constant. The compiler trusts these annotations without checking them.

Note that as described in section 4.5, container classes in `java.util` are a special case, since they require not just a simple change of signatures, but the creation of a separate container class hierarchy for containing `Objects` versus `const Objects`. This special case is resolved by providing ConstJava source code for container classes, instead of just signature annotations. These container classes are contained in a package named `constjava.java.util`, which should be imported, instead of `java.util`, by every program that wishes to use container classes.

Another special case of a Java API is the class `java.lang.ref.WeakReference`. This class has constructors `WeakReference(Object)` and `WeakReference(Object, ReferenceQueue)`. In ConstJava, however, weak references can be instantiated either from constant or non-constant references, resulting in constant or non-constant weak references respectively. Since constructor invocation cannot result in a constant reference, ConstJava provides the following methods in a special package `constjava.java.lang.ref`.

```
public class ReferenceFactory {
  template<o>
  public static const?o WeakReference weakReference(const?o Object o) {
    ...
  }
  template<o> public static const?o WeakReference
    weakReference(const?o Object o, ReferenceQueue q) {
    ...
  }
}
```

which permits the programmer to construct either constant or non-constant weak references.

While Java methods can be called from ConstJava, Java code can only call ConstJava methods that do not contain `const` in their signatures.

A final inter-operability feature, meant to ease the process of converting Java code to ConstJava code, is the "`/*=`" comment syntax described in chapter 4. This feature lets the programmer convert Java code to ConstJava without losing the ability to compile is as Java code, simply by placing all ConstJava syntax within these special comments.

5.2 Inner classes

The type-checking rules guarantee that constant methods do not change any non-static non-mutable fields of `this`. The inner class feature of Java adds complexity to this guarantee. One must ensure that no code inside an inner class can violate an immutability constraint. There are three places in an inner class where immutability violations could occur: in a constructor, in a field or instance initializer, or in a method. The ConstJava type-checking rules (chapter 6, p. 17) prevent any such violation. This section explains the rules by way of an example.

```
class A {
  int i = 1;
  public void foo() const { // should be unable to change i
    class Local() {
      Local() {
        i = 2;           // changes i
        j = 3;
      }
      int j = ( i = 4 );  // changes i
      void bar() {
        i = 5;           // changes i
      }
    }
    new Local().bar();
  }
}
```

The type-checking rules that deal with inner classes need to deal with the three possibilities shown above:

1. Change in the constructor: Constant constructors (see section 4.2, p. 9) prevent this change. There are two possibilities for a change of `i` in a constructor of `Local`. This change could happen inside a constant constructor, or inside a non-constant one. In ConstJava, a field of the class being constructed accessed by simple name is assignable and non-constant, but otherwise constant constructor bodies type check like constant method bodies. Therefore the first possibility cannot happen. If the constructor of `Local` is labeled as constant, the assignment `i = 2` will not type check, but the assignment to `j` will. The second possibility cannot happen either, since our rules allow only constant constructors to be invoked through a constant enclosing instance. In the example above, if the constructor of `Local()` is not labeled as constant, it cannot be invoked, since the enclosing instance is implicitly `this`, a constant reference inside `foo()`.
2. Assignment in the initializer of `j`: If at least one constant constructor exists in a given class or if an anonymous class is being constructed with constant enclosing instance, the ConstJava compiler treats instance initializers and instance field initializers as if they were in a body of a constant constructor. The second case is necessary because anonymous constructors have an implicit constructor, which is considered a constant constructor if the enclosing instance in constant. This rule prevents modifications to the state of a constant enclosing instance from initializers of inner classes.
3. Change in `bar()`: The rule that `new Local()` must have type `const Local` if the enclosing instance is constant prevents modification of the enclosing instance. If `bar()` is declared as constant, the assignment to `i` inside it will fail to type-check. If `bar()` is not declared as constant, then the call to `bar()` in the example above does not type check, because `new Local()` has type `const Local`.

5.3 Exceptions

An exception thrown with a `throw` statement whose argument is a constant reference should only be catchable by a `catch` statement whose parameter is declared as `const`, because otherwise the `catch` statement would be able to change the exception's state.

In ConstJava, constant exceptions cannot be thrown. Therefore the type-checking system has no hole, but it rejects many safe uses of constant references to exceptions. This restriction has so far caused no difficulty in practice. Two other possibilities for dealing with constant exceptions — wrapping and wrapping with catch duplication — lead to holes in the type system; this section briefly discusses them.

The wrapping approach wraps some exceptions at run-time in special wrapper objects, so that non-`const catch` statements do not catch constant exceptions.

Since ConstJava runs on an unmodified Java Virtual Machine, each wrapper classes should be a subtype of `Throwable`. Since the ConstJava `catch` clause `catch(const Throwable t)` should catch any throwable object, either constant or non-constant, `const Throwable` should be represented as `Throwable` in the underlying Java. Therefore, a natural approach is, for an exception class `E`, to represent `const E` in ConstJava as `E` in underlying Java, and `E` in ConstJava as a wrapper class in Java, say `wE`.

Since `E` is a subtype of `const E` in ConstJava, it would be nice to have `wE` a subtype of `E` in Java. However, since Java does not support multiple inheritance, `wE` cannot be a subtype of `E`, since it has to be subtype of `wP` (where `P` is the parent class of `E`).

The duplication approach simulates `E`'s being a subtype of `const E` by representing a `catch` clause of the form `catch(const E)` by two Java catch clause, `catch(E)` and `catch(wE)`. However, several problems with checked exceptions arise within this framework. Consider the ConstJava statement:

```
    throw new RuntimeException();
```

For this statement to correspond to legal underlying Java code, either `wRuntimeException` must be an unchecked exception class, or it must appear in the `throws` clause of the method that contains that statement. The second possibility does not work, since method overriding does not allow adding checked exceptions. Since `Object.toString()` in Java does not throw any checked exception, ConstJava would be unable to override this method with a method that contains the statement `throw new RuntimeException();`.

The first of these possibilities could work, but it presents several new problems. If `wRuntimeException` and hence, similarly, `wError`, are not subtypes of `wThrowable` in this framework, the ConstJava clause `catch(Throwable t)` now needs to correspond to at least three underlying Java catch clauses, to catch `wThrowable`, `wError` and `wRuntimeException`. Another problem is that `catch(const E e)` corresponds in this framework to two clauses, `catch(wE e)` and `catch(E e)`, in the underlying Java code. There is no guarantee that the corresponding `try` clause actually throws both `wE` and `E`, so if `E` is a checked exception, the two catch clauses are not legal Java code.

The only solution to these problems is to do a full analysis of exception throwing during ConstJava type checking. Due to time constraints, and because I do not believe that a less restrictive treatment of constant exceptions would be an important feature in the ConstJava language, I adopted a more straightforward approach of disallowing throws of constant exceptions.

Chapter 6

Type-checking rules

ConstJava has the same runtime behavior as Java[1]. However, at compile time, checks are done to ensure that modification of objects through constant references, or similar violations of the language, do not occur. Section 6.1 introduces some definitions. Section 6.2 then presents the type-checking rules.

6.1 Definitions

6.1.1 ConstJava's types

ConstJava's type hierarchy extends that of Java by including, for every Java reference type `T`, a new type `const T`. References of type `const T` are just like those of type `T`, but cannot be used to modify the object to which they refer.

Formally, the types in ConstJava are the following:

1. The null type `null`.
2. The Java primitive types.
3. Instance references. If `O` is any class or interface, then `O` is a type representing a references to an instance `O`.
4. Arrays. For any non-null type `T`, `T[]` is a type, representing an array of elements of type `T`.
5. Constant types. For any non-null non-constant type `T`, `const T` is a type.

For convenience in the usage later, we define the depth and the base of a given type. Informally, the depth is just the nesting depth of an array type, while the base of an array type is the type with all array dimensions removed. Formally, for a type `T`, we define:

Depth:

- if `T` is null, primitive, or instance reference, $depth(\texttt{T}) = 0$.
- if $depth(\texttt{T}) = n$, then $depth(\texttt{T[]}) = n + 1$.
- if $depth(\texttt{T}) = n$, then $depth(\texttt{const T}) = n$.

Base:

- if `T` is null, primitive, or instance reference, $base(\texttt{T}) = \texttt{T}$.
- if $base(\texttt{T}) = \texttt{S}$, then $base(\texttt{T[]}) = \texttt{S}$.

[1]Except for possible checks of `const_cast` described in section 4.6.

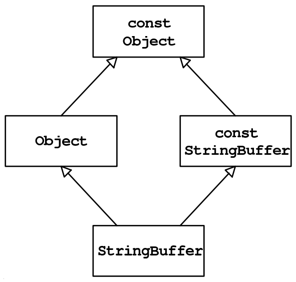

Figure 6.1: ConstJava type hierarchy, including constant and non-constant version of each Java reference type. Arrows connect subtypes to supertypes.

- if *base*(`T`) = `S`, for a constant type S, then *base*(`const T`) = `S`.
- if *base*(`T`) = `S`, for a non-constant type S, then *base*(`const T`) = `const S`.

6.1.2 Type equality and sub-typing

The equality relation is defined on the types as follows:

1. For primitive types, the null type and references to instances of classes and interfaces, two types are equal iff they are the same Java type.
2. `const T` and `const S` are equal iff *depth*(`T`) = *depth*(`S`) and *base*(`const T`) = *base*(`const S`).
3. `T[]` and `S[]` are equal iff `T`, `S` are.
4. For a non-constant type `T`, `T` and `const S` are equal iff `T` and `S` are equal, and `T` is either primitive or is a reference to an instance of an immutable class or interface.

Note that item 2 implies, for example, that `const int[][]` and `const (const int[])[]` are equivalent. In other words, a constant array of array of `int` is the same as a constant array of constant `int` arrays. Item 4 captures the notion that `T` and `const T` are equivalent for immutable types `T`.

Equal types are considered to be the same type. They are interchangeable in any ConstJava program.

A sub-typing relationship (`T` subtype of `S`, written as `T` < `S`) is also defined on types. It is the transitive reflexive closure of the following:

1. `byte` < `char`, `byte` < `short`, `char` < `int`, `short` < `int`, `int` < `long`, `long` < `float`, `float` < `double`.
2. `null` < `T` for any type `T` which is not a primitive type.
3. If `T` and `S` are classes such that `T` extends `S` or interfaces such that `T` extends `S`, or `S` is an interface and `T` is a class implementing `S`, then `T` < `S`.
4. For any non-null types `T` and `S`, if `T` < `S` then `T[]` < `S[]`.
5. For any non-constant non-null type `T`, `T` < `const T`.
6. For any non-constant non-null types `T` and `S`, if `T` < `S` then `const T` < `const S`.
7. For any non-null type `T`, `T[]` < `java.io.Serializable`, `T[]` < `Cloneable`, and `T[]` < `Object`.
8. For any non-constant non-null type `T`, `(const T)[]` < `const T[]`.

An example of the hierarchy relationship is shown in figure 6.1.

6.1.3 Definitions relating to method invocations

These definitions are the same as those in Java, except for the presence of the third clause in the definition of specificity.

Compatibility: Given a method or constructor M and a list of arguments $A_1, A_2, \ldots A_n$, we say that the arguments are compatible with M if M is declared to take n parameters, and for each i from 1 to n, the type of A_i is a subtype of the declared type of the ith parameter of M.

Specificity: Given two methods of the same name or two constructors of the same class, M_1 and M_2, we say that M_1 is more specific than M_2 if the following three conditions hold:

1. M_1 and M_2 take the same number of parameters, say with types $P_1, P_2 \ldots P_n$ for M_1, and $Q_1, Q_2 \ldots Q_n$ for M_2, and for each i from 1 to n, P_i is a subtype of Q_i.
2. The class/interface in which M_1 is declared is a subclass/subinterface of the one where M_2 is declared, or M_1 and M_2 are declared in the same class/interface.
3. Either M_1 is not constant or M_2 is constant (or both).

6.2 Type-checking rules

6.2.1 Programs

A program type checks if every top-level class/interface declaration in the program type checks.

6.2.2 Class/Interface declarations

A class or interface declaration type checks if all of the following hold:

1. (a) The class/interface is immutable and each of the methods declared in any of its superclasses or superinterfaces is private, static, or constant, and each of the fields declared in any of its superclasses is private, static, mutable, or both final and of a constant type, or
   (b) the class or interface is not immutable, and neither is its direct superclass or any of its direct superinterfaces.
2. No two fields of the same name are declared within the body of the class/interface.
3. No two methods of the same name and signature or two constructors of the same signature are declared within the body of the class/interface. Signature includes the number and the declared types of parameters, as well as whether the method is constant.
4. Every declared field, method, member type, instance initializer, and static initializer of the class or interface type checks.

6.2.3 Variable declarations

For a field or local variable declaration of type T:

- If it does not have an initializer, it type checks.
- If it has an initializer of the form "` = `E" for an expression E, it type checks iff the assignment of the expression E to a left hand side with type `T` would type check.
- If it has an initializer of the form "` = { `$I_1, \ldots I_k$` } `", where I_k are initializers, it type checks iff `T = S[]` or `T = const S[]` for some type `S`, and the declaration `S v = `I_k or `const S v = `I_k respectively would type check for every k between 1 and n.

6.2.4 Method declarations

A method, constructor, instance initializer, or static initializer type checks if every expression, local variable declaration, and local type declaration in the body of the method, constructor, instance initializer, or static initializer type checks.

6.2.5 Expressions

Each expression has a type and a boolean property called assignability associated with it. An expression is type checked recursively, with all subexpressions type checked first. If the subexpressions type check, then their types and assignabilities are used to deduce whether it type checks, and if so, to deduce its type and assignability. Otherwise, the given expression does not type check.

The rules for type checking an expression given types and assignabilities of subexpressions are given below. For brevity this section gives only the rules that are substantially different from those in Java; for a full set of type checking rules, refer to the ConstJava language definition in Appendix A.

- The rules for type checking assignments are the same as in Java, except if an expression that is determined to be not assignable according to the rules below appears as the lvalue of an assignment, the assignment expression does not type check.

  The type of any assignment expression that type checks is the same as the type of the lvalue, and the expression is not assignable.

- `(T)`A: in addition to the Java rules, a type cast must not cast from a constant type to a non-constant type in order to type check. The type of a cast exception is `T` and the expression is not assignable.

- `const_cast<`A`>`: always type checks. If the type of A is non-constant, this expression is of the same type. If the type of A is `const S[]` for some `S`, then the type of this expression is `(const S)[]`. If the type of A is `const S` where $depth$(`S`) $= 0$, the type of this expression is `S`.

- `this` does not type check in a static context; in a non-static context `this` has type `C` if `C` is a class and `this` appears inside a non-constant method or a non-constant constructor of C; `this` has type `const C` inside a constant method or a constant constructor of `C`. `this` is not assignable.

- `NAME.this` type checks if it occurs in a non-static context in a method or constructor of a class `I`, and `NAME` names a class `C` for which `I` is an inner class. The type of the expression is `C` unless it appears inside a constant method or a constant constructor of `I`, in which case the type is `const C`. This expression is not assignable.

- Class instance creation expression:

  - If the enclosing reference is constant, only constant constructors are eligible, otherwise, all constructors are eligible.
  - The expression type checks if there is a most specific accessible eligible constructor compatible with the arguments to the class instance creation expression.
  - If the class being instantiated is `T`, the type of the expression is `const T` if the enclosing reference is a constant reference, and `T` otherwise.

  A class instance creation expression is never assignable.

- A`[`E`]` type checks if E is of integral type and A is of type `T[]` or `const T[]` for some type `T`; the type of the expression is respectively `T` or `const T`. The expression is assignable in the first case, and not assignable in the second.

- Field access expression: Let `T` be the declared type of the field. Then:

  - If `T` is a constant type, or the field is accessed through a non-constant reference, or the field is a mutable or static field, the type of the expression is `T`.
  - Otherwise the type of the expression is `const T`.
  - The expression is assignable if the field is mutable or static, or if it is not accessed through a constant reference.

- Method invocation expression:
  - If the invoking reference is constant, only constant methods are eligible.
  - If there is no invoking reference, only static methods are eligible.
  - Otherwise, all methods are eligible.
  - The expression type checks if there is a most specific accessible eligible method compatible with the arguments to the method invocation. The type of the expression is the declared return type of such method.

  A method invocation expression is never assignable.

Chapter 7

Experiments

In order to evaluate ConstJava, I wrote code in ConstJava and also annotated existing Java code with `const`. Writing code in ConstJava provides experience with the language the way many people would use it. In addition, it permits greater flexibility in working around type-checking errors than working with existing code does, and it can be more beneficial than annotation of existing code, since it provides earlier indication of errors. On the other hand, annotation of existing code gives a more quantifiable experience, since it is possible to track the amount of time spent annotating, the number of problems with original code found, etc. Also, it permits evaluation of ConstJava on code written by other people. Finally, it permits evaluation of how ConstJava fits with the existing practice of code written by programmers who did not have `const` in mind while coding.

Figure 7.1 (p. 23) displays statistics about the experiments. Section 7.1 describes the experiments in writing container classes in ConstJava from scratch. The rest of the sections describe annotation experiments, with section 7.2 describing the annotation process and the programs that were annotated, section 7.3 describing the results of the annotation, including the classification of the errors detected by the annotation, and section 7.4 describing in detail the more important of these errors.

7.1 Container classes written from scratch

As explained in section 4.5 (p. 10), the container classes in `java.util` cannot be used with Const-Java, and a parameterized version of the container classes must be written. Because of this, and also to gain experience with ConstJava, I wrote many of the container classes in the `java.util` package from scratch in ConstJava, namely the classes `Collection`, `AbstractCollection`, `Set`, `AbstractSet`, `AbstractList`, `AbstractSequentialList`, `Iterator`, `List`, `ListIterator`, `ArrayList`, `Vector`, `LinkedList`, `HashSet`, `Map`, `AbstractMap`, and `HashMap`.

7.2 Annotation of Java code

The methodology for annotating Java programs proceeded in three stages. During the first stage of the annotation ("Signature" near the bottom of figure 7.1), I read the documentation and the signatures of all public and protected methods in the program, and marked the parameters, return types, and methods themselves with `const`. For example, if the documentation for a given method specified that the method does not modify its parameters, every parameter would be marked with `const`, and if the documentation stated that a given parameter may be modified, then that

	New code	Annotated Java code			
Program	`java.util`	GizmoBall	Daikon	ConstJava	`java.util`
Code size					
classes	52	172	1593	466	38
methods	359	633	4455	1007	163
lines	2687	15476	100967	15633	4795
NCNB lines	1828	9222	68889	9394	2134
Annotations					
`const`	837	657	5869	2794	611
`mutable`	27	45	103	64	6
`template`	94	13	777	79	65
`const_cast`	2	6	97	7	14
Code errors	N/A			N/A	N/A
Documentation		2	1		
Implementation		1	19		
Bad Style		3	3		
ConstJava problems					
Inflexibility		3	24		
Incompleteness		1	2		
Annotation errors					
Signature		11	124		
Implementation		31	486		
Library		3	18		
Time (hh:mm)	N/A			N/A	N/A
Signature		2:40	5:30		
Implementation		4:30	7:35		
Type check, fix errors		6:10	55:05		

Figure 7.1: Programs written in ConstJava or converted from Java to ConstJava. The number of classes includes both classes and interfaces. "NCNB lines" is the number of non-comment, non-blank lines. Section 7.3 explains the error categories. The beginning of section 7.2 explains the time categories. Errors and time were recorded for only two of the programs.

parameter would not be marked with `const`.

The second stage of the annotation ("Implementation" in figure 7.1) was to annotate the private signatures and the implementations of all methods. Finally, the third stage ("Type check, fix errors") involved running the compiler on the resulting program, and considering and correcting any type checking failures.

7.2.1 GizmoBall annotation

The GizmoBall project is the final project in a software development class at MIT (6.170 Laboratory in Software Engineering). It was written in one month by me and three other people, with about a third of the code written by me. The program uses Java to implement an extensible pinball game.

During the third stage of the annotation experiment, the invocation of the compiler on the annotated project found 55 different type-checking errors that are tabulated in figure 7.1. Section 7.3 describes the categories and gives examples of errors in each category. Section 7.4 gives a detailed explanation of each of the errors in the more important categories.

7.2.2 Daikon annotation

Daikon is a tool for dynamic detection of invariants in programs [Ern00, ECGN01]. I had no previous experience with Daikon (not even as a user).

The Daikon project contains approximately 100,000 lines of code. Again, the time spent on this experiment and the results of this annotation appear in figure 7.1.

7.2.3 The annotation of the compiler

Chronologically, the first major annotation experience was annotating the ConstJava compiler itself, which is about 15,000 lines is size. No log of this experience was kept, and as the annotation was intermixed with changes in the language and bug fixes in the compiler, this experience could not provide much information about the ease of use of ConstJava or about its utility.

7.2.4 Container classes annotated from Sun source code

In addition to writing some parameterized container classes from scratch (see section 7.1), I annotated the Sun JDK 1.4.1 reference implementation of classes `Arrays`, `SortedSet`, `SortedMap`, `TreeSet`, `TreeMap`, and `Stack`. This code was about 4800 lines long, but I did not keep a log of time spent or of the type-checking failures encountered. No bugs in `java.util` were discovered during the annotation process, but one instance of bad code style was discovered (see section 7.4).

7.3 Error classifications

This section describes what kind of errors were put into each of the categories used in figure 7.1, together with some examples of such errors.

7.3.1 Code errors

These errors are problems with the original Java program that were discovered during the annotation and type checking process. Deciding to which one of the three sub-categories a given error belong is inherently subjective, involving a judgment call on the part of the annotator. The three sub-categories are:

Documentation: This category represents errors in the documentation of a class or a public or protected method, causing an incorrect annotation. An example is when the documentation of a method states that the method does not modify a given parameter, when the method does modify the parameter.

Implementation: This category represents bugs in the original code found during the type checking of the annotated code. In the GizmoBall project, the bug was a representation exposure caused by a method returning a reference to private data of a given class. It was fixed by adding a `const` on the return type of the misbehaving method. An example of a bug found in the Daikon project was a method that sorted its input array before computing some statistics about the array. This bug was fixed by rewriting the method to do an array copy first.

Bad Style: This category represents errors caused by bad style of coding in the original project. While the code that causes these errors does not, to my knowledge, contain actual bugs, it could have easily been written in a better style that would not only allow the code to type check, but also made the program easier to maintain and debug. An example of such code is recalculating the size of gizmos that are displayed during the GizmoBall screen during each `paint()` call. A better alternative, one which would also type check under ConstJava's rules, would be to do these recalculations only when the window size changes.

7.3.2 ConstJava Problems

These errors are caused by weaknesses either in the ConstJava language or in the ConstJava compiler.

Inflexibility of the Language: This category represents safe code rejected by the compiler's conservative analysis. An example of such inflexibility is given in the following code snippet:

```
public class BuildDriver implements ActionListener, MouseListener {
  private JFrame jf;
  ....
  private void askForLoad() const {
    final JDialog jd = new JDialog(jf, true);
    // get the name of file to load using dialog box jd
  }
  ...
}
```

The call to the `JDialog` constructor does not type check, since that call cannot take a constant reference to a `JFrame`. A call to `jd.getOwner` might later return a reference that would alias `jf`, allowing the frame referenced by `jf` to be modified. This never happens in the `askForLoad()` method, and `jd` does not escape that method, but those facts are beyond the compiler's static analysis, and so the compiler rejects this safe code.

Incompleteness of the compiler: This category represents instances where reflection was used in the original program. Since reflection is not yet supported by the compiler, such code does not type check. I corrected these errors by using `const_cast` (see section 4.6).

7.3.3 Annotation Errors

This category represents errors caused by mistakes committed by me during the annotation process. Often, these were due to unfamiliarity with the code or to poor documentation. The *Signature Misannotation* category represents errors due to an incorrect annotation of a signature of a public or protected method during the first stage of the annotation. The *Implementation Misannotation* category represents the errors caused by an incorrect annotation of the type of a private field, the signature of a private method, the type of a local variable, or a type used in a cast expression. The *Library Misannotation* category represents the errors caused by an incorrect annotation of a library method, for example an AWT method. (Use of ConstJava requires annotation of Java libraries such as AWT and Swing; these libraries are now provided with ConstJava. The library annotation time is not included in figure 7.1, but some errors in the library annotations were discovered while annotating client code.)

7.4 Errors

In order to give a feel for the use of ConstJava compiler, this section describes each of the more important errors encountered during the experiments, together with how they were resolved.

7.4.1 Documentation

I found two documentation errors in the GizmoBall project.

1. In the documentation of the class `gb.gizmos.GSquare`. This class was incorrectly documented as immutable. For example, the location of a GSquare can be changed through its APIs. This was fixed by updating the documentation appropriately.
2. The documentation for the method `gb.drivers.BuildDriver.init()` states it does not modify parameter `GizmoTranslator gt`. However, the method aliases `gt` into the state of the `BuildDriver` that is being initialized, and therefore `gt` cannot be declared constant. This was also fixed by updating the documentation appropriately.

There was one documentation error found during the annotation of Daikon:

1. The documentation of `daikon.FileIO.process_sample()` method did not document the counterintuitive fact that the second parameter, `ValueTuple vt`, is mutated by a side effect in this method. To fix this bug, the documentation was updated to indicate this fact.

7.4.2 Implementation

There was one implementation bug discovered in the GizmoBall project:

1. The method `gb.gizmos.GAbsorber.releaseVelocity()` was written simply as

   ```
   public GBall.Velocity releaseVel() {
     return releaseVelocity;
   } // releaseVelocity
   ```

   which permits the caller to directly modify the state of GBall.Velocity object that is returned. This error was fixed by annotating the code in ConstJava as

   ```
   public const GBall.Velocity releaseVel() const {
     return releaseVelocity;
   } // releaseVelocity
   ```

   thus ensuring that outside code cannot modify the returned object.

In the Daikon project, I discovered 19 implementation bugs during the annotation experiment:

1. The method

   `utilMDE.MathMDE.missing_numbers()`

   takes an array and computes some information about it. It should not modify the input array, and its documentation never states that it does. Nevertheless this method sorts its input array before doing any computations. This bug was fixed by rewriting the method to clone the array first, before sorting it and doing other computations.
2. The method

   `utilMDE.MathMDE.nonmodulus_strict()`

   similarly computes some information about its input array, and it also changes its input array while doing so. This was fixed in a similar way using cloning.

3. The inner class `MathMDE.MissingNumberIterator` has a representation exposure in one of its constructors, where the input array is assigned directly to a private field. This was fixed by cloning the array before doing the assignment.

4. The method `daikon.VarInfo.getDerivedParam()` returns a cached VarInfo, that may then get modified by outside code. This is a representation exposure. This was fixed by marking the return type of the method with `const`.

5. The method

   `daikonv.inv.FeatureExtractor.printC5DataOutput()`

   takes as input some vectors in order to print out their content, modifies one of the vectors. The bug was reported to the authors of Daikon. The correct way to fix it would be to rewrite the code.

6. The constructors for class `daikon.ValueTuple` do not clone their input arrays before assigning them to internal state. The representation exposure was reported to the authors of Daikon. The correct way to fix it would be to rewrite the code.

7-8. The method

```
  PrintInvariants.print_invariants(const PptTopLevel ppt,
                                   PrintWriter out,
                                   const PptMap ppt_map)
```

modifies its argument, `ppt`, by calling `ppt.simplify_variable_names()`. This is incorrect, since the purpose `print_invariants` is to print out data, and the side effects are nowhere specified by the method's documentation. A similar bug exists in

`daikon.tools.ExtractConsequent.extract_consequent_maybe()`

These bugs were reported to the authors of Daikon. The correct way to fix them would be to rewrite the code.

9-11. The documentation for `concat()` methods in `utilMDE.ArraysMDE` states that these methods always return a new array. This is not the case. The concat method concatenates two arrays. If one of the arrays is null, then the method would return the other array. There are three `concat()` methods in `ArraysMDE` class, and therefore there were three such bugs. These bugs were fixed by rewriting the methods to always return a new array.

12-19. The method `daikon.Invariant.isObviousStatically_SomeInEquality()` returns an array that is part of its state. This is a representation exposure, and was fixed by cloning the array before returning it. An analogous bug was found and fixed within the method `daikon.Invariant.isObvious-Dynamically_SomeInEquality()`, and in the methods corresponding to the above two methods in some subclasses of `Invariant`. A total of eight such bugs were found and fixed.

7.4.3 Bad Style

There were three errors caused by bad coding style in the GizmoBall project:

1. The method `JGizmoBoard.paintComponent(Graphics g) const` resizes all gizmos in the gizmo board according to the current size of the board. A better alternative, one which would also type check under ConstJava's rules, would be to do these recalculations only when the window size changes. This problem was fixed by using a `const_cast` (see section 4.6). Note that once dynamic `const_cast` described in section 10.1 (p. 37) is implemented, this code will have to be rewritten, since this particular use of `const_cast` is not run-time safe. The `paintComponent()` method here does change the component's state, which is illegal.

2-3. `GameArea.delGizmo()` is a method that deletes a gizmo from the game area. In this method, a special case arises when the gizmo that is deleted is an absorber. Absorbers are gizmos that can contain captured balls in them; the method checks for any such balls and releases them before deleting the absorber. This makes the `delGizmo()` method fail to type check, since releasing balls modifies both the absorber and the balls being released. `delGizmo()` is not a very clean method; it would be better to write the `delGizmo()` method without the special case, and have a separate, non-constant operation of releasing the balls from the absorber. This would prevent the innocuous operation of gizmo deletion from being able to affect change on other gizmos in the game, which is a surprising side effect of the `delGizmo()` method, especially since its documentation never lists this side effect. The method `GameArea.addGizmo()` adds a gizmo to the game area. Similarly to `delGizmo()`, this method has the special case side effect of capturing any balls that overlap with the gizmo if the gizmo is an absorber. Both of the above errors were dealt with by simply updating the documentation and changing the signatures of the respective methods to reflect that they have potential side-effects.

I found three cases of bad style of coding in Daikon during the annotation:

1. In class `utilMDE.UtilMDE`, the method

   ```
   public static boolean canCreateAndWrite(const File file)
   ```

   tests whether `file` can be created by calling `file.createNewFile()` and `file.delete()`. These calls fail to type-check, since they modify the file. A cleaner way of doing this is to test whether the directory containing the file is writable. This code was rewritten to do just that.

2. The class `PptSlice` has a public field `po_lower` meant for read-only access, and a private field `private_po_lower` meant for read and write use by the implementation of the class. Unfortunately, the method `daikon.PptSlice.flow_and_remove_falsified()` uses `po_lower` in place of `private_po_lower`, and modifies its state. This is clearly in bad style. To fix this problem, `private_po_lower` was used in this method instead.

3. The method

   ```
   public boolean include(const Invariant invariant) const
   ```

   changes a state of an object, does some tests on it, then changes the state back. While the method leaves the object unmodified, this is in bad style, and is forbidden by ConstJava compiler. For this problem, `const_cast` was used to force type checking to succeed. Note that once dynamic `const_cast` described in section 10.1 (p. 37) is implemented, this code will have to be rewritten, since this particular use of `const_cast` is not safe at run-time.

I also discovered a bad style error in `java.util.TreeMap`'s code. This class has a method that takes an `Iterator` parameter; this `Iterator` iterates either over keys or over entries in the map, depending on the value of a different parameter. It would be preferable to have two separate methods that returned the two different iterators. In ConstJava there is no correct typing of the `Iterator` parameter. The `Iterator` over entries is typed as `Iterator<>`, while `Iterator` over keys is typed as `Iterator<const?k>`, where `k` is the polymorphic variable for the constness of keys in the `TreeMap`.

7.4.4 Inflexibility

In the GizmoBall project, there were three inflexibility errors.

1-2. The first inflexibility was presented as an example in section 7.3.2 (p. 25) and is repeated below:

```
public class BuildDriver implements ActionListener, MouseListener {
  private JFrame jf;
  ....
  private void askForSave() const {
    final JDialog jd = new JDialog(jf, true);  // ERROR
    ...
  }
}
```

The `JDialog` constructor takes a reference to enclosing frame, which is not permitted to be constant, because of possibility of call to `jd.getParent()` to extract the enclosing frame and subsequently modify it. However, in this particular case, the `askForSave()` method never calls `jd.getParent()`, and as the reference `jd` does not exist outside of `askForSave()`. Therefore `askForSave()` is a constant method, but the ConstJava compiler cannot prove that fact. This error was fixed by using `const_cast` to force type checking:

```
    final JDialog jd = new JDialog(const_cast<jf>, true);
```

The method `BuildDriver.askForLoad()` had a similar problem, which was also fixed using `const_cast`.

3. The field `GameOpts.FONT` is declared as:

```
public static final Font FONT =
  new Font("Times New Romans", Font.PLAIN, 10);
```

During the annotation, `FONT` was declared as `const`. Unfortunately, a large number of Swing components in the GizmoBall project have their font set to `FONT`, and the compiler cannot guarantee that no piece of code exists that retrieves this font from one of them and modifies it. Therefore, annotation of this field as `const` could not type check. This problem was fixed by simply dropping the `const` annotation on this field.

There were 24 inflexibility errors in Daikon:

1. The `Ast.getParameters()` method constructs temporary objects whose constructors need to take that need to take non-constant references as parameters, but `getParameter()` passes constant references to these parameters. Just as in the inflexibility error involving `JDialog` constructor in the GizmoBall project, the temporary objects, if they were ever accessible from the outside, could be used to modify the objects referred to by these references, but the temporary objects cannot be accessed from the outside, so `getParameter()` method is safe. This is similar to the `JDialog` constructor problem in GizmoBall's `askForSave()` method that was described above. Just as there, the temporary objects could potentially be used to modify the objects referred to by the references passed to their constructor, but since these temporary objects cannot be accessed from the outside, the `getParameter()` method is safe. This error was fixed using `const_cast`.

2. The method `PptTopLevel.addViews()` has a `const` parameter. It creates an object, assigns it to that parameter, and then does modifications on it. Of course, this is safe, since the object originally referred to by the parameter is unmodified. Unfortunately, the ConstJava compiler cannot accept this. To fix this, the method was rewritten to use a local variable instead of the parameter to hold the newly created object.

3. The method `write_serialized_pptmap` in class `daikon.FileIO` takes a constant `PptMap` and serializes it. However, before serializing it, it must put the `PptMap` into a `SerialFormat` object, whose constructor takes a non-constant `PptMap`. This is safe, since the `SerialFormat` object is not used in any way other than getting serialized. To satisfy the ConstJava compiler, this was fixed using `const_cast`.

4. The class `daikon.VarInfoName` is immutable, and also contains `readResolve()` method that needs to fix up some of the fields upon de-serialization. Unfortunately, fields of an immutable class are

read-only from anywhere except inside a constructor, so this does not type check. In order to force type checking, `const_cast` was used. Note that once dynamic `const_cast` described in section 10.1 (p. 37) is implemented, this code will have to be rewritten, since this particular use of `const_cast` is not safe at run-time.

5. In `daikon.VarInfoName`, a nested class `IOAQuantification` was declared as immutable. This caused its constructor to fail type checking, since the constructor code looked similar to this:

```
public const static class IOAQuantification {
  private VarInfoName[] setNames;

  public IOAQuantification(const VarInfo[] sets) {
    ....
    setNames = new VarInfoName[sets.length];
    for(int i = 0; i < sets.length; i++)
      setNames[i] = sets[i].name;  // ERROR
    ...
  }
}
```

Unfortunately, the array `setNames` is implicitly final and constant, since `IOAQuantification` is immutable. Therefore it cannot be assigned into as above. This is inflexibility in the language, of course, since the constructor should be able to initialize the `setNames` field. In order to work around this problem correctly, a temporary array needs to be created and initialized, and then assigned to `setNames`, as follows:

```
  public IOAQuantification(const VarInfo[] sets) {
    ....
    (const VarInfoName)[] _setNames = new VarInfoName[sets.length];
    for(int i = 0; i < sets.length; i++)
      _setNames[i] = sets[i].name;  // FINE
    setNames = _setNames;
    ....
  }
```

6-7. In class `daikon.inv.unary.sequence.OneOfScalar`, the abstract state of the object consists of a set of numbers. These are stored in an array `elts`. The `min_elt()` and `max_elt()` methods perform constant operations on the class `OneOfScalar`. Unfortunately, they sort `elts`, which modifies the representation, though not the abstract state, of `OneOfScalar`. The ConstJava compiler rejects this. The `elts` field cannot be declared as mutable, because it is part of the state of `OneOfScalar` objects, even though sorting it does not change that state. There does not seem to be a good way to fix this problem without a lot of code rewriting. During the annotation, I forced type checking of this code using `const_cast`. Note that once dynamic `const_cast` described in section 10.1 (p. 37) is implemented, this code will have to be rewritten, since this particular use of `const_cast` is not safe at run-time.

8-11. The class `daikon.derive.Derivation` contains a method `getVarInfo()`, whose simplified version is shown below:

```
public const VarInfo getVarInfo() const {
  if (this_var_info == null) {
    this_var_info = makeVarInfo();
    // const_cast is OK, since this_var_info is returned as const
    this_var_info.derived = const_cast<this>;
    ....
  }
```

```
    return this_var_info;
  }
```

Here `this_var_info` is a cache field to save the `VarInfo` that is computed by this method. The line containing `const_cast` above aliases the state of `this` into the state of the cached `VarInfo`. This is safe in this case, since the outside code will only have access to a constant reference to this VarInfo, and so cannot modify the reference to `this` aliased inside its state. However, the ConstJava compiler cannot prove this fact, and so rejects the above code unless the `const_cast` is used. Analogous inflexibility errors occurred in `getVarInfo()` methods of `BinaryDerivation`, `TernaryDerivation` and `UnaryDerivation`. As shown in the snippet above, these errors were fixed by using `const_cast`.

12-24. In various places in Daikon, code snippets similar to the below occurred:

```
  const A a = new A();
  foo(new A[] { a }); // ERROR
```

which fails type checking according to ConstJava rules, since the array created by `new A[]` is typed as `A[]`, so cannot contain a `const A`. There were 13 cases of this in Daikon, and they had to be fixed by rewriting the code similarly to the following:

```
  const A a = new A();
  const A[] as = { a }; // OK
  foo(as);
```

which is legal. One should note that use of `const_cast`, as in:

```
  const A a = new A();
  foo(new A[] { const_cast<a> });
```

would also work, and that under dynamic `const_cast` checking described in section 10.1 (p. 37), this code would work fine unless `foo()` actually does some modification to the object referenced by `a`.

Chapter 8

Discussion

This chapter first discusses the design and implementation of ConstJava and its compiler in view of the goals listed in chapter 3. Then it discusses the utility of ConstJava based on the experiments described in chapter 7.

8.1 Achievement of design goals

This section reviews the design and implementation of ConstJava compiler in view of the goals listed in chapter 3.

1. The syntax and semantics of the new language are backward compatible with existing Java. Every Java program that does not use a keyword of the extended language as an identifier works in the extended language. Java code is directly callable from ConstJava. However, Java libraries usually need to be annotated before they can be useful in ConstJava programs. This annotation process is lengthy, but only needs to be done once. Further research needs to be done in automating it, thus making ConstJava even easier to use with old Java code. See section 10.2 (p. 38) for discussion of future research directions in this area.
2. The ConstJava syntax fits in well with Java syntax, and all of the extension syntax is taken from C++, making ConstJava easy to learn.
3. The semantics of ConstJava are the same as that of Java in almost every case. A ConstJava program that type checks according to ConstJava rules has the same behavior as the corresponding Java program with all extension syntax removed. The only exception is in method overloading; ConstJava permits overloading of methods based on whether the method or a parameter of the method is constant. This means that potentially many ConstJava methods correspond to the same Java method.
4. The system does detect all violations of the immutability constraints at compile time, except for unsafe usage of `const_cast`. Unsafe usage of `const_cast` is presently not detected, and detecting it at run time is future work (section 10.1, p. 37).
5. The ConstJava compiler is reasonably usable by programmers. It gives sensible error messages on inputs that violate the rules of ConstJava. The error behavior may not yet be as good as that of a commercial Java compiler, but it is sufficient for me to have worked successfully with the ConstJava compiler for several months.
6. Since compile-time efficiency was not considered a priority, the ConstJava compiler is rather slow and takes a lot of memory. Type checking of Daikon, for example, requires 200MB of memory. As described in chapter 3, this is primarily an engineering, rather than scientific, issue.

8.2 Evaluation results

The most interesting type checking errors are implementation errors, documentation errors, bad style errors, and inflexibility errors. The first three of these categories are the errors/problems in the original program that ConstJava helps to solve, while the fourth category is the cost that is incurred by using ConstJava.

The experiments demonstrate that ConstJava catches certain errors. In the GizmoBall experiment, one real bug (that had survived extensive testing) and three instances of bad style of coding were caught. Even more bugs might have been caught at compile time if ConstJava had been used from the beginning of the project. The instances of bad style of coding that were caught in the process are also a benefit of ConstJava language. Forcing programmers to write code that is less convoluted would make code maintenance and debugging easier. Catching three errors in documentation is also certainly a benefit. Finally, another benefit was an efficiency improvement of GizmoBall's code. In addition to one instance of a representation exposure being caught by the ConstJava compiler, several other methods which correctly dealt with the representation exposure problem through data copying were made more efficient by eliminating the copying and simply declaring the return type of the method to be a constant reference.

These benefits are reinforced by the Daikon experiment, where the annotation process discovered one documentation error, 20 bugs, and three instances of bad style coding, and by the annotation of a portion of `java.util`, where one instance of bad style of coding was discovered with the help of ConstJava.

The costs of ConstJava are two-fold. Firstly, extra time must be devoted in order to use its features. This time cost should be smaller if a software project is written in ConstJava from scratch, but time would still need to be devoted to thinking about whether a given reference is constant or not, or whether a given class is immutable or not, etc., thus increasing development time. Since programmers must make such decisions regardless, the cost of adding annotations should be slight. Secondly, any conservative compile-time analysis (particularly a flow-insensitive type analysis) rejects certain safe code. There were 27 instances of this problem in 116,000 lines of annotated Java code.

Comparing the costs to the benefits, however, we can come to the conclusion that the costs are outweighed by the benefits. The extra time necessary for annotation was small by comparison with the original development time. Based on my experience with writing ConstJava code, this extra effort would have been even smaller if ConstJava had been used in the project from the start. While there are cases when safe code is rejected by the ConstJava compiler, they are fewer in number than the bugs, documentation errors, and badly written code detected thanks to ConstJava. This is true for an already completed project; for a project under development, the number of bugs caught through type checking instead of usual run-time testing would probably be significantly larger, saving the programmer a lot of debugging time. Also, the `const_cast` feature of ConstJava allows the programmer to force the ConstJava compiler to accept these instances of safe code that would otherwise be rejected by the compiler.

Chapter 9

Related work

ConstJava shares similarities with some other languages that enable the specification and checking of immutability constraints. Also, some previous research considers introduction of immutability constraints into the Java language.

9.1 C++

C [KR88] and C++ [Str00] provide `const` keywords for specifying immutability. The most notable example is C++. When `const` was added to C++, it was added for the same reason as one of the main reasons for development of ConstJava — in order to be able to specify immutability in APIs:

> The proposal focused on specifying interfaces rather than on providing symbolic constants for C. Clearly, a readonly value is a symbolic constant, but the scope of the proposal is far greater. [Str00]

ConstJava uses the same syntax for immutability specification (`const`, `mutable`, `template`, `const_cast`) to make its use easier for C++ programmers, much as Java adopts C++'s syntax for other language constructs.

Because of numerous loopholes, the `const` notation in C++ provides no guarantee of immutability even for accesses through the `const` reference. First, it is possible to use an ordinary cast to remove `const` from a variable. Second, C++'s `const_cast` may also be applied arbitrarily and is not dynamically checked; while the ConstJava `const_cast` is not yet dynamically checked, future work on the language forsees development of dynamically checked `const_cast`. The `const_cast` operator was added to C++ to discourage use of C-style casts, accidental use of which may convert a constant pointer or reference to a non-constant one. Third, because C++ is not a safe language, it is possible to (mis)use unions, varargs (unchecked variable-length procedure arguments), and other type system weaknesses to convert a const reference into a non-const one. For example:

```
void changeThroughConstPointer(const int* cpi) {
  union u {
    int *pi;
    const int* cpi;
  } a;
  a.cpi = cpi;
  a->pi = 3;
}
```

In the above code, the function `changeThroughConstPointer` succeeds in changing the value of the integer that `cpi` points to, even though `cpi` was declared as `const int*`.

C++ permits the contents of a constant pointer to be modified (constant methods protect only the local state of the enclosing object). To guarantee transitive non-mutability, an object must be held directly in a variable rather than in a pointer. However, this precludes sharing, which is a serious disadvantage. Additionally, whereas C++ permits specification of `const` at each level of pointer dereference, it does not permit doing so at each level of a multi-dimensional array.

By contrast to C++, ConstJava requires use of `const_cast` rather than ordinary cast to cast away const; `const_cast` can be dynamically checked; since Java is a safe language, unions and the like cannot be used to subvert the type system. Java does not distinguish references from objects themselves, and ConstJava permits mutability of each level of an array to be individually specified and checked. ConstJava also supports aspects of Java that do not appear in C++, such as nested classes.

9.2 Proposals for Java

Many other researchers have noticed that Java lacks the `const` operator that is so useful in C and C++ (despite its shortcomings).

Similarly to ConstJava, JAC [KT01] has a `readonly` keyword indicating transitive immutability, an implicit type `readonly` T for every class and interface T defined in the program, and a `mutable` keyword. (JAC actually provides a hierarchy (readnothing < readimmutable < readonly < writeable.) The implicit type `readonly` T has as methods all methods of T that are declared with the keyword `readonly` following the parameter list. However, the return type of any such method is `readonly`. For example, if class Person has a method `public Address getAddress() readonly`, then `readonly Person` has method `public readonly Address getAddress() readonly`. By contrast, in ConstJava the return type of a method does not depend on whether it is called through a constant reference or a non-constant one. JAC does not appear to permit arrays of readonly objects, nor does the paper explain how inner classes are treated. Finally, no experience is reported with an implementation.

Skoglund and Wrigstad [SW01] take a different attitude toward immutability than other work: "In our point of [view], a read-only method should only protect its enclosing object's transitive state when invoked on a read reference but not necessarily when invoked on a write reference." A read (constant) method may behave as a write (non-constant) method when invoked via a write reference; a caseModeOf construct permits run-time checking of reference writeability, and arbitrary code may appear on the two branches. This suggests that while it can be proved that read references are never modified, it is not possible to prove whether a method may modify its argument. In addition to read and write references, the system provides `context` and `any` references that behave differently depending on whether a method is invoked on a read or write context.

The functional methods of Universes [MPH01] are pure methods that are not allowed to modify anything (as opposed to merely not being allowed to modify the receiver object).

Pechtchanski and Sarkar [PS02] provide a framework for immutability specification along three dimensions: lifetime, reachability, and context. The lifetime is always the full scope of a reference, but that might be the complete dynamic lifetime of an object or just the duration of a method call, when a parameter is annotated. The reachability is either shallow or deep. The context is whether immutability applies in just one method or in all methods. The authors provide five instantiations of the framework, and they show that immutability constraints enable optimizations that can speed up some benchmarks up by 5–10%. ConstJava permits both of the lifetimes and supplies deep reachability (Java's `final` gives shallow reachability).

Capabilities for sharing [BNR01] are intended to generalize various other proposals for immutability and uniqueness. When a new object is allocated, the initial pointer has seven access rights: read, write, identity (permitting address comparisons), exclusive read, exclusive write, exclusive identity, and ownership (giving the capability to assert rights). Each (pointer) variable has some subset of the rights. These capabilities give an approximation and simplification of many other annotation-based approaches.

Porat et al [PBKM00] provide a type inference that determines (deep) immutability of fields and classes. A field is defined to be immutable if its value never changes after initialization and the object it refers to, if any, is immutable. An object is defined to be immutable if all of its fields are immutable. A class is immutable if all its instances are. The analysis is context-insensitive in that if a type is mutable, then all the objects that contain elements of that type are mutable. Libraries are neither annotated nor analyzed: every virtual method invocation (even `equals`) is assumed to be able to modify any field. The paper discusses only class (static) variables, not member variables. The technique does not apply to method parameters or local variables. An experiment indicted that 60% of static fields in the Java 2 JDK runtime library are immutable. This is the only other implemented tool related to immutability in Java besides mine, but I was not able to obtain the tool.

Chapter 10

Future Work

This chapter briefly describes two directions for future research: dynamic `const_cast` checking and `const` inference.

10.1 const_cast checking

At present, `const_cast` is a loophole in the ConstJava type checking system. Use of `const_cast` allows a programmer to effectively override the type checking rules at will. This is undesirable, since no guarantees about the code can be enforced when the programmer can override the type checking system. On the other hand, not including `const_cast` makes the conversion of existing Java projects to ConstJava much more difficult. As the experience with ConstJava shows, existing Java projects often contain code that is safe, but that is nevertheless rejected by the ConstJava compiler's conservative analysis. Without `const_cast`, such code would need to be rewritten, which takes programmer's time and effort, and has potential for introducing new bugs into an existing project.

A way of dealing with the above problem is by dynamically enforcing the rule that constant references cannot be used to modify the state of the referred object, even in the presence of `const_cast`. A possible approach to achieving this would be to have a boolean field `check` for every reference. `check` would be set to `true` on any constant reference when `const_cast` is applied to it. Whenever an expression of the form $a_1.a_2.a_3 \ldots a_n$ is assigned to, if `check` of any of $a_1, \ldots a_n$ is true, the assignment would cause an exception at runtime. Similarly, whenever a reference expression of the form $a_1.a_2.a_3 \ldots a_n$ is copied to a non-constant reference, its `check` field would also be copied.

The above rules, modulo some details dealing with mutable and static fields, would enforce the guarantees of the ConstJava type-checking system while still allowing `const_cast` to be used by the programmer in the cases when the static analysis is over-conservative. From section 7.4.4 (p. 28) we see that, out of the total of 27 inflexibility errors, 21 were either fixed with a dynamically safe `const_cast`, or could have been fixed in this way; the other six would require rewriting. This shows that having a dynamically safe `const_cast` is useful, as it allows programmers to avoid most of code rewriting during the annotation of Java code, while at the same time preserving the guarantees of ConstJava.

Therefore, an important future research direction would be to implement an algorithm described above for `const_cast` checking, or a different one achieving the same goal, and include this algorithm into the ConstJava compiler to ensure that `const_cast` can only be used in a legitimate way.

In the present prototype implementation, the run-time checking code is inserted by another tool rather than the ConstJava compiler. Slowdowns range from 10% to 700%, but research is in progress on a number of optimizations that should greatly reduce the upper bound; access to the JVM could reduce the cost even more.

10.2 const inference

The goal of `const` inference is to have an automated tool that would annotate Java source code with `const` by analyzing the code and determining as many references as possible that are constant.

Such a tool would be especially useful if it could annotate references in parameters and return types of methods, as well as determine whether the methods themselves are constant. Annotating local variables is far less useful. A big time investment in using ConstJava comes from having to annotate Java libraries with `const` before being able to use them. If such a tool were created, this task would be automated, and hence using ConstJava would be easier.

As noted in chapter 9, there is previous work done by Porat et al [PBKM00] that does `const` inference, but only on classes and fields. The most useful `const` inference would be on method and constructor signatures, and therefore further research needs to be done in finding good ways for doing this.

Chapter 11

Conclusion

This technical report presented the development and evaluation of ConstJava, an extension to the Java language that is capable of expression and compile-time verification of immutability constraints. The specific constraint expressed in ConstJava is that the transitive state of the object to which a given reference refers cannot be modified using that reference. Compile-time verification of such constraints has numerous software engineering benefits.

During this project, ConstJava was designed, and a compiler for it was implemented. This compiler and its documentation can be downloaded at `http://pag.lcs.mit.edu/constjava/`.

In addition, numerous experiments were done to evaluate the usefulness of ConstJava in software engineering. These experiments showed that ConstJava is capable of automatically catching implementation and documentation errors as well as instances of bad coding style even in mature, well-debugged projects. ConstJava compiler found a total of 19 implementation, one documentation, and three bad-style errors in the Daikon project, a 100,000 line project that is still under development, but the majority of which is rather stable and well-tested. The compiler also discovered one implementation, two documentation and three bad-style errors in the relatively mature GizmoBall project, a 15,000 line program that had been previously well-tested and well-debugged by its authors. ConstJava compiler even found bad coding style in Sun's `java.util` implementation (see section 7.4.3), which is rather impressive considering the maturity of that code. All of this suggests that ConstJava is a very useful tool, as it can automatically catch bugs that are exceedingly difficult to track down, and remain in even stable, mature, and well-tested software project. It can be expected that ConstJava would be even more useful if used in projects from their inception, allowing many more bugs to be caught automatically rather than through the usual time-consuming debugging techniques.

The benefits above did not come without some costs. However, the costs of using ConstJava are minor compared to the benefits. While users of ConstJava need to spend extra time typing method signatures and expressions correctly with `const`, and while some safe code instances are rejected by ConstJava compiler, the benefits of catching errors at compile-time that would otherwise be exceedingly difficult to detect strongly suggest that using ConstJava is worthwhile.

Appendix A

The ConstJava Language Definition

A.1 Introduction

The purpose of this document is to define the language extensions for the Java language that enable the programmer to specify immutability constraints about references.

It is assumed that the reader already knows the Java language. The specification in this document is for the language extensions only, and not for the syntax and semantics already present in the Java language. The Java language with the extensions described in this document is hereinafter referred to as ConstJava.

A.2 Keywords

`mutable`, `template` and `const_cast` are new keywords introduced into the language, and can no longer be used as identifiers. `const` was a keyword in the Java language, but was not used in any of its syntax; it is a keyword that is used in ConstJava.

A.3 Types

A.3.1 ConstJava's types

ConstJava's type hierarchy extends that of Java by including, for every Java reference type `T`, a new type `const T`. References of type `const T` are just like those of type `T`, but cannot be used to modify the object to which they refer.

Formally, the types in ConstJava are the following:

1. The null type `null`.
2. The Java primitive types.
3. Instance references. If `O` is any class or interface, then `O` is a type representing a references to an instance `O`.
4. Arrays. For any non-null type `T`, `T[]` is a type, representing an array of elements of type `T`.
5. Constant types. For any non-null non-constant type `T`, `const T` is a type.

For convenience in the usage later, we define the concepts of an n-dimensional array of `T`, and the depth and base of a given type. It is intuitively clear what an n-dimensional array of `T` is. The depth is just the nesting depth of an array type, while the base of an array type is the type with all array dimensions removed. Formally, for a type `T`, we define:

Given an integer n, an n-dimensional array of `T` is:

- if $n = 0$, the n-dimensional array of `T` is `T`.
- if `S` is the $(n-1)$-dimensional array of T, then `S[]` is the n-dimensional array of `T`.

Depth:

- if `T` is null, primitive, or instance reference, *depth*(`T`) $= 0$.
- if *depth*(`T`) $= n$, then *depth*(`T[]`) $= n + 1$.
- if *depth*(`T`) $= n$, then *depth*(`const T`) $= n$.

Base:

- if `T` is null, primitive or instance reference, *base*(`T`) = `T`.
- if *base*(`T`) = `S`, then *base*(`T[]`) = `S`.
- if *base*(`T`) = `S`, for a constant type S, then *base*(`const T`) = `S`.
- if *base*(`T`) = `S`, for a non-constant type S, then *base*(`const T`) = `const S`.

A.3.2 The syntax for types

Except for the null type, every type can be named in ConstJava, although not in a unique way. The following syntax is used for naming the type (for clarity, non-terminals are enclosed in quotes).

```
Type ::= [ 'const' ] BasicType ( '[' ']' )*

BasicType ::=
   PrimitiveType
   Name
   '(' Type ')'
```

The type named by the a sequence of tokens, S, that parses according to the above grammar is determined as follows. Let S' be the subsequence of S matched by the `BasicType` non-terminal. First the type named by S' is determined. If `BasicType` matches according to the `PrimitiveType` production, S' names the primitive type matched there; if `BasicType` matched according to the `Name` production, the name matched by that production must name an accessible class or interface, and S' names a reference to an instance of that class or interface. If `BasicType` matches by the `'(' Type ')'` production, then S' names the same type as the subsequence matched by `Type` in the production.

Once it is determined what S' names, the type named by the S is determined as follows. Suppose `T` is the type named by S'. If n is the number of `'[' ']'` pairs following `BasicType`, let `U` be the n-dimensional array of `T`. Let `V` be `const U` or `U`, depending on whether the initial `const` in the grammar above is present in the parsing. The type named by S is `V`.

A.3.3 Type equality and subtyping

The equality relation is defined on the types as follows:

1. For primitive types, the null type and references to instances of classes and interfaces, two types are equal iff they are the same Java type.
2. `const T` and `const S` are equal iff *depth*(`T`) = *depth*(`S`) and *base*(`const T`) = *base*(`const S`).

3. T[] and S[] are equal iff T, S are.
4. For a non-constant type T, T and const S are equal iff T and S are equal, and T is either primitive or is a reference to an instance of an immutable class or interface (see section A.4.4 for description of immutable classes/interfaces).

Note that item 2 implies, for example, that const int[][] and const (const int[])[] are equivalent. In other words, a constant array of array of int is the same as a constant array of constant int arrays. Item 4 captures the notion that T and const T are equivalent for immutable types T.

Equal types are considered to be the same type. They are interchangeable in any ConstJava program.

A subtyping relationship (T subtype of S, written as T < S) is also defined on types. It is the transitive reflexive closure of the following:

1. byte < char, byte < short, char < int, short < int, int < long, long < float, float < double.
2. null < T for any type T which is not a primitive type.
3. If T, S are classes such that T extends S or interfaces such that T extends S, or S is an interface and T is a class implementing S, then T < S.
4. For any non-null types T and S, if T < S, then T[] < S[].
5. For any non-constant non-null type T, T < const T.
6. For any non-constant non-null types T and S, if T < S, then const T < const S.
7. For any non-null type T, then T[] < java.io.Serializable, T[] < Cloneable, and T[] < Object.
8. For any type non-constant non-null type T, (const T)[] < const T[].

A.4 Other new syntax

In addition to the new type hierarchy, ConstJava has the following features: constant methods and constructors, mutable fields and immutable classes and interfaces. These are described in the subsections below.

A.4.1 Constant methods

A method declaration has the following grammar:

```
MethodDeclaration ::=
  MethodModifiers (Type | 'void') Identifier Arguments ['const'] [ThrowsClause]
    (MethodBody | ';')
```

The only difference from the Java grammar for method declaration is the optional keyword const immediately after the Arguments of the method. If the declaration contains this keyword const, it is said to declare a constant method. Only instance (non-static) methods can be declared as constant. There are no other restrictions on which methods can be declared as constant.

The semantics for constant methods are as follows. A constant method is an instance method that can be invoked through a constant reference. Non-constant methods cannot be so invoked. Additionally, this is of a constant type inside the body of a constant method, so that a constant method cannot modify the state of the object on which it is invoked. For a formal description of these rules, see the type checking rules described in section A.5.

A.4.2 Constant constructors

A constructor declaration has the following grammar:

```
ConstructorDeclaration ::=
  ConstructorModifiers Identifier Arguments ['const'] [ThrowsClause]
  ConstructorBody
```

The only difference from the Java grammar for constructor declaration is the optional keyword `const` immediately after the `Arguments` of the constructor. If the declaration contains this keyword `const`, it is said to declare a constant constructor. If a constructor for a class that is not an inner class is declared to be constant, a compile-time error occurs. There are no other restrictions on which constructors can be declared as constant.

The semantics of constant constructors are as follows. A constant constructor is a constructor for an inner class that can be invoked with a constant reference for the enclosing object. Non-constant constructors for inner classes cannot be so invoked. Inside the body of a constant constructor, no modifications to the enclosing instance are allowed. For a formal description of these rules, see the type checking rules described in section A.5.

A.4.3 Mutable fields

A field declaration has the following grammar:

```
FieldDeclaration ::=
  FieldModifiers Type Identifier [Initializer]
     (',' Identifier [Initializer])* ';'

FieldModifier ::=
  ('mutable' | 'private' | 'public' | 'protected' | 'final' |
   'static' | 'transient')*
```

The only difference from the grammar for field declaration in Java is the possibility of `mutable` appearing as a modifier. If a declaration contains `mutable` as a modifier, the fields declared in it are said to be mutable. Only instance (non-static) fields can be declared as mutable. There are no other restriction on which fields can be declared as mutable.

The semantics of mutable fields are as follows. A mutable field is not part of the state of the object to which it belongs. Thus, the state of a mutable field of a given object can be changed through constant reference to that object, or by constant methods invoked on that object, or by constant constructors invoked with that object as the enclosing instance. For a formal description of these rules, see the type checking rules described in section A.5.

A.4.4 Immutable classes and interfaces

Class and interface declarations have the following grammar:

```
ClassDeclaration ::=
  TypeModifiers class Identifier [ExtendsClause] [ImplementsClause]
    ClassBody

InterfaceDeclaration ::=
  TypeModifiers interface Identifier
    [InterfaceExtendsClause] ClassBody
```

```
TypeModifiers ::=
  ('const' | 'private' | 'protected' | 'public' | 'abstract' |
   'final' | 'strictfp' | 'static' )*
```

The only difference from the grammar for class/interface declaration in Java is the possibility of `const` appearing as a modifier. If a declaration contains `const` as a modifier, the class/interface declared in it is said to be immutable.

The semantics of immutable classes and interfaces are as follows. An instance of such a class or interface, once instantiated, cannot be modified. Therefore, in an immutable class/interface, every instance method and every constructor is implicitly declared as constant, and any instance field which is not explicitly declared as mutable is implicitly declared as final and if its type `T` is not a constant type, it is implicitly changed to be `const T`.

It is a compile-time error for a non-immutable class or interface to extend or implement an immutable one. It is a compile-time error for an immutable class or interface to inherit an instance field which is neither mutable nor final with a constant type, or to inherit, override or implement an instance method which which is not constant.

A.5 Type checking rules

ConstJava has the same runtime behaviour as Java. However, at compile time, checks are done to ensure that modification of objects through constant references, or similar violations of the language, do not occur. These rules are described in this section. Section A.5.1 introduces some definitions. Section A.5.2 then presents the type checking rules.

A.5.1 Definitions

Definitions for types

- Primitive type: any Java primitive type, e.g., `boolean` or `double`
- Reference type: any non-primitive type.
- Numeric type: any primitive type other than `boolean`.
- Integral type: any numeric type other that `float` and `double`.
- Null type: `null`.
- Array type: Any type `S` such that `S = T[]` for some type `T`.
- Constant type: Any type `S` such that `S = const T` for some type `T`.

Definitions relating to method invocations

These definitions are the same as those in Java, except for the presence of the third clause in the definition of specificity.

Compatibility: Given a method or constructor M and a list of arguments $A_1, A_2, \dots A_n$, we say that the arguments are compatible with M if M is declared to take n paremeters, and for each i from 1 to n, the type of A_i is a subtype of the declared type of the ith parameter of M.

Specificity: Given two methods of the same name or two constructors of the same class, M_1, M_2, we say that M_1 is more specific than M_2 if the following three conditions hold:

1. M_1 and M_2 take the same number of parameters, say with types $P_1, P_2 \dots P_n$ for M_1, and $Q_1, Q_2 \dots Q_n$ for M_2, and for each i from 1 to n, P_i is a subtype of Q_i.

2. The class/interface in which M_1 is declared is a subclass/subinterface of the one where M_2 is declared, or M_1 and M_2 are declared in the same class/interface.
3. Either M_1 is not constant or M_2 is constant (or both).

A.5.2 Type checking rules

Programs

A program type checks if every top-level class/interface declaration in the program type checks.

Class/Interface declarations

A class or interface declaration type checks if all of the following hold:

1. (a) The class/interface is immutable and each of the methods declared in any of its superclasses or superinterfaces is either private, static or constant, and each of the fields declared in any of its superclasses is either private, static, mutable or both final and of a constant type, or
   (b) the class or interface is not immutable, and neither is its direct superclass or any of its direct superinterfaces.
2. No two fields of the same name are declared within the body of the class/interface.
3. No two methods of the same name and signature or two constructors of the same signature are declared within the body of the class/interface. Signature includes the number and the declared types of parameters, as well as whether the method is constant.
4. Every declared field, method, member type, instance initializer and static initilizer of the class/interface type checks.

Variable declarations

For a field or local variable declaration of type `T`:

- If it does not have an initializer, it type checks.
- If it has an initializer of the form `=` E for an expression E, it type checks iff the assignment of the expression E to a left hand side with type `T` would type check.
- If it has an initializer of the form " `= {` $I_1, \ldots I_k$ `}` ", where I_k are initializers, it type checks iff `T = S[]` or `T = const S[]` for some type `S`, and the declaration `S v =` I_k or `const S v =` I_k respectively would type check for every k between 1 and n.

Method declarations

A method, constructor, instance initializer, or static initializer type checks if every expression, local variable declaration, and local type declaration in the body of the method, constructor, instance initializer, or static initializer type checks.

Expressions

Each expression has a type and a boolean property called assignability associated with it. An expression is type checked recursively, with all subexpressions type checked first. If the subexpressions type check, then their types and assignabilities are used to deduce whether it type checks, and if so, to deduce its type and assignability. Otherwise, the given expression does not type check. The rules for type-checking an expression given types and assignabilities of its subexpressions are given below.

Assignments

- A`=`B: type checks if the expression A is assignable, and one of the following holds:
  - the type of A is supertype of that of B, or
  - A is of type `byte`, `short`, or `char`, and B is of type `byte`, `short`, `char`, or `int` and is a compile-time constant whose value is within the value range of the type of A.
- A`+=`B: type checks if the expression A is assignable, and one of the following holds:
  - A and B are both of numeric types, or
  - A is of type `String`.
- A`-=`B, A`*=`B, A`/=`B, A`%=`B: type checks whenever A is assignable and A and B are of numeric types.
- A`<<=`B, A`>>=`B, A`>>>=`B: type checks whenever A is assignable and A and B are of integral types.
- A`||=`B, A`&&=`B: type checks whenever A is assignable and both A and B are of type `boolean`.
- A`&=`B, A`|=`B, A`^=`B: type checks whenever A is assignable and either both A and B have integral types or they are both of type `boolean`.

The type of any assignment expression that type checks is the same as the type of the left hand side, and the expression is not assignable.

Other compound expressions

- A`?`B`:`C: In order for this expression to type check, A must be of type `boolean`. Also, if `T1` and `T2` denote the types of B and C, then one of the following must hold:
  1. `T1` $<$ `T2`, `T1` $<$ `const T2`, `T2` $<$ `T1`, or `T2` $<$ `const T1`. In this case expression is of the least supertype of `T1` and `T2`.
  2. `T1` and `T2` are, in some order, `char` and `short`; in this case expression is of type `int`.
  3. One of B and C is of type `T`, where `T` is `byte`, `short`, or `char`, and the other is a constant expression of type `int` whose value is representable in type `T`. In this case expression is of type `T`.
- A`||`B, A`&&`B: the expression type checks if A and B are of type `boolean` and is of type `boolean`.
- A`|`B, A`^`B, A`&`B: the expression type checks if A and B are of type `boolean`, in which case it is of type `boolean`; or if A and B are of integral type, in which case its type is their least supertype.
- A`==`B, A`!=`B: always type checks[1], and is of type `boolean`.
- A `instanceof T`: always type checks, is of type `boolean`.
- A`<`B, A`>`B, A`<=`B, A`>=`B: type checks if A and B are of numeric type; it is of type `boolean`.
- A`<<`B, A`>>`B, A`>>>`B: type checks if A and B are of integral type; the expression is of the least supertype of the type of A and of `int`.
- A`+`B: type checks if A and B are of numeric type, in which case the expression is of the least supertype of `int` and the types of A and B; or if one of A and B is of type `String`, in which case the expression is of type `String`.
- A`-`B, A`*`B, A`/`B, A`%`B: type checks if A and B are of numeric type, in which case the expression is of the least supertype of `int` and the types of A and B.

[1]Of course, in this and other cases where the ConstJava rules are not stronger than the Java rules, any ConstJava expression still has to type check according to the Java rules.

- `+`*A*, `-`*A*: type checks if *A* is of numeric type; the expression is the least supertype of `int` and the type of *A*.
- `++`*A*, `--`*A*, *A*`++`, *A*`--`: type checks if *A* is assignable and of numeric type; the expression is of same type.
- `~`*A*: type checks if *A* is of numeric type; the expression is of the least supertype of `int` and the type of *A*.
- `!`*A*: type checks if *A* is of type `boolean`; the expression's type is `boolean`.
- `(T)`*A*: fails to type check iff *A* is of of type `const S` for a non-immutable class or interface `S`, and `T` is a non-constant type, or if it is of type `S[]` and `T` = `T'[]` for some types `S` and `T'`, and a cast from `S` to `T'` would have been illegal. The type of a cast exception is `T`.
- `const_cast<`*A*`>`: always type checks. If the type of *A* is non-constant, this expression is of the same type. If the type of *A* is `const S[]` for some `S`, then the type of this expression is `(const S)[]`. If the type of *A* is `const S` where *depth*(`S`) = 0, the type of this expression is `S`.

Every expression in this section is not assignable.

Primary Expressions Everywhere within these rules for type-checking primary expressions, when the location of an expression is considered, instance field initializers and instance initializers are considered to be contained in every constructor of the corresponding class. Default constructors for named inner classes are never constant. Default constructors for anonymous classes are constant iff the enclosing instance of the anonymous class instantiation is given through a constant reference.

- A literal is of type `boolean`, of a numeric type, of type `String`, or of type `null`, depending on the value of the literal. Literals are not assignable.
- `this` does not type check in a static context; in a non-static context:
  - in a non-constant method or a non-constant constructor of a class `C`, `this` has type `C`
  - in a constant method or a constant constructor of a class `C`, `this` has type `const C`

  `this` is not assignable.
- `NAME.this` does not type checks in a static context. In a non-static context, suppose that `NAME` names a class `C`, and let `I` be the innermost class in which `NAME.this` occurs. If `I` is not an inner class of `C`, the expression does not type check. Otherwise, it does, and:
  - in a non-constant method or a non-constant constructor of `I`, `NAME.this` has type `C`
  - in a constant method or a constant constructor of `I`, `NAME.this` has type `const C`

  `NAME.this` is not assignable.
- `(`*A*`)` always type checks and is of the same type as *A*, and is not assignable.
- `T.class` always type checks and is of type `Class` and is not assignable.
- *A*`.new NAME(ARGS)` where *A* is an expression: let the type of *A* be `T`, and let `NAME` name a class `C`. The expression type checks if one of the following holds:
  1. `T` is a subtype of `O` for some class `O`, `C` is an accessible direct inner class of `O`, and there exists a most specific accessible constructor of `C` that can be called on `ARGS`. In this case the expression is of type `C`.
  2. `T` is a subtype of `const O` but not of `O`, for some class `O`, and `C` is an accessible direct inner class of `O`, and there exists a most specific accessible constant constructor `C` that can be called on `ARGS`. In this case the type of the expression is `const C`.

In either case, the expression is not assignable.

- `new NAME(ARGS)`: suppose `NAME` names a class `C`; then this expression type checks if either:
  - `C` is a non-inner class, and there exists a most specific accessible constructor of `C` that is compatible with `ARGS`, or
  - `C` is a direct inner class of a class `O`, and `O.this.new NAME(ARGS)` type checks.

  The expression has type `C` and is not assignable.
- Array instance creation expression: Let `T` be the array type whose instance is being created, and let `S` be such that `T` = `S[]`. The expression type checks whenever all index expressions involved are of integral types and the array initializer $\{E_1, \ldots, E_n\}$, if any, consists of initializers E_i such that $E = E_i$ would type check for an expression E of type `S`. The type of the array instance creation is `T`. An array instance creation expression is not assignable.
- A`[`E`]`: type checks if E is of integral type, A is of type `T[]` or `const T[]` for some type `T`; the type of the expression is respectively `T` or `const T`. The expression is assignable in the first case, and not assignable in the second.
- A`.IDENTIFIER`, where A is an expression: let `T` be a non-constant reference type such that A's type is `T` or `const T` (if no such type exists, the expression does not type check). Then the expression type checks if one of the following holds:
  1. `T` is a non-array type, and `IDENTIFIER` is the name of an accessible field of the class or interface named by `T`. Let `S` be the declared type of the field. Then the expression is of type `const S` if A is of constant type and the field is not static nor mutable, and is not of a constant type. Otherwise it is `S`. The expression is assignable iff the field is static or mutable, or A is of a non-constant type.
  2. `T` is an array type, and `IDENTIFIER` is `length`, in which case the expression is of type `int` and is not assignable.
  3. `T` is an array type, and `IDENTIFIER` is a field of the class `Object`, or of one of the interfaces `Cloneable` or `java.io.Serializable`. Let `S` be the declared type of the field. Then the type of the expression is `const S` if A is of constant type and the field is not static nor mutable, and is not of a constant type. Otherwise it is `S`. The expression is assignable iff the field is static or mutable, or A is of non-constant type.
- `NAME.IDENTIFIER`: If `NAME` resolves to a field, variable or parameter of type `T`, this expression type checks iff the expression E`.IDENTIFIER` with E of type `T` typechecks; the type and assignability of the two expressions are the same. Otherwise this expression type checks whenever `NAME` is the name of an accessible class or interface `C`, and `IDENTIFIER` is the name of an accessible static field of `C`; the declared type of the field is the type of the expression and the expression is assignable.
- `IDENTIFIER`: type checks if one of the following holds:
  1. There is a visible local variable or parameter declaration with a name `IDENTIFIER`; the type of the expression then is the declared type of that local variable or parameter, and the expression is assignable.
  2. 1 does not hold, the expression occurs in a static context and `IDENTIFIER` is the name of an accessible static field of a class `C` within whose declaration the expression occurs; the type and assignability of the expression is the same as that of `C.IDENTIFIER` for the innermost such class `C`.
  3. 1 does not hold, the expression occurs in a non-static context and there exist a class `C` within whose declaration the expression occurs and which contains an accessible field with name `IDENTIFIER`, and for the innermost such class `C`, `C.this.IDENTIFIER` or `C.IDENTIFIER` type checks; the type and assignability of `IDENTIFIER` is the type of `C.this.IDENTIFIER` or `C.IDENTIFIER` respectively. An exception is that if `IDENTIFIER` is within the body of a constant constructor of the class `C`, it is assignable.

- `super.IDENTIFIER`: does not type check in static context. In non-static context, let `C` be the innermost enclosing class or interface. If `C` is an interface or the class `Object`, this expression does not type check. Otherwise let `P` be the direct superclass of `C`. If `IDENTIFIER` names an accessible field of `P`, this expression type checks. Let the declared type of that field be `T`. If the field is mutable or static, or if `T` is a constant type, the type of `IDENTIFIER` is `T`. Otherwise, the type of `IDENTIFIER` is `T` or `const T`, depending on whether the method or constructor in which it occurs is, respectively, non-constant or constant.

- A`.IDENTIFIER(ARGS)`, where A is an expression: let `T` be a non-constant reference type such that A's type is `T` or `const T` (if no such type exists, the expression does not type check). The expression type checks if one of the following holds:

  1. `T` is a non-array reference and there exists a most specific accessible method of name `IDENTIFIER` of the class or interface named by `T` which is compatible with the arguments `ARGS` and is constant or static if type of A is `const T`.
  2. `T` is an array reference. Let `S` be the type of A, let $n = \textit{depth}(\texttt{S})$, and let `S'` be the n-dimensional array of *base*(`S`). Then the expression type checks if there is a most specific accessible method of name `IDENTIFIER` in the class `Object`, or in one of the interfaces `Cloneable` or `java.io.Serializable` that is compatible with the arguments `ARGS` and is constant or static if `T` $\neq$ `S'`.

  The type of the expression in either case is the declared return type of that method.

- `NAME.IDENTIFIER(ARGS)`: If `NAME` resolves to a field, variable or parameter of type `T`, this expression type checks iff the expression E`.IDENTIFIER(ARGS)` with E of type T type checks; the two expressions have the same type and assignability. Otherwise the expression type checks whenever `NAME` is the name of an accessible class `C`, and there exists the most specific static method of name `IDENTIFIER` in `C` compatible with the arguments `ARGS`. The type of the expression is the declared return type of the method. This expression is never assignable.

- `IDENTIFIER(ARGS)`: type checks if one of the following holds:

  1. It occurs in a static context, and for some class or interface `C` inside whose declaration this expression occurs, the expression `C.IDENTIFIER(ARGS)` type checks; the type of the expression is the type of `C.IDENTIFIER(ARGS)` for the innermost such class or interface `C`. The expression is not assignable.
  2. It occurs in non-static context, and for some class or interface `C` inside whose declaration this expression occurs, `C.this.IDENTIFIER(ARGS)` or `C.IDENTIFIER(ARGS)` type checks; the type of the expression equals the type of `C.this.IDENTIFIER(ARGS)` or `C.IDENTIFIER(ARGS)` respectively for the innermost such class `C`. The expression is not assignable.

- `super.IDENTIFIER(ARGS)`: does not type check in static context. In non-static context, let `C` be the innermost enclosing class or interface. If `C` is an interface or the class `Object`, this does not type check. Otherwise let `P` be the direct superclass of `C`. If `IDENTIFIER` occurs inside a constant method or constant constructor, this type checks if there is a most specific accessible constant or static method of name `IDENTIFIER` compatible with `ARGS`. Otherwise, the expression type checks if there is a most specific accessible method of name `IDENTIFIER` compatible with `ARGS`. The method's return type is the type of the expression. The expression is never assignable.

A.6 Templates

In addition to the features of ConstJava described above, a template feature is included in ConstJava. This feature allows creation of polymorphic methods or types, with polymorphism over constness only being available.

A.6.1 Polymorphic methods and constructors

The syntax for polymorphic method/constructors is as follows

```
PolymorphicMethod ::=
  'template' '<' VariableList '>' MethodDeclaration

PolymorphicConstructor ::=
  'template' '<' VariableList '>' ConstructorDeclaration

VariableList ::=
  Identifier
  Identifier ',' VariableList
```

It is a compile-time error to repeat a variable in the variable list.

In order to expand a polymorphic method/constructor, first all templates of method, constructors, or types nested within it are expanded; the template declaration is replaced by a distinct method/constructor declaration for each possible assignment of booleans to variables in the `VariableList`. Inside the template, anywhere that `const` can appear in the usual grammar for the language, `const ? Identifier` may appear, where `Identifier` is the name of one of the variables bound in the template. When the template is expanded, in the newly created declarations corresponding to the binding of `Identifier` to true, `const ? Identifier` is replaced by `const`; in those where the variable is bound to false, `const ? Identifier` is simply removed.

For example

```
template<a> const?a Object identity(const?a Object o) {
  return o;
}
```

gets expanded as

```
Object identity(Object o) {
 return o;
}

const Object identity(const Object o) {
 return o;
}
```

A.6.2 Polymorphic types

To declare a polymorphic type, the following syntax must be used:

```
PolymorphicType ::=
  'template' '<' VariableList '>' TypeDeclaration

TypeDeclaration ::=
  ClassDeclaration
  InterfaceDeclaration
```

The template expansion happens the same way as it does for polymorphic methods and constructors. Namely, first all the templates nested within this one are expanded, then the template declaration is replaced with a separate type declaration for each possible boolean assignment to the variables in `VariableList`. `const ? Identifier` constructs are replaced within the body of the template in the same way as for method and constructor templates.

The only difference between polymorphic types and polymorphic methods is that types created from a template get distinct names. The name of a type created from the template is obtained by taking the original name of the type specified in the template, then appending `< VariableList >` (where `VariableList` is taken from the template declaration), and finally replacing each varible with `const` or nothing, depending on whether the variable is assigned true or false during the creation of this type declaration.

For example

```
template<a,b> class A ...
```

will produce four new classes, `A<,>`, `A<const,>`, `A<,const>` and `A<const,const>`.

In general, the syntax for name now changes to

```
Name ::=
  SimpleName
  Name '.' SimpleName

SimpleName ::=
  Identifier
  Identifier '<' ( 'const' | ) (',' ( 'const' | )* ) '>'
```

Of course, the second production for SimpleName can be used only to name classes or interfaces created during template expansion.

Note that the previous rule that `const ? Identifier` can appear anywhere where `const` can legally appear applies to the syntax for `SimpleName`. For example, `A<const?a,const?b>` is a legal name, which will refer to a different class depending on values of `a` and `b`.

Bibliography

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, pages 2–27, Budapest, Hungary, June 18–22, 2001.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 183–200, Vancouver, BC, Canada, October 20–22, 1998.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, Boston, MA, second edition, 2000.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Software Series. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[KT01] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. Available from `http://softech.informatik.uni-kl.de/en/publications/universe.html`.

[PBKM00] Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in java. In *CASCON*, Mississauga, Ontario, November 13–16, 2000.

[PS02] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *Joint ACM-ISCOPE Java Grande Conference*, pages 202–211, Seattle, WA, November3–5, 2002.

[Str00] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Boston, MA, special edition, 2000.

[SW01] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *3rd Workshop on Formal Techniques for Java Programs*, Budapest, Hungary, June 18, 2001. Revised.

[VS] Sreeni Viswanadha and Sriram Sankar. `http://www.experimentalstuff.com/Technologies/JavaCC/`.