Jumbo: Run-time Code Generation for Java, and Its Applications

Sam Kamin[*], Lars Clausen, Ava Jarvis
Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{s-kamin,lrclause,ajar}@uiuc.edu

August 2, 2002

Abstract

Run-time code generation is a well-known technique for improving the efficiency of programs by exploiting dynamic information. Its widespread use has been hampered primarily by the difficulty of constructing run-time code generators. We describe Jumbo, a tool for creating run-time code generators for Java. Jumbo is a compiler for a two-level version of Java, in which quoted code fragments can appear within other Java programs; the quoted fragments can be combined at run-time and then executed. We illustrate Jumbo with several examples that show significant speed-ups over similar code written in plain Java. We argue further that Jumbo can be regarded as a generalized software component system.

1 Introduction

Jumbo is a dynamic compiler for Java capable of pulling together fragments of Java code — down to the level of single expressions or statements — into a single program and compiling the combined program at run-time. Furthermore, it does not accomplish this by simply building the program as text and invoking the Java compiler from the running program (as could be done easily with existing technology). Rather, the compiler is structured in such a way that fragments can be separately compiled to an intermediate form from which the full program can be assembled without reinvoking the compiler. In other words, Jumbo builds run-time program generators.

This structure allows for great generality in the program generators Jumbo can build. Furthermore, since the program generators are binaries (that is, Java Virtual Machine code) and produce binaries, with no separate invocation of the compiler, it passes the "deployability" test for components. From this point of view, Jumbo can be seen as a component-constructing tool of great generality — much greater than traditional component systems that operate at the level of methods or classes.

Indeed, the Jumbo system combines the advantages of a traditional run-time code-generation (RTCG) system — improved execution with little or no programmer input — and a macro system — improved programmer productivity without a change of language. After introducing the system, we will present examples to show how powerful this combination can be.

[*]Partial support for the first and second authors was received from NSF under grant CCR-9619644.

The next section gives an overview of the Jumbo system. Sections 3 through 5 give examples of increasing complexity. In section 3, we take the client's point of view, showing examples where Jumbo components can be used without using the Jumbo compiler. More power is available to the client who also uses Jumbo; examples of this are given in section 4 and 5. Section 6 presents an example illustrative of Jumbo's use as a component system, in which a client and a server wish not only to communicate code, but also to specialize it dynamically. References to related work and conclusions are presented in the final sections.

2 The Jumbo System

The simplest way to understand Jumbo is to think of it as a "compiler API." Jumbo consists of about a dozen classes, of which the most important is `Code`. The API contains the following static operations, among others:

```
static Code binop (int, Code, Code)
static Code ifThenElse (Code, Code, Code)
static Code returnVal (Code)
```

That is, the Jumbo compiler contains operations corresponding to abstract syntax operations. The operations act upon and produce values of type `Code`; when using Jumbo, it is not misleading to think of `Code` as containing abstract syntax trees, or even strings. However, `Code` is not either of these things, but is instead a partially compiled value. Programmers will rarely, if ever, use these AST operations directly. Rather, a special quotation syntax is provided to produce calls to these operations. We will see examples later in this section.

In addition, the API contains these two instance methods of the `Code` class:

```
void generate ()
Object create (String classname)
```

The `generate` method converts a `Code` value into JVM code and writes the corresponding class files; that `Code` value must correspond to a class or list of classes. `create` first calls `generate` and then loads the named class and returns a new object of that class. (There is at present no mechanism to pass arguments to the constructor for that class.)

The basic idea behind Jumbo is *compositional compilation* [4]. The crucial point is that the abstract syntax operations of the Jumbo API *are* the compiler. Unlike an ordinary compiler in which the syntax tree is a passive data structure upon which the compiler operates, the Jumbo operators are genuine functions that perform compilation. This is called a *compositional compiler* because the compilation of each construct is a function only of the compilation of its sub-constructs. This is a very different structure from conventional compilers. The advantage is that any particular piece of syntax can be easily abstracted from and filled in at a later time.

The idea of compositional compilation is widely applicable, to different languages and target architectures. In [4], we used a variant of Java and targetted Sparc machine language. However, some languages and target architectures are much more difficult to compile compositionally, and trade-offs in code quality must be made. The structure of Java and its implementation by an abstract machine are particularly well suited to our approach; Jumbo produces virtually the same code as the `javac` compiler. There is just one feature of Java — inner classes — which is especially difficult to handle, and these are not yet implemented in Jumbo.

2.1 Using code generators

In discussing Jumbo programs, we often refer to the `server` as the supplier of the code generator, and the *client* as its user. Following this metaphor, we also sometimes refer to Jumbo programs as *components*.

A typical Jumbo client is the following code, which uses a vector dot-product generator supplied by a server. `Dot` is an interface containing the method `double dot(double[])`.

```
double[] vec1 = getInputVector();
Code dotprodcode = codegenDot(vec1);
Dot dotprod = (Dot)dotprodcode.create("DotProd");

while (true) {
     double[] vec2 = getInputVector();
     double p = dotprod.dot(vec2);
     outputResult(p);
}
```

`codegenDot` is the Jumbo method supplied by the server; we will show its definition shortly. Its function is to produce a class named `DotProd` that implements the `Dot` interface. As with any other library code, the client knows only its type:

```
Code codegenDot (double[])
```

and, of course, its purpose. It produces a `Code` value to which the Jumbo method `create` can be applied to produce the `DotProd` class and return an object of that class. Since that class does not exist at the time the client code is compiled, we must also give the `Dot` interface.

In summary, the client uses the method `codegenDot` to produce a method that is specialized to multiply other vectors by the fixed vector `vec1`. Fixing one of the vectors in a dot product calculation offers the potential to speed up the calculation by using this "dynamic" information. That information includes the length of the vector — which allows the dot product loop to be unrolled — and its values,which may be exploited to gain efficiency — for example, by omitting multiplications by zero or one.

The pure Java alternative to the above would use a non-generating dot product routine, say `dotprod`:

```
double[] vec1 = getInputVector();
while (true) {
     double[] vec2 = getInputVector();
     double p = dotprod(vec1, vec2);
     outputResult(p);
}
```

where `dotprod` is the library method

```
double dotprod (double[] V1, double[] V2) {
     double sum = 0.0;
     for (int i=0; i<V1.length; i++)
          sum += V1[i]*V2[i];
     return sum;
}
```

Vector length	Iterations	Properties	Jumbo	Java	Jumbo runtime
100	100	random	0.25	0.0	0.0
100	1000	random	0.33	0.01	0.0
100	10000	random	0.34	0.14	0.01
100	100000	random	0.36	0.34	0.04
100	1000000	random	0.53	1.18	0.29
100	10000000	random	3.06	9.59	2.82
100	100	50% 0's	0.31	0.0	0.0
100	1000	50% 0's	0.34	0.01	0.0
100	10000	50% 0's	0.35	0.14	0.0
100	100000	50% 0's	0.36	0.26	0.02
100	1000000	50% 0's	0.50	0.97	0.15
100	10000000	50% 0's	1.62	8.25	1.3

Table 1: Effect of dynamically compiling dot product routine

The advantage of the dynamically compiled version is that, in some cases, it may be much faster; the disadvantage is that, in some cases, due the cost of run-time compilation, it may be much slower. Like any other programming tool, Jumbo must be used judiciously. However, when applicable, it gives speed-ups that are very difficult to obtain by conventional means.

To give an idea of the kinds of speed-ups that are possible, Table 1 shows the timings for various runs of the two programs above. As input data, we used randomly-generated 100-element vectors of two kinds: no bias, and bias towards generating zeroes. The dot product computation was repeated a varying number of times ("iterations" in Table 1) with the same two vectors; in particular, for the Jumbo version, the code generation process occurs just once in each run. The first "Jumbo" column represents total code generation and run time, and the "Java" column is for the conventional two-argument dot product routine. The last column is the runtime for the Java code, discounting code generation time.

All timings given in this paper were produced on an unloaded 1.47GHz Athlon PC with 1GB of main memory, running Linux kernel 2.4.18. The `java -version` command reports

```
java version "1.3.1"
Java(TM) 2 Runtime Environment, Standard Edition (build Blackdown-1.3.1-FCS)
Java HotSpot(TM) Client VM (build Blackdown-1.3.1-FCS, mixed mode)
```

Times are in seconds, obtained with a microsecond timer routine.

The results of this run show the potential for significant speed-ups when the cost of dynamic code generation is amortized over a sufficient number of uses. Dynamic code generation for these examples is taking about 0.3 seconds. For small runs, this time dominates, but by the time the Java computation exceeds one second, the Jumbo version is already twice as fast. For the unbiased vectors — where the only benefits of Jumbo come from loop unrolling — the Jumbo version is eventually over three times as fast; for the biased vector, the improvement factor is about five.

We are currently investigating methods of decreasing run-time compilation time, and we expect great improvements there. (Also, about 10% of the dynamic compilation cost is in the verification phase, which might be safely eliminated in some circumstances.) However, Jumbo can be used statically as well as dynamically, and in some cases (we will see examples later in this paper) this is quite appropriate. In those case, the run-time seen by the user would be that shown in the last column.

2.2 Writing code generators

Someone has to write the code generators, of course. Jumbo is easy enough to use that many programmers could do so. This requires the use of the Jumbo compiler (unlike the example shown above, where the client need only have the Jumbo API and could use any Java compiler). The Jumbo compiler processes a "two-level" version of Java to produce code generators. This compiler is based on a quote/anti-quote syntax similar to that of MetaML [7].

In this notation, a fragment of Java code within brackets `$<` and `>$` represents a value of type `Code`, as in

```
Code C = $< class CodeExample { ... } >$;
InterfaceType obj = (InterfaceType)C.create("CodeExample");
... use obj ...
```

The quotation syntax works much like ordinary quotation, except that the result is a value of type `Code` rather than `String`. (Java type-checking requires that we use an interface `InterfaceType`, since the class `CodeExample` does not exist at the time this code fragment is compiled.)

The Jumbo compiler is used to compile the code within quotes. The only restriction on that code is that it cannot use inner classes, because these are not yet implemented in Jumbo. In our examples, we sometimes use inner classes, but never inside quotes.

Within a quoted Java fragment, values of type `Code` can be "spliced," using the anti-quotation syntax

‘*syntax-category*(*Java code fragment*)

The contained Java fragment must be an expression of type `Code`. This entire anti-quoted section is replaced, for parsing purposes, by a generic fragment of syntactic type *syntax-category*; the only reason we require the syntax category to be present is to allow for parsing of the surrounding code.

For example, we can write

```
Code addxy = $< x+y >$;
Code assignz = $< z = x*5; >$;
...
Code example = $< class CodeExample {
                     ... ‘Expr(addxy) ...
                     ... ‘Stmt(assignz) ...
                  } >$;
InterfaceType obj = (InterfaceType)C.create("CodeExample");
... use obj ...
```

The anti-quoted expressions `‘Expr(addxy)` and `‘Stmt(assignz)` are "holes" are filled by the code in `addxy` and `assignz`. As mentioned earlier, the casual user can think of this as splicing strings into the string `example`; however, they are not strings but rather values of type `Code`.[1]

[1] We insist on this perhaps pedantic-seeming distinction, because it is in fact crucial. If these were strings, the `create` function would do nothing but invoke the regular Java compiler, and Jumbo would be simply a Java compiler with an anti-quotation notation. The ability to partially compile code results from this distinction.

The anti-quoted parts can be any Java expression of type `Code`. If the method `f` has signature `Code f(int)`, we can write

```
$< ... `Expr( f(3) ) ... >$
```

At run-time, `f(3)` would be evaluated and the resulting `Code` value spliced into the surrounding `Code` value.

Here is the complete list of legal syntactic categories for anti-quotations:

Category	**Expression value expected (Type)**
`Expr`	Expressions (Code)
`Stmt`	Statements (Code)
`Name`	Identifiers (String) (If the category is omitted, `Name` is used as the default.)
`Type`	Types (Type)
`Case`	List of case branches (MonoList containing Code values)
`Method`	Method declaration (Code)
`Field`	Field declaration (Code)
`Body`	List of class members (MonoList containing Code values)
`Char`	Character constant (char)
`Int`	Integer constant (int)
`Float`	Float constant (float)
`Long`	Long constant (long)
`Double`	Double constant (double)
`Bool`	Boolean constant (boolean)
`String`	String constant (String)

We have given in each case the type of the expression that must be given as an argument to the syntax category. In the examples above, we used syntax categories `Expr` and `Stmt` and their arguments were of type `Code`. As the table shows, there are some syntax categories that require different types of arguments:

- `Name` is a category used when the only syntactically valid item is an identifier. This would be used, for example, to abstract the name of a class. (As noted, this is the default category; when we write `` `ident `` with no syntax category and no parentheses, `ident` must be a variable of type `String`.)
- `Type` is used wherever a type is required. For ``$< `Type(t) x, y; >$`` to be a valid declaration, `t` must be a Java expression of type `Type`. The simplest way to create a value of type `Type` is to use the method `static Type parseType (String typename)`. For example, `parseType("int")` returns the `Type` value representing the `int` type. `Type` also defines a constant `int_type` containing that same type, and the other analogous constants.
- The `Case` and `Body` categories require `MonoList`s of `Code` values. `MonoList` is a collection class interface included in the Jumbo API (along with several implementations). It differs from Java's `List` class primarily in that the `add` operation has type `MonoList add(Object)` instead of `boolean add(Object)`; the more "functional" style `add` operator turns out to

be more convenient for our purposes. For the same reason, a quoted list of switch cases (`$<case 0: ... case 1: ...>$`) returns a `MonoList` of `Code` values; all other quoted fragments return `Code`.

- The last few categories (`Char`, ..., `String`) must have arguments of the corresponding types. (That is, expressions of those exact types, not `Code`.) These allow Java values to be "reified," turned into syntax that can be included in generated programs.

There is only one `Code` type, but it covers a variety of syntactic types. Intuitively, `$< x+y >$` should denote a "`Code` value of type `Expr`" and `$< x = y; >$` a "`Code` value of type `Stmt`." Then, when we write, for instance, `$<... ‘Expr(e) ...>$`, we should insist that `e` evaluate to a `Code` value of type `Expr`. Indeed, we might go further and insist that `e` evaluate to a `Code` value of type `Expr` having, say, data type `double`. However, we make no such distinctions, instead having only a single type, `Code`. In discourse, we often refer to a `Code` value "corresponding to" an expression or statement, etc. We do no type-checking of generated code when we compile the program generator (as is done in MetaML). Such type-checking is done when `generate` is called.

It is legal to apply `generate` or `create` only to a `Code` value corresponding to a class definition or list of same.

We have now completed our presentation of Jumbo and are able to present the definition of `codegenDot`. We will start with a simpler version than the one we used for the timings above. It just unrolls the loop:

```
public static Code makeSumCode(double[] V1, Code V2) {
    Code sumcode = $<0.0>$;
    for (int i = 0; i < V1.length; i++)
            sumcode = $<‘Expr(sumcode)+(‘Expr(V2))[‘Int(i)] >$;
    return sumcode;
}

public static Dot codegenDot(double[] V1) {
    Code c = $<public class DotProd implements Dot {
        public double dot(double[] V2) {
            double product =
              ‘Expr(makeSumCode(V1, $<V2>$));
            return product;
        }
    }>$;
    Dot dotV1 = (Dot)c.create("DotProd");
    return dotV1;
}
```

For example, if `v1` were the array `{5.2, 0.0, 2.4, 1.0}`, the generated class would contain a method equivalent to

```
public double dot(double[] V2) {
     double product =
          0.0 + 5.2*vec2[0] + 0.0*vec2[1] + 2.4*vec2[2] + 1.0*vec2[3]
     return product;
}
```

As shown in Table 1, elimination of the loop overhead can produce significant benefits on its own. Our second version takes advantage of the values in `v1`, when those include 0's and 1's. It is identical to the above, except for the definition of `makeSumCode`:

```
public static Code makeSumCode(double[] V1, String V2) {
     Code sumcode = $<0.0>$;
     for (int i = 0; i < V1.length; i++) {
          if (V1[i] == 1.0)
               sumcode = $<‘Expr(sumcode)+‘V2[‘Int(i)] >$;
          else if (V1[i] != 0.0)
               sumcode = $<‘Expr(sumcode)+‘Double(V1[i])*‘V2[‘Int(i)] >$;
     }
     return sumcode;
}
```

Then the generated `dot` method would contain

```
double product = 0.0 + 5.2*vec2[0] + 2.4*vec2[2] + vec2[3]
```

This is the version we used above. If the vector `v1` were expected to have other characteristics that might make the dot product calculation more efficient — such as frequent repeated values or some known symmetries — then these could be incorporated as well.

We have explained essentially all of the features of Jumbo: a "compilation API" with a parser that adds a quotation/anti-quotation feature. These conceptually simple facilities are remarkably powerful, as we will illustrate in the remainder of this paper. Further examples can be found in [3, 4], which describe a predecessor of Jumbo.

3 First-order program generators

It is possible to use methods created with Jumbo with little knowledge of Jumbo itself.

Jumbo methods whose arguments include no `Code` values are called "first-order" methods. For clients, these are the easiest methods to use. Furthermore, they do not require that the client have the Jumbo compiler, but only the Jumbo API. The dot product example above is first-order, since the client supplies only an ordinary Java value — an array of doubles — and never creates *Code* values directly. The client code could be compiled by any Java compiler.

In this section, we give two examples to show the performance benefit obtained using RTCG. We will show the implementations of these components, which are quite straightforward, but keep in mind that the client does not need to know how they are implemented. We will pair each Jumbo method with the analogous non-Jumbo method — as we did for the dot product calculation — so we can compare their running times.

3.1 Graphical transformations

Our first example is not essentially different from the dot product, but addresses a more common and realistic problem. Many graphics systems represent standard linear transformations on images by *transformation matrices* [2]. A 2-D transformation matrix is a 3×3 matrix M. When a transformed image I is to be displayed, each point (x, y) in the image is mapped using M.

Specifically, (x, y) is transformed to (x', y'), where $(x', y', z) = M \cdot (x, y, 1)$; that is, the point is extended with an additional one and multiplied by M, and the third element of the result is dropped.

Run-time code generation can significantly speed up calculations involving transformation matrices. The matrix multiplication is often done thousands of times using the same transformation matrix, suggesting that the cost of generating specialized code for a given matrix would be amortized over many uses. Furthermore, sometimes only simple transformations are performed, in which case the transformation matrices tend to contain zeros and ones; these allow for simplifications in the matrix multiplication, just as they did for the dot product calculation. For example, a common transformation is the identity transformation, represented by the identity matrix.

We write a Jumbo method of type

```
TransPoint createTrans (double[][] transMat)
```

where `TransPoint` is the interface[2]

```
interface TransPoint {
    void transPoint (Point p);
}
```

When applied to a transformation matrix, `createTrans` produces a class that implements `TransPoint`, and returns an object of that class. Typical client code would be

```
TransPoint tp = createTrans(M);
...
for (each location p = (x,y) in image) {
     ...
     tp.transPoint(p);
     ...
}
```

(In practice a single transformation matrix is often used for many images, so that `tp` might be created just once and used numerous time.)

We will present the definition of `createTrans` later in this section.

Using conventional Java methods, the client code above would be

```
for (each location p = (x,y) in image) {
     ...
     transPoint(M, p);
     ...
}
```

As we did for the dot product, we will give the execution-time comparisons for the two versions on several examples. For the Jumbo version, we give the timings both including and excluding the Jumbo compilation and load time (the call to `createTrans`).

As usual, the overall time savings from RTCG depend on the size of the images and the number of images that are transformed by a given matrix — that is, the opportunity to amortize the cost

[2] `Point` is a Java library class representing pairs of integers, with selectors `x` and `y`.

Points (millions)	Transformation	Jumbo	Java	Jumbo runtime
1	identity	0.36	0.32	0.02
10	identity	0.48	0.87	0.16
100	identity	1.81	7.14	1.51
1000	identity	15.44	69.16	15.11
1	rotate	0.39	0.26	0.07
10	rotate	0.92	0.89	0.60
100	rotate	6.38	7.12	6.06
1000	rotate	59.81	69.16	59.49
1	scale by 2	0.38	0.26	0.06
10	scale by 2	0.85	0.87	0.53
100	scale by 2	5.62	7.01	5.30
1000	scale by 2	53.03	67.80	52.71

Table 2: Effect of dynamically compiled transformation matrix routine.

of RTCG. In this case, the contents of the transformation matrix are also relevant. In Table 2, we gives times for transforming various numbers of points, from 1,000,000 up to 1,000,000,000. The matrices are:

Transformation	Matrix
identity	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
scale 2	$\begin{pmatrix} \cos \pi/2 & \sin \pi/2 & 0 \\ -\sin \pi/2 & \cos \pi/2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
rotate π	$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

For this example, run-time compilation takes about 0.36 seconds; because of this cost, the Jumbo version is slower than the Java version for one million points; the cross-over point is at about three million, although the advantage of the Jumbo version depends upon the transformation matrix. When transforming one billion points, Jumbo is as much as four times faster (for the identity transformation) and as little as 15% faster (for rotation).

We now present the definition of `createTrans`, which is similar to `codegenDot`. One difference is that the client does not call `create` directly, but leaves that to `createTrans`. This is simply a matter of style.

```
static TransPoint createTrans (double[][] M) {
```

```
   Code transCode =
       $< import java.awt.Point;
          public class TransPointByM implements TransPoint {
            public void transPoint (Point p) {
               p.x = (int)`Expr(transCoord(M, 0, $<p>$));
               p.y = (int)`Expr(transCoord(M, 1, $<p>$));
            }
          } >$;
   return (TransPoint)transCode.create("TransPointByM");
}

static Code transCoord (double[][] M, int coord, Code p) {
   Code c = null;

   if (M[0][coord] == 0.0)
     c = $<0>$;
   else if (M[0][coord] == 1.0)
       c = $<(`Expr(p)).x>$;
   else
       c = $<(`Expr(p)).x*`Double(M[0][coord])>$;

   if (M[1][coord] == 1.0)
       c = $<`Expr(c)+(`Expr(p)).y>$;
   else if (M[1][coord] != 0.0)
       c = $<`Expr(c)+(`Expr(p)).y*`Double(M[0][coord])>$;

   if (M[2][coord] == 1.0) // M[2] == 0 or 1
       c = $<`Expr(c)+ `Double(M[2][coord])>$;

   return c;
}
```

(Note: We wrote `(‘Expr(p)).x`, instead of `‘Expr(p).x`, for purely syntactic reasons. The syntax of Java calls for the name preceding the dot to be a "primary," and not an arbitrary expression. Enclosing an expression in parentheses makes it a primary.)

3.2 Generic data types

In Java, most collection classes are generic in a trivial way: they contain only `Objects`. One disadvantage is that primitive values must be boxed, by creating objects with type names like `Integer` and `Character`, before being placed into a collection. Boxing has a significant run-time cost due to the extra object creation. Yet, these generic collection classes can be easily transformed into non-generic classes: just replace the type `Object` by the desired type of element, and recompile. This is, in essence, the effect of using templates in C++.

We can get the same benefit in Jumbo by supplying the typename as a parameter to a program generator. For example, the `Vector` class in the Java library is a heavily used container class, with operations like `add` and `iterator`. It contains `Objects`, so primitive values must be boxed if they are to be placed in vectors. We created a non-generic collection class generator using Jumbo, which generates classes similar to Java's `Vector` class. The generator provides three operations:

```
static Type vector (String type)
static Type iterator (String type)
static Code newVector (String type)
```

The `vector` and `iterator` methods are used to create the names of the vector type corresponding to a particular non-generic instance, and the type of its iterator. The argument gives the name of the component type. `newVector` creates an expression whose value is an object of the desired collection type. The non-generic class is created implicitly whenever one of these operations is called. (This mimics C++ template usage. We could as easily have allowed the user to give specific names for these types and then use those names.)

The implementation consists largely of a Java class placed within quotations and abstracted on the names of the element type and the collection type:

```
Code C = $<public class ‘vectorname {
             ‘Type(elttype)[] elements;
             int numelements;

             public ‘vectorname() {
                 elements = new ‘Type(elttype)[10];
             }

             public void add(‘Type(elttype) o) {
                  ...
             }
             .....
          >$;
```

The only additional code, which we haven't shown, is the code to insure that the desired collection class is generated exactly once.

The timings shown in Table 3 were obtained by creating vectors of various lengths (`numints`) and then summing them:

```
‘Type(vector("int")) v = ‘Expr(newVector("int"));
for (int i = 0; i < numints; i++) {
    v.add(i);
}
int sum1 = 0;
for (‘Type(iterator("int")) i = v.iterator(); i.hasNext(); ) {
    sum1 += i.next();
}
```

For the Java column, we mimicked Java by creating a generic class `vector("Object")` and using it as one would an ordinary generic collection in Java:

```
‘Type(vector("Object")) v = ‘Expr(newVector("Object"));
for (int i = 0; i < numints; i++) {
    v2.add(new Integer(i));
}
int sum2 = 0;
```

numints	vector("int")	vector("Object")
100	0.001	0.001
1000	0.002	0.002
10000	0.007	0.022
100000	0.023	0.132
1000000	0.186	0.976

Table 3: Effect of dynamically compiled non-generic collection class

```
for (‘Type(iterator("Object")) i = v2.iterator(); i.hasNext(); ) {
    sum2 += (Integer)(i.next()).intValue();
}
```

Table 3 shows only the execution times, since we assume the various vector classes would be created and compiled first, using `generate`. After that, they would be ordinary class files, and would be used without further compilation. It will come as no surprise to Java programmers that a vector of `ints` is much more efficient than a vector of `Integers`.

3.3 Discussion

The first-order Jumbo methods are easy to use and require no special knowledge of Jumbo. It is easy to imagine libraries of such code, perhaps including both code-generating and conventional versions of each routine and leaving it to the programmer to decide which to use. These libraries would be usable, of course, only if the Jumbo API were available, but they would not require that the programmer use the Jumbo compiler. On the other hand, the first-order components are also relatively easy to write; with the Jumbo compiler available, it is not too much of a leap from *using* Jumbo methods to *creating* them. The "second-order" Jumbo methods require clients to use the Jumbo compiler, but only in simple ways. We illustrate this next.

4 Second-order program generators

Much more can be done with Jumbo if clients have the ability to pass code fragments into Jumbo methods. When a method has arguments and result of type `Code`, we call it a *second-order* Jumbo method.

4.1 Lexer generator

A lexer generator, similar to `lex`, can be built with Jumbo. In contrast to `lex`, the Jumbo version does not require a compiler, and it can generate lexers dynamically.

Specifically, suppose we define a method `parseRE` of type

```
RegExp parseRE (String re, Code action, RegExp tail)
```

so that the expression

```
parseRE(R1, $<A1>$,
parseRE(R2, $<A2>$, ...,
parseRE(Rn, $<An>$, null) ... ))
```

would correspond to the `lex` specification consisting of regular expressions $R_1, \ldots, R_n$, with corresponding associated actions $A_1, \ldots, A_n$. `parseRE` would produce a `RegExp` value, which could then be translated to a DFA by the usual method. Let us further suppose that there is a method

```
makeScanner(Code decls, RegExp re, Code report)
```

Here, for example, is a specification of the word count (`wc`) utility:

```
makeScanner(
   $< int nlcount=0; int wordcount=0; int charcount=0; >$,
   REParse.parse("\n", $< nlcount++; charcount++; >$,
   REParse.parse("[^ \t\n]+", $< wordcount++; charcount += theToken.length(); >$,
   REParse.parse(".", $< charcount++; >$, null)));
   $< System.out.println(nlcount + " " + wordcount + " " + charcount); >$);
```

The actions have access to a variable `theToken` containing the token just scanned, as a `String`.

In fact, such a component could also be written in conventional Java, using function objects for the actions. In Java, there is little to be gained in terms of efficiency, since the most efficient implementation — using goto's — is not available in Java. However, the Jumbo version does have a slight edge in performance because a function call is not needed for each action, and it has a slight edge in compactness of specification, since a class does not need to be defined for each action.

4.2 Loop unrolling

A simple example of a second-order generator — which at the same time illustrates the limitations of second-order generators — is a loop-unroller:

```
Code unroll (int n, Code C) {
    if (n==1)
        return C;
    else
        return $< `Stmt(C) `Stmt(unroll(C, n-1)) >$;
}
```

If the user is willing to supply a Code value `C` and an integer `n`, this method produces code that consists of `n` copies of `C`.

```
Code print10 = unroll(10, $< System.out.println(0); >$);
```

When compiled, `print10` produces a class `Print10` with a method `print10` which prints zero ten times. In comparison to a loop or recursive function, this code will be more efficient because it will avoid the control overhead.

`unroll` is not restricted to unfolding a constant number of times. The call

```
unroll(i, $< System.out.println(0); >$)
```

will produce a routine that prints `i` zeroes. However, the body of the loop is restricted in that it cannot depend upon an iteration variable. It must just be a single unvarying piece of code that will be executed repeatedly. Clearly, this will not get us very far.

5 Higher-order program generators

The full power of Jumbo can be employed only when the client has the Jumbo compiler and can write his own code-producing methods. This provides for communication between the component and client.

5.1 Loop unrolling revisited

Intuitively, loop unrolling is second order. The client supplies the body of the loop and the unrolling method repeats it as necessary. But, as we have seen, a strictly second order loop unroller is of limited utility.

The solution is for the body of the loop, supplied by the client, to itself be a `Code`-producing function. We will give two versions. The first approach is to make the body a function from the iteration count to a statement. Define the interface

```
interface LoopIteration {
    Code iteration (Code i);
}
```

A simple form of loop unrolling is then easily written:

```
public static Code unroll (int n, LoopIteration F) {
   Code C = $< ; >$ ;
   for (int i=0; i<n; i++)
      C = $< `Stmt(C) `Stmt(F.iteration($<`Int(i)>$)) >$;
   return C;
}
```

The client provides a function object of type `LoopIteration`:[3]

```
`Stmt(unroll(5,
      new LoopIteration () {
           public Code iteration (Code x) {
               return
                  $< System.out.println(`Expr(x)); >$;
           }
      }))
```

The generated code contains the statement sequence

[3]As noted earlier, inner classes cannot be used *inside* quotations.

```
System.out.println(0);
System.out.println(1);
System.out.println(2);
System.out.println(3);
System.out.println(4);
```

(A more convenient way to specify anonymous functions would improve the appearance of this example.)

We prefer a more general form, for reasons that will become clear shortly. This loop unroller takes, as arguments, not only the number of iterations and the loop body, but also the initial value of the iteration variable, a loop increment, and the iteration variable itself. Of these, only the iteration count and the increment are constants; the initial value and the loop variable are expressions, and the loop body is, of course, a `LoopIteration` value. (N.B. The loop count, `n`, is the total number of iterations to be taken, not the final value; it is independent of the initial value or the increment.)

This more general form will allow us to write several loop unrollers. We can unroll the loop all the way, as above:

```
public static Code unroll_all (
                         Code i,   // iteration variable
                         Code z,   // initial value
                         int incr, // increment
                         int n,    // number of iterations
                         LoopIteration F  // body
                        ) {
    Code C = $< ; >$ ;
    for (int x=0; x<n; x++)
       C = $< `Stmt(C) `Stmt(F.iteration($<`Expr(z)+`Int(x*incr)>$)) >$;
    C = $< `Stmt(C) `Expr(i) = `Expr(z)+`Int(n*incr); >$ ;
    return C;
}
```

The call

```
unroll_all($<i>$, $<0>$, 1, 5,
      new LoopIteration () {
           public Code iteration (Code x) {
               return
                  $< System.out.println(`Expr(x)); >$;
           }
      })
```

produces code (equivalent to)

```
System.out.println(0);
System.out.println(1);
System.out.println(2);
System.out.println(3);
System.out.println(4);
i=5;
```

(Part of the "contract" of this unroller is to increment the index variable by the appropriate amount.)

An advantage of this approach over the first is that the code-generating function has more flexibility. Another "loop unroller" is the one that doesn't unroll at all:

```
public static Code unroll_none (
                         Code i,   // iteration variable
                         Code z,   // initial value
                         int incr, // increment
                         int n,    // number of iterations
                         LoopIteration F  // body
                        ) {
    return $< for (`Expr(i) = `Expr(z);
                   `Expr(i) < `Expr(z) + `Int(n*incr);
                   `Expr(i) += `Int(incr)){
                `Stmt(F.iteration(i))
              } >$;
}
```

In fact, there are occasions when this produces better code than full unrolling. For example, we can apply loop unrolling to the dot product calculation given in the first part of this paper:

```
static Code init (double[] V, Code newV) {
    Code C = $< ; >$ ;
    for (int j = 0; j < V.length; j++) {
       C = $< `Stmt(C) (`Expr(newV))[`Int(j)] = `Double(V[j]); >$;
    }
    return C;
}

public static Dot codegenDot(double[] V1) {

    Code c = $<public class DotProd implements Dot {
        double[] V1copy;

        public DotProd () {
           V1copy = new double[`Int(V1.length)];
           `Stmt(init(V1, $<V1copy>$));
        }

        public double dot(double[] V2) {
            double sum = 0.0;
            int i;
            `Stmt(Unroll.unroll_none($<i>$, $<0>$, 1, V1.length,
                new LoopIteration () {
                   public Code iteration (Code x) {
                      return $< sum += V1copy[`Expr(x)]*V2[`Expr(x)]; >$ ;
                   }
                }))
            return sum;
        }
```

```
        }>$;
        Dot dotV1 = (Dot)c.create("DotProd");
        return dotV1;
    }
```

The call `codegenDot(Vec1)` produces the class `DotProd`. The constructor of that class will copy `Vec1` into a local array (`V1copy`), and the `dot` method will calculate the dot product of that vector and its argument, `V2`.

For this example, full unrolling of the loop in `dot` can be highly disadvantageous. With an array of size 1000, in fact, the code above (which does no unrolling) is approximately eight times faster in run time than the fully unrolled version (obtained by replacing `unroll_none` by `unroll_all`. The run-time code generation cost is also lower for the `unroll_none` version.

The reason for this difference in speed comes from one of two causes, or a combination of them: The HotSpot runtime system for Java performs optimizations at run time selectively; longer methods are optimized much less aggressively then shorter ones. Also, longer methods can result in misses in the instruction cache, which are very costly. In any case, the phenomenon that long methods, even though they execute fewer instructions, can be slower than short ones, has been frequently observed.

This suggests a compromise: partially unrolling the loops. In fact, this is often done by traditional optimizing compilers. (Full unrolling is not normally possible, because the bounds of the loop are rarely known at compile time.) This unroller has an additional argument, the "block size," that is, the number of iterations that should be unfolded in each loop iteration.

```
    public static Code unroll_part (
                            int BlockSize, // unrolling factor
                            Code i,   // iteration variable
                            Code z,   // initial value
                            int incr, // increment
                            int n,    // number of iterations
                            LoopIteration F  // body
                           ) {
        int loops = n/BlockSize,
            leftover = n%BlockSize;
        Code C;
        if (2*BlockSize > n) // 0 or 1 iteration - don't bother
           C = unroll_all(i, z, incr, n , F);
        else
           C = $< for (`Expr(i)=`Expr(z);
                       `Expr(i) < `Expr(z) + `Int(loops*BlockSize*incr); ) {
                           `Stmt(unroll_all(i, i, incr, BlockSize, F))
                  }
                  `Stmt(unroll_all(i, i, incr, leftover, F))
              >$;
        return C;
  }
```

If the `BlockSize` argument is 1, this gives the same result as `unroll_none`, and if it is larger than or equal to `n`, it gives the same result as `unroll_all`.

For example, when run on a vector of 1000 elements, the most efficient dot product computation (counting only runtime, not code generation time, and repeating the dot product computation 100,000 times), was achieved using a blocking factor of 24. Its execution speed was about 14% faster than no unrolling, and about ten times faster than complete unrolling.

It would also be useful to be able to make unrolling decisions based on the size or complexity of the loop body `C`, but we have at present no facility in Jumbo to produce such information.

5.2 A sorting component

Higher-order program generators are often needed to achieve generality. Sorting routines are among the most common "components," but they usually lack generality or efficiency or both. They lack generality in part because they usually have a fixed type of sort key (a requirement of most statically-typed languages). More subtly, they lack generality in that they are efficient for large data sets but not small ones; the programmer who needs to sort just two or three values must always write the sort by hand or pay the price of a function call. Sort routines are inefficient because the comparison operation must be provided as a method and therefore requires a call for each comparison (or build in a single comparison operation and sacrifice generality).

Using Jumbo, we can write a sort component that will allow for different types of sort keys, can be inlined for small data sets, and can incorporate new comparison operations without requiring a method call for each comparison.

In this code, a `SortCall` is a function (that is, a function object) that, given the name of an array to sort, produces a "call" to a sorting routine. This "call" may be an actual method call, or, if the array is small, it may be the sorting code itself. In either case, it is a piece of code that can be inserted into the client and will result in the array being sorted. The `makeSorter` method produces a `SortCall` and, if required, produces a class containing a sort function. Either way, both the type of the array and the comparison function are built into the generated code, eliminating the need for a method call for each comparison.

The arguments to `makeSorter` are a type, a comparison function (given as a function from the code for the arguments to the code for the comparison expression), and a length.

```
public class Sort {
    public static void main(String[] argv) {
        SortMaker sm = new SortComponent();
        SortCall inline_it =
               sm.makeSorter(Type.int_type,
                             new Comparison () {
                                public Code make(Code x, Code y) {
                                   return $<‘Expr(x) < ‘Expr(y)>$;
                                }
                             },
                             2);
        SortCall call_it =
               sm.makeSorter(Type.int_type,
                             new Comparison () {
                                public Code make(Code x, Code y) {
                                   return $<‘Expr(x) > ‘Expr(y)>$;
                                }
                             },
```

```
                             100);
        Code c = $<
            public class SortClient {
         ...
                 int[] B = new int[2];
                 int[] C = new int[100];
 ...
                 `Stmt(inline_it.makeCall($<B>$));
                 ...
                 `Stmt(call_it.makeCall($<C>$))
                 ...
                }
            } >$;
            c.generate();
   }
}
```

The generated class, `SortClient`, will contain code to sort the two-element array `B` in ascending order, and a call to a sorting routine for the 100-element array `C` that will sort `C` in descending order. The latter will be specifically for sorting integer arrays in descending order.

```
class SortComponent implements SortMaker {
    public SortCall makeSorter(Type t, Comparison cmpFun, int len) {
        if (len == 2) return new SortInliner(t, cmpFun);
        return new SortCaller(t, cmpFun);
    }
}

interface Comparison {
    Code make(Code arg1, Code arg2);
}

interface SortCall {
    Code makeCall(Code array);
}

interface SortMaker {
    SortCall makeSorter(Type t, Comparison cmpFun, int len);
}

class SortInliner implements SortCall {
    Comparison cmpfun;
    Type t;
    public SortInliner(Type t, Comparison cmpfun)
        { this.t = t; this.cmpfun = cmpfun; }
    public Code makeCall(Code array) {
        return $<
            if (`Expr(cmpfun.make($<(`Expr(array))[1]>$,
                                  $<(`Expr(array))[0]>$))) {
                `Type(t) temp = (`Expr(array))[0];
                (`Expr(array))[0] = (`Expr(array))[1];
```

```
                (‘Expr(array))[1] = temp;
            }>$;

    }
}

class SortCaller implements SortCall {
    static int id = 0;
    String sortclassname;

    public SortCaller(Type t, Comparison cmpfun) {
        sortclassname = "sort"+(id++);
        Code sorter =
            $<class ‘sortclassname {
                public static void sort(‘Type(t)[] A) {
                    int i = 1;
                    while (i < A.length) {
                        ‘Type(t) temp = A[i], j = i-1;
                        while (j >= 0)
                            if (‘Expr(cmpfun.make($<temp>$, $<A[j]>$))) {
                                A[j+1] = A[j]; j--;
                            } else break;
                        A[j+1] = temp;
                        i++;
                    }
                }
            } >$;
        sorter.load();
    }
    public Code makeCall(Code array) {
        return $<‘sortclassname.sort(‘Expr(array));>$;
    }
}
```

6 Jumbo as a generalized component system

The term “software component” has no universally accepted meaning. Intuitively, it is a piece of software that is “mobile” in the sense that it can be deployed on a remote computer easily. Ordinary Java programs are highly mobile, because they they depend only upon the remote machine having the Java Virtual Machine, which is ubiquitous and well standardized, and do not require that the remote computer have a compiler installed. Jumbo routines are ordinary Java programs, so are also mobile. They do require a fairly large API, but so do many Java programs.

Yet Jumbo classes also have some of the power normally associated only with source-level program generators, which are generally considered less deployable for the reasons cited above. Lex, for example, requires the presence of a C compiler. Our scanner generator, having essentially the same functionality and producing essentially the same code, requires no compiler.

We close with an example that is particularly illustrative of the use of Jumbo as a component system. It is a marshaller generator. Given a specification of a data format, it produces a marshaller (aka "serializer", "pickler") which writes data of that format into a byte array.

The obvious question when marshalling data is what the byte stream output should look like, that is, how do the sender and receiver come to an agreement? If there were an efficient, universal data format, this question would be moot, but there is not.

Let us set up the scenario in more detail. A client is hoping to get data from a server. Client and server agree on the general form and meaning of the data. If the server knew the exact byte stream format the client wanted, there would be no problem at all; but it does not. If the client knew exactly how the server represented the data internally, it could send a piece of plain Java code to transform that data into a byte stream; but it does not. If the client and server had a long-term contract, they could agree on these protocols, but the client would rather "shop" for servers dynamically.

As usual, a traditional, compile-time program generator is one solution. The server obtains the program generator from the client, inputs a description of the local data format, and the result is a special-purpose internal-data-to-byte-stream transformer, i.e. a marshaller. (Equivalently, the server could provide a generator to the client, and the client input a description of its desired byte stream format.) Engler gives a marshaller generator as an example of his Magik system [1].

The advantage of using Jumbo is that the process can be carried out without invoking a compiler. This saves time, avoids the need for either party to reveal source code, and eliminates problems related to incompatabilities among compilers.

We illustrate with a simple implementation. The `MarshallerGen` component, which is sent from the client to the server, consists of an operation for each primitive type, plus one for homogeneous collections and one for nonhomogeneous collections:

```
static MarshallerGen send_int () ;
static MarshallerGen send_char () ;
...
static MarshallerGen send_array (int L, MarshallerGen contents) ;
static MarshallerGen send_struct (String[] fieldnames,
                                  MarshallerGen[] fields) ;
```

Each operation says how to convert data of that type into a byte stream. The arguments to `send_array` are the length of the array and the routine to marshall its contents; since the array is homogeneous, only one routine for the contents is needed. The arguments to `send_struct` are the list of field names (i.e. instance variables) of the structure and a marshaller for each; since the structure is nonhomogeneous, there must be as many marshallers as there are fields.

There is, in addition, one top-level routine:

```
Code send (String srctype) ;
```

`send` produces a class implementing the `Marshaller` interface:

```
interface Marshaller {
    public byte[] fillbuf (Object O) ;
}
```

`fillbuf` allocates a byte array of the correct size and fills it from `O`, which must have type `srctype`. (We could generate the interface as well, giving it an argument of type `srctype`, but this would defeat the purpose of having the interface, which is to declare the type of the generated code *before* it is has been generated.)

Our implementation is given in Appendix A. Here we give an example of its use. Suppose the client is to receive a collection of points with corresponding values. Specifically, it should get a pair of arrays, the first being an array of points (pairs of integers) and the second being an array of characters. The server writes the data format description using the operations provided by the component:

```
// Marshaller for Point values
String[] Point_fieldnames = {"x", "y"};
MarshallerGen[] Point_fields = { MarshallerGen.send_int(),
                                 MarshallerGen.send_int() };

// Marshaller for the pair of arrays
String[] Grid_fieldnames = {"locs", "vals"};
MarshallerGen[] Grid_fields = {
             MarshallerGen.send_array(gridsize,
                            MarshallerGen.send_struct(
                               Point_fieldnames, Point_fields)),
             MarshallerGen.send_array(gridsize, MarshallerGen.send_char()) };
Code Mcode = MarshallerGen.send_struct(Grid_fieldnames, Grid_fields).
                                     send("Grid");

Marshaller sender = (Marshaller)(Mcode.create("Marshall0"));
```

This is *fixed* code on the *server*. Different clients may provide different implementations of the component operations. None of the clients know that the server has a type called **Grid**, or represents points using the Java **Point** type, or anything else about the internal format used by the server. The final line above creates an object of type **Marshall0** which implements the **Marshaller** interface; that is, it has the **fillbuf** operation. Using the implementation given in Appendix A, the definition of `fillbuf` is substantially identical to what would be produced by the `javac` compiler from this source (assuming **gridsize** had value 1000 at the time the marshaller was created):

```
public byte[] fillbuf (Object O) {
   Grid S = (Grid)O;
   byte[] buffer = new byte[15 + 9000];
   int buffpos = 0;
   // Write type tag
   buffer[buffpos++] = 4;  // struct type tag
   buffer[buffpos++] = 2;  // length of struct
   // Tag for point array
   buffer[buffpos++] = 3;
   buffer[buffpos++] = 0; // These three bytes are 1000 (base 10)
   buffer[buffpos++] = 3;
   buffer[buffpos++] = 232;
   buffer[buffpos++] = 4;  // tag for Point struct
   buffer[buffpos++] = 2;  // length of Point struct
   buffer[buffpos++] = 1;  // tag for x field (int)
```

```
    buffer[buffpos++] = 1;  // tag for y field (int)
    // Tag for value array
    buffer[buffpos++] = 3;
    buffer[buffpos++] = 0; // These three bytes are 1000 (base 10)
    buffer[buffpos++] = 3;
    buffer[buffpos++] = 232;
    buffer[buffpos++] = 2;  // tag for value contents (char)

    for (int j1=0; j1<1000; j1++)
        buffer[buffpos++] = (byte)(S.locs[j1].x >> 24);
        buffer[buffpos++] = (byte)(S.locs[j1].x >> 16);
        buffer[buffpos++] = (byte)(S.locs[j1].x >> 8);
        buffer[buffpos++] = (byte)(S.locs[j1].x);
        buffer[buffpos++] = (byte)(S.locs[j1].y >> 24);
        buffer[buffpos++] = (byte)(S.locs[j1].y >> 16);
        buffer[buffpos++] = (byte)(S.locs[j1].y >> 8);
        buffer[buffpos++] = (byte)(S.locs[j1].y);
    }

    for (int j2=0; j2<1000; j2++) {
        buffer[buffpos++] = (byte)(S.vals[j2]);
    }

    return buffer;
}
```

A more elaborate marshaller could apply compression, and it could supply additional operations for specific types of data (`send_gif()`, `send_XML()`, etc.).

7 Related work

Several researchers have noted that program generators might be useful in ways that conventional components are not. An example, is Batory's notion of components as *layers* [6]. In this view — greatly simplified — a component is a collection of classes whose *superclasses* are undetermined. (In ordinary class libraries, of course, every class has a fixed, known, superclass, although it can have yet-undetermined *subclasses*.) From our point of view, then, a "layer" is a function from a list of superclass names to a list of classes. This is easy to implement in Jumbo (we simplify here by assuming the component contains just one class):

```
Code myComponent (String superclass) {
     return $< class myClass extends `superclass {
                    ...
               } >$;
}
```

Instantiating a layer means applying it to a particular class name:

```
myComponent("MySuperClass").generate();
```

We believe that many like examples could be adduced.

Jumbo looks very much like a partial evaluation system with explicit staging, like MetaML [7]. (Another example is the JSpec system [5], which performs partial evaluation on Java programs; the programmer indicates where and when specialization should occur in a "specialization class" separate from the Java program itself.) However, it is really quite different. Those systems are designed to be transparent to the programmer, in the sense that the only result of omitting the staging information will be slower execution. Such transparency has obvious advantages, but its disadvantage is that it limits the kinds of transformations that can be performed. For example, abstraction on types is generally not permissible, because the underlying language is statically typed. Also, the kind of transformation we did in our sorting example, where the generated code *might* or *might not* be inlined, does not seem to be possible. In reality, Jumbo is much more like a macro or template system, in which the object language is simply data to the metalanguage.

Indeed, having the same language as metalanguage and object language is rather misleading: it implies a necessary connection between the semantics of the two languages, when this connection is really just a convention. In MetaML and JSpec, this connection is embodied in the promise that the semantics of the program is the same with or without the staging annotations. In Jumbo, such a promise cannot be made, because removing the quotations will not usually result in a legal Java program. The same remark can be make with regard to C++ templates and Lisp macros. In our earlier work [4], the metalanguage and object language were, in fact, distinct — the one a functional language and the other an imperative language. Despite the appearance of having merged the two levels in Jumbo, in reality there is no fundamental difference between that approach and Jumbo. Having said this, there clearly is *some* connection between the meaning of quoted Java and unquoted Java in a Jumbo program. It remains, however, for that connection to be formalized.

8 Conclusions

The Jumbo system correctly compiles most of Java. All the examples shown in this paper compile and run as advertised. More information about Jumbo — including, in the near future, the system itself — can be obtained from our web site, `fuji.cs.uiuc.edu`.

Besides working on specific applications, we have two immediate goals with respect to the Jumbo system itself. The first is to complete Jumbo by implementing inner classes. At present, Jumbo compiles the rest of Java, the only exceptions being certain static checks that are omitted (though they are discovered by the verifier); it produces essentially the same code as `javac`. It will be especially useful to have a complete implementation of Java, because it will then be possible to quote — and thus abstract parts of — *any* Java code.

The other goal is to optimize the code generation process. The structure of Jumbo should permit this to be done very effectively: Even if there is a hole in a piece of quoted Java code, most of the work involved in compiling that code can normally be done statically. In principle, the cost of code generation could be dramatically reduced in most cases. The result would be that code-generating components could be used much more routinely, since the need to amortize its cost would be that much less.

The general idea of compositional compilation has broad applicability. In [4], we described an earlier implementation of the idea that targetted Sparc machine code instead of JVM code. Targetting machine code might be useful for producing code for small computers than cannot support a JVM. The concept can also be applied to other languages, although for many languages the implementation would be far more complex than Jumbo.

Security and correctness are major issues in this context. The behaviors of components can be hard to describe, since their "holes" can be filled by arbitrary Java fragments. (Indeed, this is true also of functional languages in which functions can have side effects, and of object-oriented languages, in which a method can be overridden by another with no enforceable constraints on the latter's behavior.) To what extent can the Jumbo compiler verify safety of the generated code? For that matter, to what extent can it be verified at code-generation time? These questions are the subjects of current research.

In the meantime, the system offers a good deal of power and is, we believe, very encouraging for the use of the two-level approach to code-generating components.

References

[1] Dawson Engler. Incorporating applications semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, USA, October 15–17 1997.

[2] D. Hearn. M.P. Baker. **Computer Graphics**. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[3] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components I: Source-level components. In *First International Symposium on Generative and Component-Based Software Engineering (GCSE'99)*, September 28–30 1999.

[4] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components II: Binary-level components. In W. Taha (ed.), *Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, Montreal, Sept. 20, 2000.

[5] U. P. Schultz, J. L. Lawall, C. Consel and G. Muller. Towards Automatic Specialization of Java Programs. Appeared at ECOOP'99 (slides).

[6] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers". 12th European Conference on Object-Oriented Programming, (ECOOP '98), July 1998.

[7] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 203–217, New York, June 12–13 1997. ACM Press.

Appendix — Implementation of a marshaller generator (section 6)

This code generates marshallers that fill the byte stream first with the tag for the entire value (including all subvalues), and then the values themselves. For simple values, the tag is a single byte (1 for integers, 2 for characters); for homogeneous collections (arrays) it consists of a single byte containing 3, three bytes containing the size of the collection, and then the tag for the elements; for heterogeneous collections (structures) it consists of a single byte containing 4, one byte containing the size of the collection, and then the tags for the each of the elements. The length of the entire byte stream must be calculated so that a buffer of the correct size can be allocated.

The implementation works like this: Each `MarshallerGen` object contains four values: the code to write the tag, the code to write the data, an expression giving the length of the tag, and an expression giving the length of the data. At the top level, the method send produces the class `Marshall0` containing `fillbuf`, the marshalling routine.

```
import uiuc.Jumbo.Util.*;
import uiuc.Jumbo.Jaemus.*;
import uiuc.Jumbo.Compiler.*;

class MarshallerGen {

   // tagCode and dataCode write code into "buffer" at location
   //   "buffpos", and increment "buffpos" to point past value just written
   // tagCode just write the type tag and is independent of the exact
   //   values of the data;  dataCode writes the actual data and must
   //   therefore know the source.
   // tagLeng and dataLeng are expressions giving the length of the
   //   tag and data, respectively;  again, the source of the data is
   //   not important, only its type

   final Code tagCode;
   final Codefun dataCode;
   final Code tagLeng, dataLeng;

   static int nextIterVar = 0;  // generate new loop variables

   MarshallerGen (Code tc, Codefun dc, Code tl, Code dl) {
       tagCode = tc; dataCode = dc; tagLeng = tl; dataLeng = dl;
   }

   public Code send (String srctype) {
      return $<
         import uiuc.Jumbo.Util.*;
         import uiuc.Jumbo.Jaemus.*;
         import uiuc.Jumbo.Compiler.*;

         public class Marshall0 implements Marshaller {
             public byte[] fillbuf (Object O) {
                  `srctype S = (`srctype)O;
                  byte[] buffer = new byte[`Expr(tagLeng) + `Expr(dataLeng)];
                  int buffpos = 0;
                  `Stmt(tagCode)
                  `Stmt(dataCode.apply($<S>$))
                  return buffer;
             }
         }
```

```
   >$;
}

public static MarshallerGen send_int () {
// int:  type tag = 1, tag length = 1, data length = 4
     return new MarshallerGen (
          $< buffer[buffpos++] = 1; >$,
          new Codefun () {
               public Code apply (Code I) {
                   return $<
                      buffer[buffpos++] = (byte)(`Expr(I) >> 24);
                      buffer[buffpos++] = (byte)(`Expr(I) >> 16);
                      buffer[buffpos++] = (byte)(`Expr(I) >> 8);
                      buffer[buffpos++] = (byte)(`Expr(I));
                   >$;
               }
          },
          $< 1 >$,
          $< 4 >$);
}

public static MarshallerGen send_char () {
// char:  type tag = 2, tag length = 1, data length = 1
     return new MarshallerGen (
          $< buffer[buffpos++] = 2; >$,
          new Codefun () {
               public Code apply (Code I) {
                   return $< buffer[buffpos++] = (byte)(`Expr(I)); >$;
               }
          },
          $< 1 >$,
          $< 1 >$);
}

public static MarshallerGen send_array (final int L,
                                        final MarshallerGen contents) {
// homogeneous collection of contents, of length L:
//    type tag = 3, plus length (max 2^25-1), plus tag of contents
//    tag length = 1 + 3 + length of content tag
//    data length = length of content times number of elements (L)
     return new MarshallerGen (
        $< buffer[buffpos++] = 3;
           // Maximum of 2^25-1 = 33,554,432 elements
           buffer[buffpos++] = (byte)(`Int(L) >> 16);
           buffer[buffpos++] = (byte)(`Int(L) >> 8);
           buffer[buffpos++] = (byte)(`Int(L));
           `Stmt(contents.tagCode) >$,
        new Codefun () {
           public Code apply (Code A) {
               String J = "j"+(nextIterVar++);
               return
```

```
                $< for (int `J=0; `J<`Int(L); `J++) {
                     `Stmt(contents.dataCode.apply($<(`Expr(A))[`J]>$))
                   } >$;
             }
          },
          $< 4 + `Expr(contents.tagLeng) >$,
          $< `Int(L)*`Expr(contents.dataLeng) >$ );
   }

   public static MarshallerGen send_struct (final String[] fieldnames,
                                          final MarshallerGen[] fields) {
   // heterogeneous collection with given fieldnames and corresponding
   // marshallers:
   //    type tag = 4, plus length (max 255), plus tags of fields
   //    tag length = 1 + 1 + length of field tags
   //    data length = sum of lengths of fields
        Code tagCode =
             $< buffer[buffpos++] = 4;
                // Maximum of 255 fields
                buffer[buffpos++] = (byte)(`Int(fields.length)); >$;
        for (int i=0; i<fields.length; i++)
             tagCode = $< `Stmt(tagCode) `Stmt(fields[i].tagCode) >$;

        Codefun dataCode = new Codefun() {
             public Code apply (Code A) {
                  Code data =  $< ; >$ ;
                  for (int i=0; i<fields.length; i++)
                     data = $< `Stmt(data)
                              `Stmt(fields[i].dataCode.apply(
                                    $<(`Expr(A)).`Name(fieldnames[i])>$)) >$;

                  return data;
              }};

        Code tagLeng = $< 2 >$;
        for (int i=0; i<fields.length; i++)
             tagLeng = $< `Expr(tagLeng) + `Expr(fields[i].tagLeng) >$;

        Code dataLeng = $< 0 >$;
        for (int i=0; i<fields.length; i++)
             dataLeng = $< `Expr(dataLeng) + `Expr(fields[i].dataLeng) >$;

        return new MarshallerGen(tagCode, dataCode, tagLeng, dataLeng);
   }
}
```