Lightweight Abstraction for Mathematical Computation in Java

Pavel Bourdykine and Stephen M. Watt

Department of Computer Science
University of Western Ontario
London, Canada
pbourdyk@csd.uwo.ca Stephen.Watt@uwo.ca

Abstract. Many object-oriented programming languages provide type safety by allowing programmers to introduce distinct object types. In the case of Java, having objects as the sole abstraction mechanism also introduces a considerable or even prohibitive cost, especially when dealing with small objects over primitive types. Consequently, Java library implementations frequently avoid abstraction and are not type safe in practice. Many applications, including computer algebra, use values logically belonging to many different non-interchangable types. Languages such as Java are then either unsafe or inefficient to use in these applications. We present a solution allowing type safety in Java with little performance penalty. We do this by introducing a specialzed kind of object that provides distinct types for type checking, but which can always be removed entirely at compile time. In our implementation, programs are compiled twice, first with objects to verify type safety, and then with the objects removed for efficiency. This gives significant performance gains across a range of tests, including the generic SciGMark tests.

1 Introduction

A large part of the art of programming language design lies in how one assembles a multitude of ideas that are in principle distinct into a few simple constructs that work well together. When this is done well, it can be beautiful. When this is done badly, it can make programs inefficient and error-prone. This paper argues that this is what has happened in languages such as Java, where objects are the sole data abstraction mechanism, and we present a solution.

A simple example of separate considerations that can be nicely combined is given by the modern `return` statement. In principle, setting the value of a function and the transfer of control back to the caller are completely separate ideas. Indeed, in older programming languages these were done separately. One might easily want to perform some clean up actions, such as closing files, returning resources or updating global state, after the return value is determined. However, in these cases, using a temporary variable with modern `return` is neither costly nor dangerous. At first sight, using objects as the sole data abstraction mechanism would seem to be a similar happy combination. Abstract data types typically have several fields in a hidden representation and provide operations on the abstract values, just as with classes in an object-oriented world. But there are several problems with this:

Abstraction is not just hiding record fields. Providing data abstraction only via objects forces all thinking about abstraction into the model of field visibility in composite structures. Quite often, one wishes to consider simple values as elements of a distinct type. For example, even though window IDs may be represented as integers, it would enhance program safety if they were treated as a distinct type. Likewise for values in different prime fields should be of different types from the integers and from each other. Additionally, composite data is often not represented as fields in an object. For example, it is not uncommon to represent colors as 32-bit integers with bit fields representing component values. Abstraction can help ensure that only integers intended to be color information are used as such.

Abstraction does not always need dynamic allocation, inheritance, synchronization, or other heavy-weight mechanisms. Sometimes we want abstraction only to ensure that programs do not depend on details that may later change and to enhance safety by ensuring values are not used inappropriately. It may be known from the outset that these values will not ever be used in any fancy ways. For example, we may know that there will never be any derived types from colors, there will be no subtle multiprocessing on single color values, *etc.* Requiring all of these features to be supported on abstract values first has a cost, and second reduces flexibility to have these abstract values treated in other interesting ways.

Abstraction is not used if it is too inefficient or onerous. When data abstraction carries a significant efficiency penalty *and* thought on the part of the programmer, then it is not used.

In languages such as `C++`, where objects and primitive types are on a similar footing, the extra cost need not be large. Nevertheless, even here, mechanisms have been proposed for opaque type definitions in `C++` [1]. In languages such as `Java`, where there is a strong distinction between primitive types and object types, the cost to use objects is many times that of using primitive types. Programs are inefficient, programmers circumvent the type system or both. This has many obvious problems. If Java were not such a widely adopted language, we could reject it as being ill suited in these circumstances. As it is, some solution is needed.

The contribution of this paper is to show how light weight abstraction may be provided in Java. This provides type safety without introducing any significant inefficiencies. It is therefore suitable for creating light-weight abstract types for computationally intensive, efficiency-critical tasks such as computer algebra and scientific computing. Section 2 shows how this may be achieved by introducing object types with sufficient restrictions that they are guaranteed to be removed at compile time. While these ideas are presented for Java, the same ideas could equally well be applied in other settings. Section 3 then describes a tool that implements this mechanism that can be used in conjunction with standard Java compilers. Section 4 presents performance results, comparing the usual use of objects, the present light-weight abstraction mechanism and raw primitive types. These comparisons are made using the SciGMark test suite and details are shown for polynomial and matrix multiplication. Finally Section 5 concludes the paper.

2 Opaque types in Java

To deal with the problems outlined, we introduce the notion of *opaque* types in the Java programming language. These types allow development of Java code that is reusable, elegant, and efficient. *Opaque* types are meant to be used as regular object types that can be represented internally by any other Java type with a focus on representation via primitive built-in Java types. The new types are required to behave and act like regular object types in the way they interact with the Java class hierarchy and the static type-checker. An example of this kind of application may be an object that has a small finite number of different states that can be intuitively represented by a set of bit patterns. Although this can be implemented similarly to something written in assembly language, by using *int* types, resulting in code that is quite efficient, the code's extensibility would suffer. Moreover, like assembly, this type of code is difficult to maintain, and debug[2, 3]. This may lead to errors that could have been easily avoided if object types were used.

This approach encompasses a core notion of *opacity*. High level Java objects do not necessarily have to be represented or compiled as such. Objects simply serve as identification handles for static type-checking prior to compilation. The underlying type of these objects may be anything suitable for internally representing the construction. In this fashion, an alternative *String* object may be represented by a character array allowing for operations very similar to those on strings implemented in C or C++. In turn, a more complex object may be represented by such an alternative *String* thus creating an artificial class hierarchy that remains consistent and type-safe. In this work, however, we are concerned mostly with objects that may be represented by primitive types in order to boost performance.

Along with the optimized version of the *opaque* type the regular unchanged version of the class is kept for reference and debugging purposes. Leaving the user code unchanged after compilation allows for more straight forward top-level design where good Object Oriented Design practices may be followed. The user may also choose to compile the *opaque-typed* code and run it as is, without conversion, in order to ensure correctness. Keeping both versions of the class demonstrates the type safety of *opaque* Java types as either version of the project will produce identical execution results.

In order to implement Java *opaque* types, we introduce a set of type rules that have to be followed in order to use such objects safely and efficiently. These rules may be used by a preprocessor to transform the user's regular objects into those for which the generated code will use the underlying primitive types. We now give a more detailed description of these rules.

2.1 *Opaque* type rules

We use a Java code annotation (called *Opaque*) to identify classes as *opaque* types. Java annotations allow embedding of meta-data directly into Java source code. "Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program."[4] The annotation has a single *String*

type field that denotes the primitive representation type of the opaque object. For example, the annotation `@Opaque("int")` indicates that the object is *opaque* and that its primitive representation is of type *int*. Currently, the annotation field serves as a way to quickly identify the underlying type and speed up *opaque type* file analysis but could be left out in later versions of the solution. The single annotation dictates all the required information to the preprocessor. The next restrictions/rules must be followed in order to guarantee successful conversion consistent with the Java language standard:

- **Rule 1** object must have a single `protected` field of the underlying type unless it is a subclass of an *opaque type*
- **Rule 2** object constructor(s) must be declared `private`
- **Rule 3** all methods accessing or modifying the underlying type field representation must be declared `static` (or `final static` if no subclasses override the methods)

Rule 1 enforces *opaque* type representation and assures that it matches with the type specified by the annotation. The field (from here on in referred to as `rep`) takes place of the *opaque* object whenever it appears in translated user code. It is important that its uses are properly implemented and ensures there are no compilation issues post-conversion. The approach to having a single field for the representation is similar to that used in Aldor [5].

If the new *opaque* object `extends` an *opaque* type (the inheritance property detected by the preprocessing utility), the new object must *not* include a `rep` field in its declaration. The `rep` field is instead inherited from the superclass and bares the same primitive type. This ensures consistency in method inheritance and conversion.

Rule 2 follows the Java convention that only object types require a constructor. Since the new *opaque* object is to be converted to its underlying primitive type representation wherever it is used, its constructor must remain `private`. Creating new instances of the *opaque* object is still possible through the use of the `static` method "New". This method should be implemented by the user as a means of converting from the underlying primitive type to the object type primarily for testing purposes and initial implementation of code that uses *opaque* types. The typical implementation is outlined in Figure 1(a).

Rule 3 places a restriction on the other methods possibly acting on the object representation. Default visibility `static` methods allow inheritance and class access to regularly used operations within the new object. At first glance this may seem limiting for using the object; however, since object instances are all converted to the underlying primitive type, only class methods remain as valid operations that can act upon the object's actual implementation, i.e., its `rep` field. This method declaration simplifies the preprocessor task of handling extension quickly and efficiently and assures the *opaque* object is not inflated by non-static behaviors.

The approach takes advantage of the way Java class hierarchy works by allowing subclasses to preserve the "is a" relationship and properly inherit methods with default visibility. Properties of method overloading are also preserved due to use of default visibility and the requirement of using the class name whenever a method is called.

```
// a. Opaque object,
//    typical "New" implementation
@Opaque("short")
public class MyOpaqueObject {
    protected short rep;
    private MyOpaqueObject(short r) {
        rep = r;
    }
    ...
    public static MyOpaqueObject
    New(short r) {
        return new MyOpaqueObject(r);
    }
}

// b. Opaque object before conversion
@Opaque("int")
public class BaseClass {
    protected int rep;
    private BaseClass(short r) {
        rep = (int) r;
    }
    public static void
    operator(BaseClass bc, short modifier) {
        ...
    }
    ...
}
```

```
// c. Opaque object after conversion
public class BaseClass {
    protected int rep;
    private BaseClass(short r){ rep = (int) r; }

    public static void
    operator(int bc, short modifier) {
        ...
    }
    ...
}
// d. Regular main class
@Opaque("user")
public class TopLevel {
    public static void main(String[] args){
        OpaqueType var = OpaqueType.New(param);
        ...
    }
}
// e. Opaque array initialization
@Opaque("user")
public class TopLevel {
    public static void main(String[] args){
        OpaqueType[] ots = new OpaqueType[DATA_SIZE];
        for(int i=0; i < DATA_SIZE; i++)
            ots[i] = OpaqueType.New(param);
        ...
    }
}
```

Fig. 1. Opaque object creation and use

2.2 Converted classes

The style for new object creation is modified slightly when attempting to make use of opaque types. It is useful to illustrate exactly what changes in the type declaration following invocation of the code conversion utility. Figures 1(b) and 1(c) show the original version of a simple class and its converted result respectively.

In Figure 1(b), object `BaseClass` is a Java opaque type represented by the built-in Java `int` type. The important details to notice regarding this class are its `protected int rep` field, `private` constructor and `static void operator` method. These three points are required by the semantic rules outlined in Section 2.1 and allow `BaseClass` to be converted and compiled as its underlying built-in type (`int`).

The converted `BaseClass` can be seen in Figure 1(c). In the new version of the class, the *@Opaque("int")* annotation has been removed, and the `static` method `operator` has been modified to use only arguments of the proper underlying types. The class retains its "high level" handle - `BaseClass`. Hence the user code that makes use of the class only needs minor typing modifications.

2.3 Opaque user classes

Java classes that declare or make use of opaque objects are also annotated with the *@Opaque* annotation. Instead of an underlying `rep` type, the user classes contain the keyword "user" as the single parameter to the annotation. A user class may look as simple as in Figure 1(d).

Classes declaring objects of opaque type that are not opaque themselves (annotated as *@Opaque("user")*) undergo only minor changes during the conversion step. Opaque types are constructed in a way such that their declaration, initialization and usage do not require any object-exclusive syntax aside from declaring

data structures whose elements are opaquely typed. The most common and primitive of these data structures is an array. Declaration of Java arrays containing opaque typed members is syntactically identical to any other array declaration (for any number of dimensions). Initialization, however, is dictated by the nature of the opaque members themselves - each opaque object is initialized via the `New` method as opposed to using the Java `new` keyword. This simple yet notable concept is summarized in Figure 1(e).

2.4 Annotation processing example

The structure that opaque annotations impose on Java source code is non-hierarchical despite playing a role in the hierarchical class structure of Java. The traversal through this construction of annotated source files is straight forward for the most part and the conversions applied to the code are often influenced directly by information contained in the same processing step. Figure 2 illustrates some subtleties during the conversion step that arise when analyzing a deep class hierarchy for opaque objects.

```
@Opaque("int")
public class BaseClass {
    protected int rep;
    private BaseClass(short r){
        rep = (int) r;
    }
    public static void operator
       (BaseClass bc,
        short modifier)
    { ... }
    ...
}

@Opaque("int")
public class ChildClassOne
   extends BaseClass
{
    private ChildClassOne(short r){
        rep = (int) r;
    }
    ...
}
```

```
@Opaque("int")
public class ChildClassTwo
    extends BaseClass
{
    private ChildClassTwo(long r){
        rep = (int) r;
    }
    ...
}

@Opaque("int")
public class ChildClassThree
    extends ChildClassTwo
{
    private ChildClassThree(int r){
        rep = r;
    }
    public static ChildClassThree operator
        (ChildClassTwo modifier, short cc){
        ...
    }
    ...
}
```

Fig. 2. Inheritance during conversion

The analysis of such a hierarchy takes place as follows. The *Opaque*-annoted classes are identified among the source files and a list of them is stored along with their representation (taken from the `String`-type annotation argument). In Figure 2 this list would consist of pairs `BaseClass & int`, `ChildClassOne & int`, `ChildClassTwo & int`, `ChildClassThree & int`. The annotations make the suggested representation clear, but we still have to check that the class does not attempt to use a differently typed field. As you can tell, the underlying representation property is inherited, in this case all the way down the hierarchy from `BaseClass`. Method inheritance is taken care of by standard Java, for example, the `operator` method of `ChildClassThree` is overwritten for only that

class and the overloaded method works as expected. The goal of our approach is to make the preserved object properties as intuitive as possible (i.e., make them work as the programmer would expect) during application of the *opaque* types mechanism.

3 Java Implementation

Implementing opaque Java types requires careful considerations in order to abide by the set restrictions and still make the new data declaration forms useful. For performance, ideally, the underlying representation of a particular type could be determined during the compilation process and the underlying type used for code generation. This kind of automated optimization would mean a seamless implementation of a significant performance gain. However, allowing the programmer to specify the underlying type of the opaque object allows for greater flexibility for accomplishing a certain task, even if at the cost of some efficiency.

A sophisticated mechanism to determine the underlying object representation on the fly could be an area of significant research, however, at this time we have elected to make the choice explicit. Hence the steps to build projects containing opaque types are quite straight forward.

Code conversion utility In order to realize the potential of Java *opaque* types we need to develop a dual view of the annotated objects to the compiler. The first is the object view – necessary to take advantage of Java's inherent ability to handle a rich type hierarchy. The second is the underlying representation view, the one to be used during optimization and code generation phases of compilation. This dual representation is achieved using a code conversion utility written in Java itself making use of `Pattern` and `Matcher` classes from the `java.util.regex` package.

These classes provide a convenient way to identify where and how *opaque* types are used and apply conversions directly to Java source code. This allows the utility to finish its tasks in a timely manner without complicating the process of going from regular-looking objects (*opaque* types) to the immediate underlying representation.

The utility performs the following steps:

1. identify all recently modified Java source files in target project
2. sort source files into regular, opaque typed, and opaque user classes
3. build record of all opaque types and their underlying representations
4. convert all opaque sources

Automating the building process Utilizing a pre-processor-like code conversion application prior to compilation complicates the building process by adding a necessary intermediate step to the routine mechanism. However, Java is a flexible language with a relatively long standing industry and research history. By this virtue a number of tools have been developed that augment various features of the language in particular when it comes to its compilation and building process.

One of such tools is the Ant scripting language[6]. We use an Ant build script to perform the following tasks:

1. back up original source files
2. invoke converter on files modified since last invocation
3. compile newly converted files

Eclipse IDE Integration into a main-stream development environment may seem like an extraneous task; however, this discussion follows naturally due to the Eclipse's ability to use Ant build scripts instead of the default compiler or build-chain. Implementing the build script directly into the Eclipse IDE allows the user to seamlessly develop code utilizing *opaque* types in the IDE.

Further details of the implementation are described in the first author's master thesis [7].

4 Performance Results

If regular Java objects performed as well as built-in types, there would be no need to invent a new mechanism for abstraction. This, however, is not the case. Primitive types in Java perform far better than objects.

We consider the overall application performance for synthetic tests by the time it takes the program to execute, and the memory consumed during its execution. Computational benchmark performance is compared using the number of floating operations per second performed by various implemented algorithms. We compare performance of Java code using regular objects, code which has been converted to use *opaque* objects, and code implemented with the use of primitive types only (dubbed "specialized"). For the purposes of measuring performance in such a way we have devised several synthetic tests that demonstrate *opaque* type advantages using brief implementations and included two modified benchmarks from the SciMark[8] and SciGMark[9] performance benchmark suites. The measurements for testing performance that could be adjusted to utilize *opaque* types most naturally have been included in this report. The particular benchmarks chosen from SciMark 2.0 and SciGMark 1.0 suites are dense polynomial multiplication with integer field coefficients originally developed for SciMark benchmark and modified by SciGMark and sparse matrix multiplication with real coefficient values.

The modified applications accomplish identical tasks and have minimal implementation differences aside from the use of *opaque* types and corresponding annotations. Along with the borrowed benchmarks, the synthetic tests that range from simple classes implementing only a few methods with shallow class hierarchy to classes with a large internal representation (e.g. a large integer array), several constructors, and a large number of methods are used to measure "bare bone" performance. All tests were executed 10 times in order to compensate for varying CPU and memory loads on different platforms. The averaged results were recorded and are shown next. The computationally intensive benchmarks

were executed on large data sets in order to maximize the effect of data allocation and access on performance when dealing with objects versus more primitive structures in large quantities. This in turn increased result accuracy due to floating point operations being used as the measurement units. The simple tests were chosen to reflect varying uses and applications developers may encounter when writing Java code for a typical project.

Benchmark implementations were tested on several different platforms in order to demonstrate *opaque* types' independence of environment when increasing computational performance. The platforms used for testing were as follows:

- Intel C2Q Q6600 @ 2.4GHz, 4GB RAM, Windows 7 x86_64, JRE 1.7 (**lambda**)
- Intel I7-870 @ 2.93GHz, 16GB RAM, Ubuntu Server 10.04 x86_64, JRE 1.6 (**tedium**)
- Intel Xeon E5620 @ 2.4GHz, 24GB RAM, Ubuntu 10.04 x86_64, JRE 1.6 (**z600**)
- Intel I5-660 @ 3.33GHz, 4GB RAM, Ubuntu 10.04 x86_64, JRE 1.6 (**PCA-45**)
- Intel C2D E4600 @ 2.4GHz, 2GB RAM, Ubuntu 10.04 x86_64, JRE 1.6 (**orccapc02, orccapc03, orccapc04**)

Running of experiments on different platforms has also given us an opportunity to look at the variance in underlying software that affects the performance of Java applications using *opaque* types. The results were not significantly impacted by execution on different platforms, and even the JVM versions used did not incur a great deal of variance on the results.

Execution time and memory use were measured using built-in Java tools for determining system time (method `currentTimeMillis()` in `java.lang.System`), and tools for determining how memory is currently used by the Java Virtual Machine - `Runtime` methods called `totalMemory()` and `freeMemory()`. All tests measuring memory use were carefully designed to avoid involuntary garbage collection and execution time tests were averaged to account for varying CPU load during the experiments and were generally run at the highest CPU affinity.

Complex internal representations Similarly to the *opaque* types used through this work, it is possible to represent object types by a single primitively typed array fields of fixed size. For example, an *opaque* type object may be represented by 256 bits, or an array of size 4 of type `long[]`. The next set of tests deals with objects represented by different sized arrays of primitively typed variables. The tests use the same, previously shown, metrics to measure execution speed and memory use. The implementation of the actual accomplished operation is kept as identical as possible to avoid performance differences due to algorithmic discrepancies. This assures that we compare directly the speed and size of regular objects versus *opaque* objects without introducing unnecessary bias.

Figure 3 illustrates an *OpaqueObject* represented by the `long[]` type and a *RegObject* that has a field of type `long[]`. Both objects have the similarly implemented method called *setBit*. Method *setBit* takes an argument of type `int` that corresponds to the bit number that must be turned on in the internal representation of the *OpaqueObject* or the field of the *RegObject* with 0 being the least significant bit. Imagine arranging either the internal `long[]` representation of *OpaqueObject* or the field of *RegObject* as sets of back-to-back 64 bit sets (each

```
@Opaque("long[]")
public class MyOpaqueObject {
    protected long[] rep;
    private OpaqueObject(long[] arg){
        rep = new long[arg.length];
        for (int i = 0; i < arg.length; i++)
            rep[i] = arg[i];
    }
    public static OpaqueObject
    New(long[] arg) {
        return new OpaqueObject(arg);
    }
    public static OpaqueObject
    setBit(OpaqueObject o, int i) {
        long mask = (long) (1 << (i % 64));
        o.rep[i / 64] |= mask;
        return o;
    }
    ...
}
```

```
public class RegObject {
    private long[] rep;
    public RegObject(long[] arg){
        rep = new long[arg.length];
        for (int i = 0; i < arg.length; i++)
            rep[i] = arg[i];
    }
    public void setBit(int i){
        long mask = (long) (1 << (i % 64));
        rep[i / 64] |= mask;
    }
    ...
}
```

Fig. 3. Regular and *opaque* objects with array typed fields

set represented by a `long` type value) where significance of the bits increases with the array index of the respective field. Thus operation *setBit* is potentially able to turn on a single bit in a bit set of size over 2,000,000,000. For the test, however, we limit the size of the `long` array to 4.

Putting it together The next set of performance comparison tests consists of two standard benchmarks taken from the SciMark and SciGMark suites. In order to implement polynomial multiplication and sparse matrix multiplication benchmarks we build on the conventions established previously and reuse some implementations from the synthetic benchmarks.

The first test performed is sparse matrix multiplication with double precision floating point coefficients taken randomly from the complex number set. This is one of the most natural performance indicators for a language feature or a hardware benchmark. In this case, the test's aim is to demonstrate that

	Code			**Improvement**	
PC	**Generic** (mflops)	**Specialized** (mflops)	**Opaque** (mflops)	**Opaque** vs. **Generic**	**Opaque** vs. **Specialized**
lambda	131.84	475.22	383.64	2.91	0.81
tedium	194.64	1199.08	968.4	4.98	0.81
z600	175.16	1044.22	833.06	4.76	0.80
PCA-45	158.82	1077.7	845.84	5.33	0.78
orccapc04	57.20	363.5	303.68	5.31	0.84
orccapc03	62.34	371.1	299.45	4.80	0.81
orccapc02	60.94	368.4	301.94	4.95	0.82
sodium	54.82	311.26	248.48	4.53	0.80
Overall average improvement:				4.70	0.81

Table 1. Matrix Multiplication

	Code			Improvement	
PC	**Generic** (mflops)	**Specialized** (mflops)	**Opaque** (mflops)	**Opaque** vs. **Generic**	**Opaque** vs. **Specialized**
lambda	75.54	279.56	223.86	2.96	0.80
tedium	147.02	900.44	729.06	4.96	0.81
z600	131.32	800.68	639.42	4.87	0.80
PCA-45	136.54	910.64	723.38	5.30	0.79
orccapc04	56.29	355.15	285.91	5.08	0.81
orccapc03	54.98	350.90	288.3	5.24	0.82
orccapc02	57.84	355.32	287.92	4.98	0.81
sodium	38.90	223.68	179.52	4.61	0.80
Overall average improvement:				4.75	0.81

Table 2. Polynomial Multiplication

it is possible to significantly increase the raw number of floating point calculators (measured here in millions of floating point operations per second) without losing correctness by reducing abstraction (or removing it altogether) in the implementation. Unfortunately, fully disposing of abstraction, as SciGMark implementation shows, highly obscures the code. Use of primitive types yields high performance and optimized execution, however, the code becomes more complex and is difficult to modify. The matrix sizes used for measurement are `N` by `N` matrices with $\texttt{N} = 10,000$ averaging $100,000$ non-zero coefficients per matrix.

The purpose of the implementation using *opaque* types is to preserve abstraction introduced by the generic object implementation utilized by SciGMark while pushing performance figures towards that of the specialized code. Figures 4(a), 4(b), 4(c) show snippets of the code implementing the underlying complex data and algorithms used in this benchmark. The included code shows implementation of the type creation, summing, and multiplication.

Performance results using these varying implementations of the complex data types are summarized in Table 1. Analyzing the data it's easy to conclude that without loss of much generality, the *opaque* implementation is on average 4.7 times faster than the general object implementation and is only about 20% slower than the specialized implementation from Figure 4(b).

The second benchmark used in our final set of tests is polynomial multiplication with dense polynomials of degree ≤ 40. The polynomial coefficients are once again taken from the complex set and are implemented in three different ways according to each multiplication algorithm (generic objects, specialized, *opaque*). Table 2 summarizes obtained results measured in millions of floating point instructions per second with similar conclusions being drawn from this set of data as the sparse matrix multiplication.

Implementing dense polynomial multiplication using the proposed *opaque* typed method allows for an average of 4.75 times the number of operations per second while accomplishing the same task. The *opaque* implementation loses out to specialized code by an average of only 19%. This is an expected and impressive

```
// a. Generic object implementation
public class Complex <R extends IRing<R>> {
    private R re;
    private R im;
    public Complex<R> create(R re, R im) { return new Complex<R>(re, im); }
    public Complex<R> s(Complex<R> o) { return new Complex<R>(re.s(o.re()),im.s(o.im())); }
    public Complex<R> m(Complex<R> o) {
        return new Complex<R>(re.m(o.re()).s(im.m(o.im())), re.m(o.im()).a(im.m(o.re())));
    }
}
// b. Specialized implementation
public class Complex {
    private double re;
    private double im;
    public Complex create(double re, double im) { return new Complex(re, im); }
    public Complex s(Complex o) { return new Complex(re + o.getRe(), im + o.getIm()); }
    public Complex m(Complex o) {
        return new Complex(re*o.getRe() + im*o.getIm(), re*o.getIm() + im*o.getRe());
    }
}
// c. Opaque object implementation
@Opaque("double[]")
public class Complex {
    protected double[] rep;
    public static Complex create(double re, double im) { return Complex.New(re, im); }
    public static Complex s(Complex o) { return Complex.New(s.rep[0]+o.rep[0], s.rep[1]+o.rep[1]); }
    public Complex m(Complex o) {
        return new Complex(s.rep[0]*o.rep[0]+s.rep[1]*o.rep[1], s.rep[0]*o.rep[1]+s.rep[1]*o.rep[0]);
    }
}
```

Fig. 4. Multiplication: generic, specialized and opaque object implementation

result considering how much abstraction is preserved through the use of *opaque* types.

5 Conclusions and Further Directions

We have observed that Java programmers and library designers have been forced to work around the language's abstraction mechanisms for performance-sensitive code. In practice, programs have used primitive types, such as `int`, when an abstraction should be used. The recent addition of `Enumeration`s to the language help in some settings, but is of no help when the values are used in mathematical computations.

We have shown how type-safe, but very efficient programs may be obtained with the concept of an *opaque* type in Java. An *opaque* type is distinct and incompatible with its underlying representation type, which may be a primitive type or an object type. We have shown how *opaque* types may be provided via classes with only static methods, and annotated for handling with a software tool in a standard Java environment. *Opaque* types are type checked as though they were object types, but compiled as the actual representation values. This allows *opaque* values to benefit from all the optimizations on primitive types without relying on sophisticated data structure elimination optimizations.

At the moment, our software tool operates by compiling the code twice, but of course this could easily be integrated into any compiler. While we focus on

Java for practical reasons, we expect the same observations and techniques to be directly applicable in other similar settings.

References

1. Brown, W., E. Progress toward Opaque Typedefs for C++0X. 2005.
2. Johnston, B. Java programming today. Upper Saddle River, NJ; Pearson Prentice Hall, 2004.
3. Koffman, E., B. Objects, abstraction, data structures and design using Java. John Wiley and Sons, 2005.
4. Sun Microsystems, Inc. Annotations. 2004. `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`
5. Watt, S.M. *Aldor* in Handbook of Computer Algebra. 265–270. Grabmeier, J., Kaltofen, E., Weispfenning V. (eds), Springer Verlag, Heidelberg. 2003
6. The Apache Ant Project. 2010. `http://ant.apache.org`
7. Bourdykine, P. Type Safety without Objects in Java, MSc. Thesis, U. Western Ontario, 2009.
8. Miller, B., Pozo, R. SciMark 2.0 Java Benchmark. National Institute of Standards and Technology, 2004.
9. Dragan, L., Watt, S.M. Performance Analysis of Generics for Scientific Computing, 93–100, Proc. SYNASC 2005, IEEE Press.