Complex numbers for Java

Michael Philippsen and Edwin Günthner

Computer Science Department, University of Karlsruhe
Am Fasanengarten 5, 76128 Karlsruhe, Germany
phlipp@ira.uka.de, edwin.guenthner@gmx.de
http://wwwipd.ira.uka.de/JavaParty/

Abstract. Efficient and elegant complex numbers are one of the preconditions for the use of Java in scientific computing. This paper introduces a preprocessor and its translation rules that map a new basic type `complex` and its operations to pure Java. For the mapping it is insufficient to just replace one `complex`-variable with two `double`-variables.
Compared to code that uses `Complex` objects and method invocations to express arithmetic operations the new basic type increases readability and it is also executed faster. On average, the versions of our benchmark programs that use the basic type outperform the class-based versions by a factor of 2 up to 21 (depending on the JVM used).

1 Introduction

In regular Java there is just one reasonable way to use complex numbers, namely to write a class `Complex` containing two values of type `double`. Arithmetic operations have to be expressed by method invocations as shown in the following code fragment. The alternative, to manually use two `double`-variables where a complex number is needed, is too error-prone and too cumbersome to be acceptable.

```
Complex a = new Complex(5,2);
Complex b = a.plus(a);
```

Class-based complex numbers have three disadvantages: Once written without operator overloading, arithmetic operations are hard to read and maintain. Second, since Java does not support so-called value classes, object creation is slower and objects need more memory than variables of a basic type. Arithmetic operations based on classes are therefore much slower than arithmetics on built-in types. Even worse, method-based arithmetic causes frequent creation of temporary objects to return values. To return temporary arithmetic results with basic types, no such object creation is needed. The third disadvantage is that class-based complex numbers do not seamlessly blend with basic types and their relationships. For example, an assignment of a `double`-value to a `Complex`-object will not cause an automatic type cast – although such a cast would be expected for a genuine basic type `complex`. Additionally, there is no natural way to express complex literals; instead a constructor call is needed.

The fraction of people using Java for scientific computing is quite small, so it is unlikely that the Java Virtual Machine (JVM) or the Java bytecode will be extended to support a basic type `complex` – although this might be the best solution from a technical point of view. It is also hard to tell whether Java will ever be extended to support operator overloading and value classes; and if so, whether there will be efficient implementations. But even given such features our work would remain important because, first, the same level of seamlessness cannot be achieved, see the above type cast problem. And second, our work can still be used to rate the efficiency of implementations of the general features.

The next section discusses the related work. Section 3 gives an overview of `complex` numbers in the *cj* preprocessor/compiler. The central ideas of the translation are presented in Section 4. Section 5 shows the quantitative results.

2 Related work

With support from Sun Microsystems, the Java Grande Forum [7, 13] strives to improve the suitability of Java for scientific computing. The challenge is to identify and bundle the needs of this small user group in such a way that they can be respected in the continuing evolvement of Java although that is driven by the main stream.

The Java Grande Forum is working on a reference implementation of a class `Complex` that can be used to express arithmetics on complex numbers [14, 6]. Special attention is paid to problems of numerical stability. IBM is extending their Java-to-native compiler to recognize the use of this class [15]. By understanding the semantics of the `Complex` class, the compiler can optimize away method invocations and avoidable temporary objects. Hence – at least on some IBM machines – high performance can be achieved even when using class-based complex numbers. However, the other disadvantages mentioned above still hold, i.e. there is no operator overloading and `Complex` objects lack a seamless integration into the basic type system.

There are considerations to add value classes to the official Java language [5, 12]. But although there is no proper specification and no implementation yet, the Borneo project [3] is at least in a stage of planning. Since there are already object-oriented languages that support value classes, e.g. Sather [11], the basic technical questions of compiling value classes to native code can be regarded as solved.[1] However, it is still unclear whether and how value classes can efficiently be added to Java by a transformation that expresses value classes with original language elements. In particular, it remains to be seen whether value classes will require a change of the bytecode format.

The discussion on how to add a primitive type `complex` to a given language has a long tradition. Java is special because of its very strict definitions of evaluation orders, in particular with respect to the visibility of side effects in case of exceptions. A related question of extending a language with a primitive type

[1] C++ can emulate value class semantics by means of pass-by-value mechanisms.

`complex` is whether there should be a separate primitive type for imaginary numbers as well, as proposed by Kahan [9]. It is debated whether and how such a separate imaginary type should be included in the C9X proposal [2]. To avoid the complexity of an extra type, our work only adds the type `complex` although future extensions might have the `imaginary` type as well.

3 Complex numbers in *cj*

Cj extends the set of basic types by the type `complex`. A value of type `complex` represents a pair of two double precision floating point numbers. All the common operations for basic types are defined for `complex` in a straightforward way. The real and imaginary part of a `complex` can be accessed through the member fields `real` and `imag`. Note, that the names of these fields are not new keywords. Since the basic type `complex` is a supertype of `double`, a `double`-value will be implicitly casted where a `complex` is expected. A second new keyword, `I`, is introduced to represent the imaginary unit and to express constant expressions of type `complex`, as shown in the following code fragment.

```
void foo(complex x, complex y) {
  complex const = 5.0 + y.real * I;
  complex sum   = const + x + y;
  ...
```

In our experience, any extension of Java will only be accepted if there is a transformation back to pure Java. But in general better efficiency can be achieved by using optimization techniques during bytecode generation. Our compiler *cj*, which is an extension of *gj* [1], therefore supports two different output formats, namely Java bytecode and Java source code. In this paper, we focus on the latter and mention optimizations only briefly.

4 Recursive transformation rules for `complex`

In any current Java compiler, inner classes are transformed to earlier Java 1.0 (without inner classes). Similarly, *cj* has another transformation phase that resolves `complex` numbers and `complex` operations. This section discusses the central ideas of this transformation.

4.1 Name mangling for separate compilation

The basic part of the transformation is done by modifying the names used by the programmer. For example, by appending `$cj$real` to a variable name, the name of the resulting real part of the corresponding `complex`-variable is derived. A similar technique is used to adapt the names of methods when their signature includes arguments of type `complex`.

By making the name mangling rules also known to the bytecode loader it is possible to compile different Java programs separately, even when `complex` is already transformed into bytecode. Hence, if the compiler finds a mangled method definition in a bytecode file it internally generates an additional method symbol entry representing the method with its original signature and type information.

4.2 Transformation of `complex` expressions

The central part of the transformation deals with `complex` expressions. For simplicity, we call any expression that uses `complex` values a `complex` expression. The transformation of `complex` expressions to expressions that only use `double` causes several problems.

Transformation locality. If a `complex` expression is used where only an *expression* is allowed it must not be mapped to a sequence of statements.

```
while (u == v && x == (y = foo(z))) {...}
```

For example, for transforming the `complex` condition of this `while`-loop into its real and imaginary parts it is necessary to introduce several temporary variables whose values have to be calculated within the body of the loop. Therefore, the loop has to be reconstructed completely.

In general, to replace `complex` expressions by three-address statements, one needs non-local transformations that reconstruct surrounding statements as well, although local transformation rules that replace expressions by other expressions (not statements) were simpler to implement in a compiler and it would be easier to reason about their correctness.

Semantics. To achieve platform independence, Java requires a specific evaluation order for expressions (from left to right). Any transformation of `complex` arithmetics must implement this evaluation order using `double`-arithmetic.

To preserve these semantics for `complex` expressions, it is not correct to fully evaluate the real part before evaluating the imaginary part. Instead, the transformation has to achieve that a side effect is only visible on the right hand side of its occurrence, but both for the real and imaginary part. Similarly in case of an exception, only those side effects are to become visible that occur on the left side of the exception. Additionally, by separating the real part from the imaginary part, it is also unclear how to treat method invocations (`foo(z)` in the above example). Shall `foo` be called two times? Is it even necessary to create two versions of `foo`?

4.3 Sequence methods

To avoid both types of problems we introduce what we call *sequence methods* as a central idea of *cj*. Each `complex` expression is transformed into a sequence of expressions. These new expressions are then combined as arguments of a sequence method. The return value of a sequence method is ignored. This technique enables us to keep the nature of an expression and allows our transformation to be local. A sequence method has an empty body; all operations happen while evaluating the arguments of the method invocation. The arguments are evaluated in typical Java ordering from left to right.[2] In case of nested expressions, the arguments of a sequence method invocation are again invocations of sequence methods. By using this concept we are able to evaluate both parts of each node of a `complex` expression tree at a time. Moreover, the evaluation order (in terms of visibility of side effects and exceptions) is guaranteed to be correct.

When *cj* is used as preprocessor and Java code is produced, the method invocations of the sequence methods – which are declared `final` in the surrounding class – are not removed. However, they may be inlined by a Just-in-time (JIT) compiler. When *cj* generates bytecode, the compiler directly removes the method invocations – only the evaluations of the arguments remain. The resulting bytecode has the same efficiency as if C/C++'s comma operator was available in Java.

4.4 An example of Sequence methods

Let us first consider the right hand side of the `complex` assignment `z = x + y`. To avoid any illegal side effects we use temporary variables to store all operands. The following code fragment shows the (yet unoptimized) result of the transformation of the right hand side.

```
seq(seq(tmp1_real = x_real, tmp1_imag = x_imag),
    seq(tmp2_real = y_real, tmp2_imag = y_imag),
    tmp3_real = tmp1_real + tmp2_real,
    tmp3_imag = tmp1_imag + tmp2_imag)
```

In this example, 6 `double`-variables would have to be declared in the surrounding block (not shown in the code). When evaluating this new expression, Java will start with the inner calls of sequence methods (from left to right). Thus, both parts of `x` and `y` are stored in temporary variables. The subsequent call of the enclosing sequence method performs the addition (in the third and fourth argument). A subsequent basic block optimization detects the copy propagation and eliminates passive code. So we only need a minimal number of temporary variables and copy operations. In the example, just two temporary variables and one sequence method remain.

```
seq(tmp3_real = x_real+y_real, tmp3_imag = x_imag+y_imag)
```

Now look at the assignment to `z` and the two required elementary assignments.

[2] Exceptions are not thrown within a sequence method but within the invocation context. Hence, it is unnecessary to declare any exceptions in the signature of sequence methods.

```
seq(seq(tmp3_real = x_real+y_real, tmp3_imag = x_imag+y_imag),
    z_real = tmp3_real, z_imag = tmp3_imag)
```

In this case the basic block optimization also reduces the number of temporary variables and prevents the declaration of a sequence method. Thus, the resulting Java code does not need any temporary variables; only a single sequence method needs to be declared in the enclosing class.[3]

```
seq(z_real = x_real + y_real, z_imag = x_imag + y_imag)
```

When we directly construct bytecode, there is no need for the sequence method. Instead, only the arguments are evaluated. The resulting bytecode is identical to the one that would result from a manual replacement of complex expressions with three-address statements.

4.5 Basic transformation rules in detail

In the next sections we consider an expression E that consists of subexpressions e_1 through e_n. The rewriting rule $eval[E]$ describes (on the right hand side of the $\mapsto$-symbol) the recursive transformation into pure Java that applies $eval[e_i]$ to each subexpression. In most cases complex expressions are mapped to calls of sequence methods whose results are ignored. Sometimes it is necessary to access the real or imaginary part of a `complex` expression. For this purpose there are $evalR$ and $evalI$. Both cause the same effect as $eval$ but are mapped to special sequence methods (`seqREAL` or `seqIMAG`) that return the real or the imaginary part of the `complex` expression. If $evalR$ or $evalI$ are applied to an array of `complex`-values the corresponding sequence methods will return an array of `double`-values. Expressions that are not `complex` remain unchanged when treated by $eval$, $evalR$, or $evalI$. The =-symbol refers to Java's assignment operator. In contrast, we use $\equiv$ to define an identifier (left hand side of $\equiv$) that has to be expanded textually by the expression on the right hand side.

To process the left hand side of assignments we use another rewriting rule: $access[E]$ does not return a value but instead returns the shortest access path to a subexpression, requiring at most one pointer dereferencing.

From the above example, it is obvious that a lot of temporary variables are added to the block that encloses the translated expressions. Most of these temporary variables are removed later by optimizations.[4] The following transformation rules do not show the declaration of temporary variables explicitly. However, they can easily be identified by means of the naming convention: if e is a `complex` expression, the identifiers e_{real} and e_{imag} denote the two corresponding temporary variables of type `double`. The use of any other temporary variables is explained

[3] Since user defined types may appear in the signature of sequence methods it is impossible to predefine a collection of sequence methods in a helper class.

[4] In case of static code or the initialization of instance variables the remaining temporary variables are neither static nor instance variables: they can be converted to local variables by enclosing them with static or dynamic blocks.

in the text. Arrays of `complex` are discussed in Section 4.6; method invocations are described in Section 4.7. The rules for unary operations, constant values, and literals are trivial and will be skipped. Details can be obtained from [8].

Plain identifier: The transformation rule for $E \equiv c$ is:

$$eval[c] \mapsto seq(E_{real} = c_{real}, E_{imag} = c_{imag})$$

Both components of the `complex` variable c are stored to temporary variables that represent the result of the expression E. If c is used as left hand side of an assignment, it is sufficient to use the mangled names.

Selection: The transformation rule for $E \equiv F.e$ is:

$$eval[F.e] \mapsto seq(tmp = eval[F], E_{real} = tmp.e_{real}, E_{imag} = tmp.e_{imag})$$

F is evaluated once and stored in a temporary variable *tmp*. Then *tmp* is used to access the two components.

If $F.e$ is used as the left hand side of an assignment, F is evaluated to a temporary variable that is used for further transformations of the right hand side:

$$\begin{aligned} access[F.e] \mapsto\ & tmp = eval[F] \\ & \wedge\ E^{\downarrow}_{real} \equiv tmp.e_{real}, E^{\downarrow}_{imag} \equiv tmp.e_{imag} \end{aligned}$$

It is important to note that the transformation rule for assignments (see below) demands that the code on the right hand side of the $\mapsto$-symbol is inserted at the position where $access[F.e]$ is evaluated. Secondly, the identifiers $E^{\downarrow}_{real}$ and $E^{\downarrow}_{imag}$ have to be replaced textually with the code following the $\equiv$-symbol. (The $\downarrow$-notation and the textual replacement are supposed to help understanding by clearly separating the issues of the access path evaluation from the core assignment. See example below.)

Assignment: The transformation rule for $E \equiv e_1 = e_2$ is:

$$\begin{aligned} eval[e_1 = e_2] \mapsto seq(\ & access[e_1], eval[e_2], \\ & E_{real} = e^{\downarrow}_{1real} = e_{2real}, E_{imag} = e^{\downarrow}_{1imag} = e_{2imag}) \end{aligned}$$

First the access to e_1 is processed. Then the right hand side of the assignment is evaluated. The last two steps perform the assignment of both parts of the `complex` expression. Since the assignment itself is a Java expression it is necessary to initialize additional temporary variables that belong to E. Occurrences of $e^{\downarrow}$ are inserted textually according to *access*.

For example, consider the following statement `X.Y.z = x`. With $e_1 =$ `X.Y.z` and $e_2 =$ `x`, the transformation results in:

$$eval[e_1 = e_2] \mapsto seq(access[\texttt{X.Y.z}], eval(\texttt{x}), \ldots)$$

The evaluation of *access* needs the transformation rules for selections, see above.

$$tmp = eval[\texttt{X.Y}] \text{ and } E^{\downarrow}_{real} \equiv tmp.z_{real}, E^{\downarrow}_{imag} \equiv tmp.z_{imag}$$

We ignore the fact that `X.Y` is itself is a selection and that it would require to apply the same transformation rule again. The assignment to `tmp` is the immediate result of *access* and will show up in the sequence method's first argument. Since *access* has been applied on e_1, the $\downarrow$-expressions are to be inserted textually for every occurrence of e_{1real} and e_{1imag} in the subsequent arguments of the sequence method:

$$eval[e_1 = e_2] \mapsto seq(\ldots, E_{real} = \underbrace{tmp.z_{real}}_{e^{\downarrow}_{1real}} = e_{2real}, E_{imag} = \underbrace{tmp.z_{imag}}_{e^{\downarrow}_{1imag}} = e_{2imag})$$

Since in the example the assignment is a statement there is no need to actually assign to E_{real} and E_{imag}. Hence, after removing temporary variables and redundant calls of sequence methods, the final result is:

```
seq(tmp = X.Y, tmp.z_real = x_real, tmp.z_imag = x_imag)
```

Combination of assignment and operation: The transformation rule for $E \equiv e_1 \diamond = e_2$, where $\diamond \in \{+, -, *, /\}$, is:

$$eval[e_1 \diamond = e_2] \mapsto seq(access[e1], e_1^{\downarrow} = eval[e_1^{\downarrow} \diamond e_2])$$

The address of the left hand side of the assignment is evaluated; every occurrence of $e_1^{\downarrow}$ is replaced textually with the code determined by *access*. The address is used as left operand of the operation. Finally, the result of the operation is written to the calculated address. This strategy is essential to avoid repetition of side effects while evaluating e_1.

Comparison: The transformation rule for $E \equiv e_1 == e_2$ is:

$$\begin{aligned} eval[e_1 == e_2] \mapsto seq_{value}(\ & eval[e_1], eval[e_2], \\ & e_{1real} == e_{2real} \ \&\& \ e_{1imag} == e_{2imag}) \end{aligned}$$

In contrast to the sequence methods used before, this one is not returning a dummy value. Instead seq_{value} returns the value of its last argument. The result of the whole expression is a logical AND of the two comparisons. Inequality tests can be expressed in the same way, we just have to use $!=$ and $||$ instead of $==$ and &&. This special kind of sequence method can also be removed while generating bytecode.

Addition and subtraction: The transformation rule for $E \equiv e_1 \diamond e_2$, where $\diamond \in \{+, -\}$, is:

$$\begin{aligned} eval[e_1 \diamond e_2] \mapsto seq(eval[e_1], eval[e_2], \ & E_{real} = e_{1real} \diamond e_{2real}, \\ & E_{imag} = e_{1imag} \diamond e_{2imag}) \end{aligned}$$

Multiplication: The transformation rule for $E \equiv e_1 * e_2$ is:

$$\begin{aligned} eval[e_1 * e_2] \mapsto seq(eval[e_1], eval[e_2], \ & E_{real} = e_{1real} * e_{2real} + e_{1imag} * e_{2imag}, \\ & E_{imag} = e_{1real} * e_{2imag} - e_{1imag} * e_{2real}) \end{aligned}$$

Division: The rule for division is structurally identical to the rule for multiplication but the expressions are considerably more complicated. *cj* offers two versions to divide `complex` expressions: a standard implementation and a slower but numerically more stable version. The second alternative is based on the reference implementation [14]. For brevity, neither of the versions is shown.

Type cast: Because `complex` is defined as a supertype of `double`, implicit type casts are inserted where necessary. Furthermore, it is appropriate to remove explicit type casts to `complex` if the expression to be casted is already of type `complex`. The only remaining case ($E \equiv (complex)\ e$) can be handled with the following rule:

$$eval[(complex)\ e] \mapsto seq(eval[e], E_{real} = e, E_{imag} = 0)$$

String concatenation: Since string concatenation is not considered as time-critical *cj* creates an object of type `Complex` and invokes the corresponding method `toString`. An additional benefit is that the output format can be changed without modifying the compiler. The transformation rule is:

$$eval[str + e] \mapsto str + (new\ Complex(evalR[e], e_{imag}).toString())$$

$EvalR[e]$ evaluates e and returns its real part. Furthermore $evalR$ declares a temporary variable e_{imag} and initializes it with the imaginary part of e. This asymmetry is necessary to ensure that e is evaluated exactly once. For brevity, we skip similar rules for $e + str$ and the $+ =$-operation.

4.6 Transformation rules for arrays

Although it is obvious that a variable of type `complex` must be mapped to a pair of two `double`-variables, there is no obvious solution for arrays of `complex`. There are two options: an array of `complex` can either be replaced by two `double`-arrays or by one `double`-array of twice the size. For the latter, our performance measurements indicated that in general it is faster to store pairs of `double`-values in adjacent index positions than to store all real parts *en bloc* before storing all the imaginary parts.

For various array sizes, we compared the speed of array creation, initialization and garbage collection (called 'init' in Figure 1) for the 1-array and the 2-array solution. The lines show how much faster (< 1) or slower (> 1) the 2-array solution is over the 1-array solution ($= 100\%$). Similar for read and write access to the array elements. All measurements have been repeated several times to be able to ignore clock resolution and to achieve a small variance.

It can be noticed that there is no clear advantage of either the 1-array solution of the 2-array solution. The average of all measurements is within $[0.98; 1, 02]$ for both platforms. We got similar results on other platforms and with other JITs. Read and write access is much more stable with HotSpot, however there is some peculiar behavior for initialization. HotSpot probably uses different mechanisms for arrays of different sizes.

We implemented the 2-array solution, since it is neither faster nor slower than the 1-array solution and since use of the 2-array solution eases the implementation of *cj*.

Array creation: Java offers different language elements to create arrays or to create and initialize arrays in one step. Let us first discuss the transformation rule for pure array creation:

$$\begin{aligned}
&eval[new\ complex[e_1] \ldots [e_n]] \mapsto \\
&\quad seq(E_{real} = new\ double[e'_1 = eval[e_1]] \ldots [e'_n = eval[e_n]], \\
&\qquad E_{imag} = new\ double[e'_1] \ldots [e'_n])
\end{aligned}$$

When calculating E_{real}, additional temporary variables e'_i are used to allow the reuse of size expressions in the imaginary part.

Array creation with initialization: Array initialization is done according to the following rule:

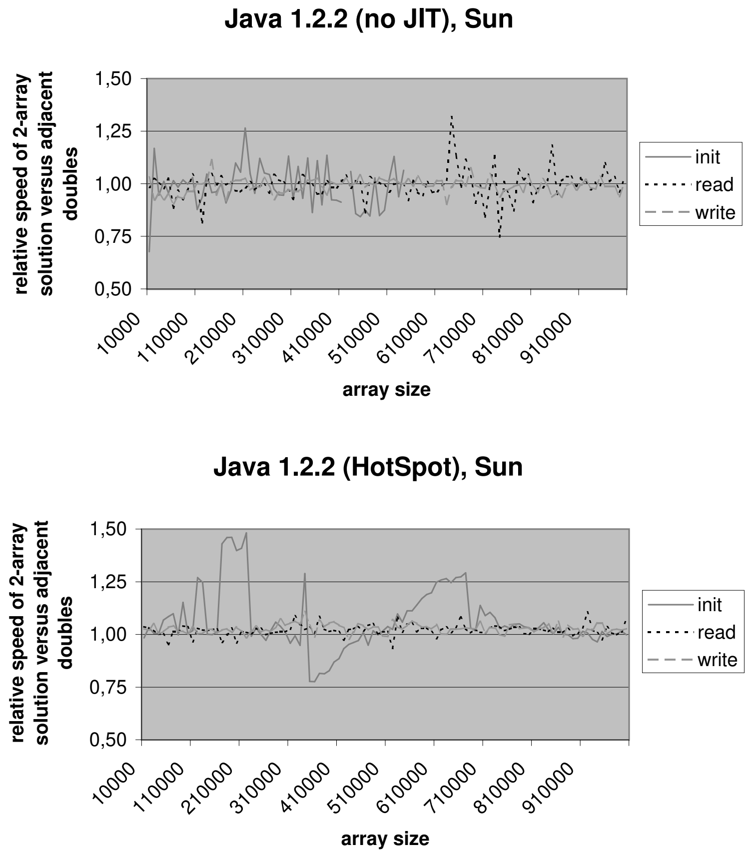

Fig. 1. Performance of 2-array solution compared to 1-array solution in two different JVM/JIT-environments.

$$\begin{aligned}&eval[new\ complex\ []\dots[]\{e_1,\dots,e_n\}] \mapsto\\&\quad seq(E_{real} = new\ double[]\dots[]\{evalR[e_1],\dots,evalR[e_n]\},\\&\qquad E_{imag} = new\ double[]\dots[]\{e_{1imag},\dots,e_{n\,imag}\})\end{aligned}$$

EvalR is able to handle inner array initialization by applying the same rule recursively to array initializations with a smaller number of dimensions.

Array access: The transformation rule for $E \equiv F[e_1]\dots[e_n]$ is:

$$\begin{aligned}eval[F[e_1]\dots[e_n]] \mapsto seq(\ & eval[F],\\& E_{real} = F_{real}[e'_1 = eval[e_1]]\dots[e'_n = eval[e_n]],\\& E_{imag} = F_{imag}[e'_1]\dots[e'_n])\end{aligned}$$

Again, temporary variables e'_i are used to allow the reuse of index expressions in the imaginary part.

Array access used as left hand side of an assignment: Such expressions may be affected by side effects because the evaluation of index expressions could modify the array itself. For example, in `F[foo()]◇=...`, `foo()` could alter some elements of `F`. Hence, the address evaluation for `F` which is needed repeatedly for the `◇=`-operation must be separated from the evaluation of the index expression which must be evaluated exactly once. To achieve this it is in general necessary to store a reference to the array in a temporary variable.

For one-dimensional arrays, the transformation rule is:

$$access(F[e]) \mapsto tmp = eval[F]$$
$$\wedge\ E^{\downarrow}_{real} \equiv tmp_{real}[e' = eval[e]], E^{\downarrow}_{imag} \equiv tmp_{imag}[e']$$

As before, the $\downarrow$-notation emphasizes that the given expressions are to be textually inserted on the right hand side of the assignment. Hence, in the example, the resulting sequence method would execute `foo()` exactly once.

For n-dimensional arrays ($n > 1$), it is sufficient to keep a temporary reference to the (n-1)-dimensional sub-array $F[e_1]...[e_{n-1}]$. The transformation rule for the general case is:

$$access[F[e_1]\ldots[e_n]] \mapsto tmp = eval[F[e_1]\ldots[e_{n-1}]]$$
$$\wedge\ E^{\downarrow}_{real} \equiv tmp_{real}[e'_n = eval[e_n]], E^{\downarrow}_{imag} \equiv tmp_{imag}[e'_n]$$

Again we are using a new temporary variable e'_n to make sure that the index expression is evaluated exactly once.

4.7 Transformation rules for method calls

We discuss `complex` parameters and `complex` return values separately. Moreover, constructors must be treated differently.

Complex return value: There are no means in the JVM instruction set to return two values from a method. An obvious work-around would be to create and return an object (or an array of two `doubles`) every time the method is called. In most cases, this object is only necessary to pass the result out of the method and can be disposed right afterwards. In contrast, *cj* creates a separate array of two `doubles` for *each textual method call.* This array is not defined in the enclosing block but at the beginning of the method that encloses the call. This strategy minimizes the number of temporary objects that have to be created, e.g. for a call inside a loop body. Instead of calling the original method *foo* we are calling a method $\widehat{foo}$ with a modified signature: we pass a reference to this temporary array as an additional argument. This temporary array is created once per textual call of the enclosing method and may be reused several times. So the transformation rule for $E \equiv foo()$ is (similar for methods with arguments):

$$eval[foo()] \mapsto seq(\widehat{foo}(tmp), E_{real} = tmp[0], E_{imag} = tmp[1])$$

Two details are important to ensure the correctness of this transformation for recursive calls and in multithreaded situations: First, the temporary array is local to the enclosing method and second, every textual occurrence of a call of *foo* causes the creation of a different temporary variable.

The return type of $\widehat{foo}$ is not `void`. Instead it returns a dummy value (`null`) so that $\widehat{foo}$ still can be used inside expressions.[5]

[5] Before returning, the elements of the newly added array argument are initialized.

Complex argument: We use the obvious approach by again modifying the signature of the method. Instead of passing one argument of type `complex` we hand over two `double`-values. It is important that not only the argument list of the method but also its name is changed. This is necessary to avoid collisions with existing methods that have the same argument types as the newly created one. The transformation rule can be formalized as (similar for methods expecting several arguments of type `complex`):

$$eval[bar(e)] \mapsto \widehat{bar}(evalR[e], e_{imag})$$

Constructor method: Calls of constructor methods can be treated in the same way – except for the fact that it is not possible to modify the name of a constructor.[6] Since the first statement in the body of a constructor needs to be a call of another constructor the techniques described above would cause the creation of invalid programs if one of its parameters is of type `complex`. The solution is demonstrated by means of the following example.

```
public Foo(complex x, complex z) {
  super((x+x)+z);
}
```

In this example the transformation of `(x+x)+z` would insert statements to declare temporary variables before the call of `super`. As pointed out, this is not legal in Java. To allow the use of `complex` in constructor calls an additional constructor is created. This new constructor expects the same argument list as the first one plus additional arguments, one for each of the necessary temporary variables.

```
private Foo(double x_real, double x_imag, //arguments of the
            double z_real, double z_imag, //first constructor
            double tmp1_real, double tmp1_imag,  //temp.vars.
            double tmp2_real, double tmp2_imag) {
  super(seqREAL(seq(tmp1_real = x_real+x_real, //generated
                    tmp1_imag = x_imag+x_imag),//when
                tmp2_real = tmp1_real+z_real,  //translating
                tmp2_imag = tmp1_imag+z_imag), //super((x+x)+z)
        tmp2_imag);
}
```

The call of `super` has also been modified to accept two `double`-values instead one complex expression.

As a last step, the method call itself is transformed. Instead of calling `super` the newly created constructor is called (by using `this`). All parameters of the surrounding constructor plus an arbitrary value for each temporary variable are passed as arguments.

[6] To avoid collisions that may be caused by altering the argument list, *cj* adds a dummy parameter of a helper type. This is not shown in the example.

```
public Foo(double x_real, double x_imag,
           double z_real, double z_imag) {
  this(x_real, x_imag, z_real, z_imag, 0, 0, 0, 0);
}
```

The semantic analysis of *cj* is able to detect pathological situations where this strategy fails; *cj* issues suitable error messages. For example, an assignment within an argument of a constructor call (in our above example: `super(x = 0.0+x)`) cannot be handled.

5 Benchmarks

5.1 Setup

On a Pentium 100 with 64 MB of RAM and 512 KB of cache we have installed two operating systems: Linux 2.0.36 (Suse 6.0) and Windows NT Version 4 (service pack 4). We have studied several different Java virtual machines for our tests: a pre-release of SUN's JDK 1.2 for Linux, SUN's JDK 1.2.1 for Windows, a JDK from IBM, the JVM that is included in Microsoft's Internet Explorer 5, and the beta release of SUN's new JIT compiler HotSpot.

Our benchmarks fall into two groups: the group of kernel benchmarks measures array access, basic arithmetics on `complex`, and method invocations with `complex` return values. The other group measures small applications: `complex` matrix multiplication, `complex` FFT, and Microstrip calculation. (The Microstrip benchmark [10] computes the value of the potential field $\Phi(x, y)$ in a two-dimensional microstrip structure [4].) There are at least two versions of each program – one uses our basic type `complex` and the other uses a class to represent complex numbers.

5.2 Results

On average over all benchmarks, the programs using the basic type `complex` outperform the class-based versions *by a factor of 2 up to 21*, depending on the JVM used. We achieve the best factor with SUN's JDK 1.2 for Windows, which is the slowest JVM in our study. The smaller improvement factors are achieved with better JVMs (HotSpot and Internet Explorer) that incorporate certain optimization techniques, e.g., removal of redundant boundary checks, fast creation and handling of objects, and aggressive inlining of method bodies.

5.3 Results in detail

Figure 2 gives an overview over all benchmarks, labeled (a) to (f). In each of these six sub-figures there are five groups of bars, each group represents a different JVM. The most important item within a group is the black bar. This bar shows the relative execution time of the class-based version. The factor by which this

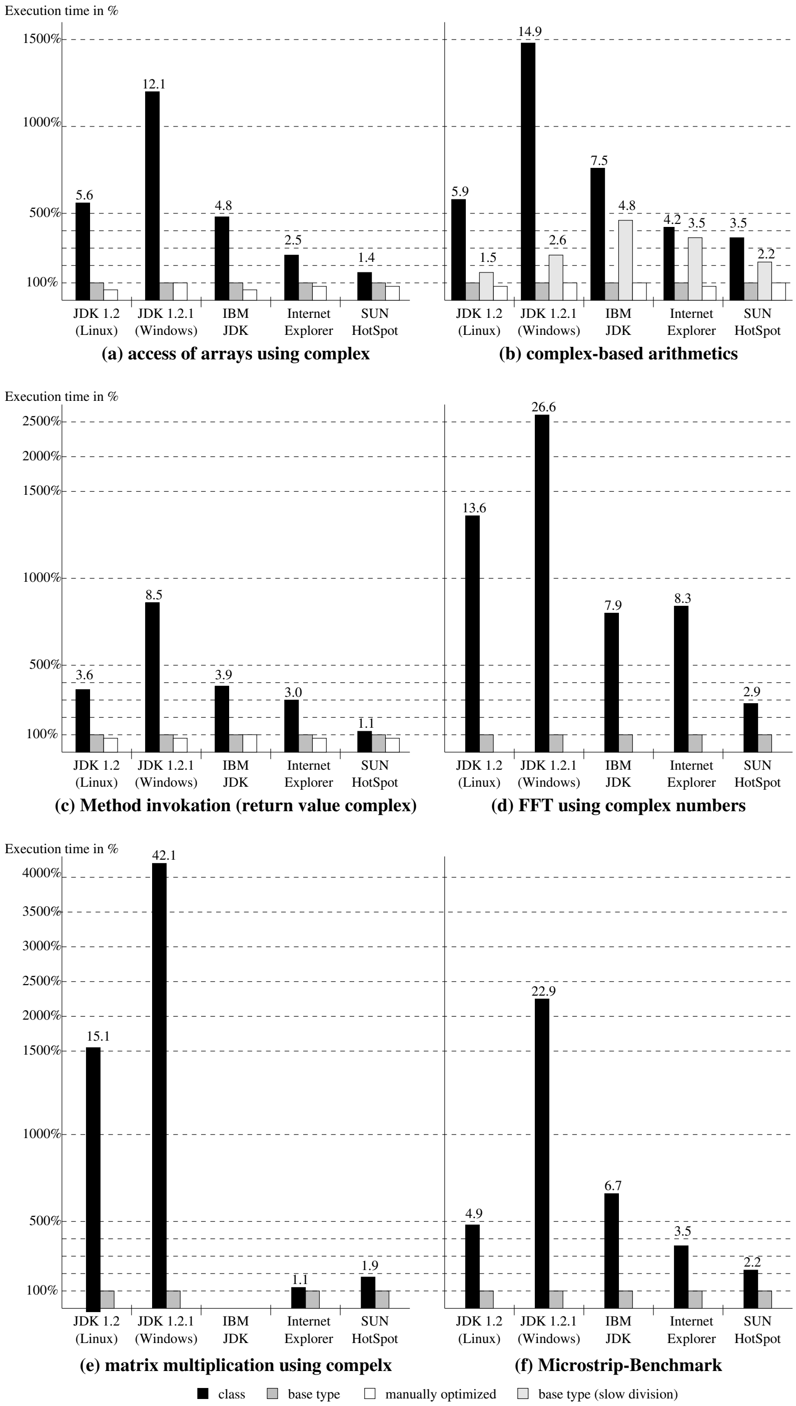

Fig. 2. Results of the benchmark programs

version is slower than the basic type version (grey bar) is printed on top of the black bar. Some groups have more than two bars: here we did an additional transformation by hand, substituting each `complex` by two variables of type `double`. Those manually optimized programs (white bar) are just slightly faster than code generated by *cj*.

In sub-figures (a) to (c) the improvement is smaller than in the other figures. It is also apparent that better implementations of the JVM (Internet Explorer and HotSpot) are quite good in eliminating the overhead of object creation within the class-based solutions. But *cj* still performs better by 10% to 40%.

Benchmarks (a) and (c) focus on array access and method invocation. In contrast, programs (b) and (d-f) are predominantly calculating arithmetic expressions, where (d) and (e) are also showing some amount of array accesses. For arithmetics the techniques applied by *cj* (inlining of all method invocations and reducing the number of temporary variables) perform noticeably better than the class-based solution. Even on the better JVMs *cj* is 3 times faster. On slow JVMs *cj* achieves a factor of 8 or more. The main reason is that *cj* does a better inlining and can avoid temporary objects almost completely.

6 Conclusion

Complex numbers can be integrated seamlessly and efficiently into Java. Because of Java's strict evaluation order it is by far not enough to simply double the operations for their real and imaginary parts. Sequence methods enable a formalization of the necessary program transformations in a local context. Our technique for dealing with complex return values is efficient because it avoids the creation of many temporary objects. In comparison with their class-based counterparts, the benchmark programs that use the new primitive type perform better by a factor of 2 up to 21, on average, depending on the JVM used.

Acknowledgements

The Java Grande Forum and Siamak Hassanzadeh from Sun Microsystems supported us in understanding the necessity of complex numbers in Java for scientific computing. Thanks to Martin Odersky for providing *gj*. Bernhard Haumacher and Lutz Prechelt gave valuable advice for improving the presentation.

References

1. Gilard Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA'98, 13th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1998. http://www.cis.unisa.edu.au/~pizza/gj/.
2. C9x proposal. ftp://ftp.dmk.com/DMK/sc22wg14/c9x/complex/ and http://anubis.dkuug.dk/jtc1/sc22/wg14/.

3. Joseph D. Darcy and William Kahan. Borneo language. http://www.cs.berkeley.edu/~darcy/Borneo.
4. Terence C. Edwards. *Foundations for Microstrip Circuit Designs*. John Wiley & Sons, 1992.
5. James Gosling. The evolution of numerical computing in Java. http://java.sun.com/people/jag/FP.html.
6. IBM. Numerical intensive Java. http://www.alphaWorks.ibm.com/tech/ninja/.
7. Java Grande Forum. http://www.javagrande.org.
8. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
9. William Kahan and J. W. Thomas. Augmenting a programming language with complex arithmetics. Technical Report No. 91/667, University of California at Berkeley, Department of Computer Science, December 1991.
10. José Moreira and Sam P. Midkiff. Fortran 90 in CSE: A case study. *IEEE Computational Science & Engineering*, 5(2):39–49, April–June 1998.
11. Stephen M. Omohundro and David Stoutamire. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, 1996.
12. Guy Steele. Growing a language. In *Proc. of OOPSLA'98, 13th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1998. key note.
13. George K. Thiruvathukal, Fabian Breg, Ronald Boisvert, Joseph Darcy, Geoffrey C. Fox, Dennis Gannon, Siamak Hassanzadeh, Jose Moreira, Michael Philippsen, Roldan Pozo, and Marc Snir (editors). Java Grande Forum Report: Making Java work for high-end computing. In *Supercomputing'98: International Conference on High Performance Computing and Communications*, Orlando, Florida, November 7–13, 1998. panel handout.
14. Visual Numerics. Java grande complex reference. http://www.vni.com/corner/garage/grande/index.html, 1999.
15. Peng Wu, Sam Midkiff, José Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*, pages 109–118, San Francisco, 1999.