Compiler-enforced immutability in Java

Adrian Birka and Michael D. Ernst

Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square
Cambridge, MA 02139, USA
adbirka@mit.edu, mernst@lcs.mit.edu

Abstract

This paper describes 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.

The paper describes the design and implementation of ConstJava, including the type-checking rules for the language. This paper also discusses experience with writing ConstJava code in and with converting Java code to ConstJava.

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`, whose usage is modeled after C++. The language is backwards compatible with Java. In addition, ConstJava code is interoperable with Java code, and runs on an unmodified Java Virtual Machine. The compiler for ConstJava can be obtained by e-mailing the authors of this paper.

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 behaviour of methods, and eases reasoning about and optimization of programs.

We 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 us 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 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 1: A partial implementation of a set of integers.

This paper is organized as follows. Section 2 further motivates immutability constraints. Section 3 describes the ConstJava language, section 4 discusses the design of ConstJava, and section 5 gives its type-checking rules. Then section 6 describes our experience with using ConstJava. Finally, section 7 considers related work, and section 8 concludes.

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 paper 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 section gives three examples of such benefits, showing enforcement of interface contracts, prevention of representation exposure, and granting clients read-only access to internal data. We use a class representing a set of integers (Figure 1) to illustrate the problems and benefits.

2.1 Enforcement of contracts

Method specifications describe both what a method does and what it does 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 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

Accessor methods 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()` will be unable to modify the returned array, thus permitting the simple and efficient implementation of the method to remain in place without exposing the representation.

3 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 7 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 3.1.
  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 3.2.
  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. See section 3.3.

- `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 3.4.

- `template` can be used in a declaration of a method, constructor, class, or interface, to parametrize the declaration based on constness of some type. This can shorten code and remove error-prone duplication. See section 3.5.

- `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 the object. The `const_cast` operator does not introduce a loophole into the type system, for constness of the reference can be enforced at run time. See section 3.6.

ConstJava is backward compatible with Java: any Java program that uses none of ConstJava's keywords is a valid ConstJava program. Also, ConstJava is interoperable with Java. As described in section 4.1, any Java code can be called from ConstJava code.

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 `const` without losing the ability to compile that program with Java.

3.1 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

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

(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 section 5 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 3.2) 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 typecast 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
```

3.2 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.

3.3 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 3.1). Subclasses or subinterfaces of immutable classes and interfaces must be declared immutable.

3.4 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 type checker, 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.

3.5 Templatization over constness

ConstJava allows method definitions to be templatized 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:

```
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 Object`, 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 defines classes `Vector<>`, `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, templatized as follows:

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

```
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`.

3.6 Casting away of const

The keyword `const_cast` permits casting away `const` from a type. 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 type checker cannot prove this fact, and hence the method cannot be declared as constant. For an example of this, see section 6.3.2.

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>
```

`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 type checker ordinarily guarantees that no mutation can occur through a `const` reference. The presence of `const_cast` enables code to violate that restriction. In order to regain soundness, ConstJava can insert 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. (In our 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 we are working on a number of optimizations that should greatly reduce the upper bound; access to the JVM could reduce the cost even more.)

4 Language design

This section describes three immutability checking issues that have not been handled by previous research, along with how we address them.

4.1 Interoperability with Java

A major goal during the design of ConstJava was ensuring that ConstJava is interoperable with Java. The language treats any Java method as a ConstJava method with no `const`ness in the parameters, return type, or on the method, and similarly with constructors and fields. In other words, since ConstJava type checker 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 `const`ness 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 type checker permits the user to specify alternative signatures for methods and constructors, and alternative types for fields in Java libraries. ConstJava comes with the following libraries annotated: `java.lang`, `java.io`, `java.util`, `java.util.zip`, `java.awt` and subpackages, `java.applet`, and `javax.swing` and subpackages.

The type checker 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 type checker that the `toString` method is constant. The type checker trusts these annotations without checking them.

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 interoperability feature, meant to ease the process of converting Java code to ConstJava code, is the "`/*=`" comment syntax described in section 3. 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.

4.2 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. We must ensure that no code inside an inner class can violate an immutability contraint. 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 (Section 5) 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 3.2) 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 type checker 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 typecheck. 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.

4.3 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.

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 catch(const Throwable t) should catch everything in ConstJava, 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 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();
```

In order 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 possibility 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 we do not believe that a less restrictive treatment of constant exceptions would be an important feature in the ConstJava language, we adopted a more straightforward approach of disallowing throws of constant exceptions.

5 Type-checking rules

ConstJava has the same runtime behaviour as Java[3]. 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 5.1 introduces some definitions. Section 5.2 then presents the type-checking rules.

5.1 Definitions

5.1.1 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 $base(\texttt{T[]}) = \texttt{S}$.
- if $base(\texttt{T}) = \texttt{S}$, for a constant type S, then $base(\texttt{const T}) = \texttt{S}$.
- if $base(\texttt{T}) = \texttt{S}$, for a non-constant type S, then $base(\texttt{const T}) = \texttt{const S}$.

5.1.2 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 $depth(\texttt{T}) = depth(\texttt{S})$ and $base(\texttt{const T}) = base(\texttt{const S})$.
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.

[3] Except for possible checks of `const_cast` described in section 3.6.

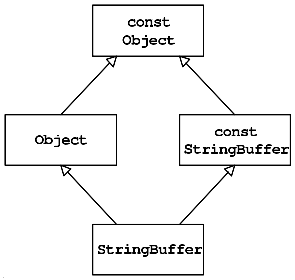

Figure 2: ConstJava type hierarchy, including constant and non-constant version of each Java reference type. Arrows connect subtypes to supertypes.

Item 2 implies 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.

Equal types are considered to be the same type. They should be 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` 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 2

5.1.3 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).

5.2 Type-checking rules

5.2.1 Programs

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

5.2.2 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 are declared within the body of the class/interface. Signature includes number and declared types of parameters, as well as whether the method is constant.

4. Every field, method, member type, instance initializer and static initializer declared within the class or interface type checks.

5.2.3 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 " `= { ... }` ", it type checks iff `T` is an array type and the declaration would type check had the initializer been " `= new T { ... }`".

5.2.4 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.

5.2.5 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 assignability are used to type check the given expression and 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 we give only the rules that are substantially different from those in Java; for a full set of type checking rules, refer to the technical report.

- 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.

- A==B, A!=B: Let T1 and T2 be the types of A and B respectively. The expression type checks if T1 $<$ T2, T1 $<$ const T2, T2 $<$ T1, or T2 $<$ const T1. The expression is of type boolean, and 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.
- 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, a non-constant constructor, or an initializer 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, constructor or initializer 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.

6 Experiments

In order to evaluate ConstJava, we wrote code in ConstJava and annotated 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 3 displays statistics about our experiments.

6.1 Container classes written from scratch

As explained in section 3.5, the container classes in `java.util` cannot be used with ConstJava, and a templatized version of the container classes must be written. Because of this, and also to gain experience with ConstJava, we wrote many of the container classes in the `java.util` package from scratch in ConstJava, namely the classes `Collection`, `AbstractCollection`, `Set`, `AbstractSet`, `List`, `AbstractList`, `AbstractSequentialList`, `Iterator`, `ListIterator`, `ArrayList`, `Vector`, `LinkedList`, `HashSet`, `Map`, `AbstractMap`, and `HashMap`.

6.2 Annotation of Java code

Our methodology for annotating Java programs with `const` proceeded in three stages. During the first stage of the annotation ("Signature" in Figure 3), we 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 parameter would not be marked with `const`.

The second stage of the annotation ("Implementation" in Figure 3) was to annotate the private signatures and the implementations of all methods. Finally, the third stage ("Type check, fix errors") involved running the type checker on the resulting program, and considering and correcting any type checking failures.

6.2.1 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 a group of four people that included the first author of this paper, who wrote about a third of the code. The program uses Java to implement an extensible pinball game.

During the third stage of the annotation experiment, the invocation of the type checker on the annotated project found 55 different type-checking errors that are tabulated in figure 3. Section 6.3 describes the categories and gives examples of errors in each category.

6.2.2 Daikon annotation

Daikon is a tool for dynamic detection of invariants in programs [Ern00, ECGN01]. Much of the Daikon code was written by the second author of this paper, but the annotation was performed without any help by the first author, who had no previous experience with the system (not even as a user).

We have annotated approximately 28,000 lines during this experiment[4]. Again, the time spent on this experiment and the results of this annotation appear in figure 3. Note that because this experiment is still in progress, six type checker errors have not yet been unresolved.

[4] The Daikon project is approximately 80,000 lines in total, but as of the writing of this paper, only 28,000 lines were annotated.

	New code	Annotated Java code			
Program	`java.util`	Gizmoball	Daikon	ConstJava	`java.util`
Code size					
classes	52	172	438	466	38
methods	359	633	1114	1007	163
lines	2687	15476	28381	15633	4795
NCNB lines	1828	9222	20072	9394	2134
Annotations					
`const`	837	657	1068	2794	611
`mutable`	27	45	27	64	6
`template`	94	13	143	79	65
`const_cast`	2	6	0	7	14
Code errors	N/A			N/A	N/A
Documentation		2	3		
Implementation		1	3		
Bad Style		3	1		
ConstJava problems					
Inflexibility		3	1		
Incompleteness		1	0		
Annotation errors					
Signature		11	24		
Implementation		31	85		
Library		3	13		
Time (hh:mm)	N/A			N/A	N/A
Signature		2:40	5:30		
Implementation		4:30	6:55		
Type check, fix errors		6:10	15:40		

Figure 3: 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 6.3 explains the error categories. The beginning of section 6.2 explains the time categories. Errors and time were recorded for only two of the programs.

6.2.3 The annotation of the type checker

Chronologically, the first major annotation experience was annotating the ConstJava type checker 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 type checker, this experience could not provide much information about the ease of use of ConstJava or about its utility.

6.2.4 Container classes annotated from Sun source code

In addition to writing some templatized container classes from scratch (Section 6.1), we 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 we have not kept 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 6.3).

6.3 Type checker error classifications

This section describes of what kind of type-checker errors were put into each of the categories used in figure 3, together with some examples of such errors.

6.3.1 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 subcategories a given error belong is inherently subjective. The three subcategories 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 our 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.

Another example of bad coding style that is prevented by the type checker is 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`.

6.3.2 ConstJava Problems

These errors are caused by weaknesses either in the language or in the type checker.

Inflexibility of the Language: This category represents safe code rejected by the type checker'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 type checker's static analysis, and so the type checker rejects this safe code. We corrected this error by using `const_cast` (see section 3.6).

Incompleteness of the type checker: This category represents instances where reflection was used in the original program. Since reflection is not yet supported by the type checker, this code did not type check. We corrected this error by using `const_cast` (see section 3.6).

6.3.3 Annotation Errors

This category represents errors caused by mistakes committed by us during the annotation process. 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 3, but some errors in the library annotations were discovered while annotating client code.)

6.4 Discussion

The most interesting type checking errors are implementation errors, documentation errors, bad style errors, and inflexibility of the language. 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 type checker, 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 partially completed Daikon experiment, where three bugs and one instance of bad-style code were found, and by the annotation of a portion of `java.util`, where one instance of bad style of coding was discovered with the help of the language.

The costs of ConstJava are two-fold. Firstly, extra time must be devoted in order to use its features. This time cost should be expected to 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 is slight. Secondly, any conservative compile-time analysis (particularly a flow-insensitive type analysis) rejects certain safe code. There were four instances of this problem in 44,000 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 our 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 type checker, 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 type checker to accept these instances of safe code that would otherwise be rejected by the type checker.

7 Related work

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

7.1 C++

C [KR88] and C++ [Str00] provide `const` keywords for specifying immutability The most notable example is C++. 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. The `const_cast` operator was added 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 nonconst one.

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.

7.2 Proposals for Java

Many others before us 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 5 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 7 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 for immutability in Java besides ours, but we were not able to obtain the tool.

8 Conclusion

ConstJava is 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.

The paper presents the ConstJava language, including design tradeoffs and type-checking rules. Experiments show that 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 type checker, the benefits of catching errors at compile-time that would otherwise be exceedingly difficult to detect strongly suggest that using ConstJava is worthwhile.

References

[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.