Source-level Optimization of Run-Time Program Generators*

Samuel Kamin, Barış Aktemur, and Philip Morton

University of Illinois at Urbana-Champaign, USA
{kamin, aktemur, pmorton}@cs.uiuc.edu

Abstract. We describe our efforts to use source-level rewriting to optimize run-time program generators written in Jumbo, a run-time program generation system for Java. Jumbo is a compiler written in *compositional* style, which brings the advantage that any program fragment can be abstracted out and compiled to an intermediate form. These forms can be put together at run-time to build complete programs. This principle provides a high level of flexibility in writing program generators. However, this comes at the price of inefficient run-time compilation. Using source-level transformations, we optimize the run-time generation of byte code from fragments, achieving speedups of 5–15%. We discuss the optimization process and give several examples.

1 Introduction

Jumbo [1–4] is a Java compiler with code quotation and anti-quotation for run-time program generation (RTPG). In this, it is similar to such systems as MetaML [5], MetaOCaml [6], 'C [7–9], and DynJava [10]. However, it has a unique design based on the principle that, if the static compiler is structured "compositionally," there need be only that one compiler — its back end can serve as the code-generating engine for RTPG. We have in the past [3] described the advantages of this approach, and will again briefly do so in Section 2. It does, however, possess one distinct disadvantage: inefficiency. This paper describes our on-going efforts to address this problem.

In each of the systems just mentioned, program generators are created by using a code quotation/anti-quotation syntax. For example, in Jumbo, the notation `$<while (x>0) 'Stmt(getBody())>$` indicates that, at run time, a while statement is to be generated with its body returned from the method call `getBody()`. Since the latter is not known at compile time, it is called a "hole." Quotation marks, `$<` and `>$`, can contain an entire compilation unit — an interface or list of classes — or a fragment as small as a single variable or constant. Quoted code cannot be compiled to Java virtual machine (JVM) code: either it has holes or it is not a full class and is therefore missing necessary context, such as field declarations. (Technically, it is legal to quote a complete compilation unit, without holes, but it is pointless, since it could be compiled statically.)

* Partial support for this work was received from NSF under grant CCR-0306221.

That fragments cannot be compiled all the way to bytecode does not mean they cannot be compiled at all. Consider the case of a quoted class definition with a hole where a method should go. In essence, we have a partial evaluation problem: The compiler has two inputs — the quoted class and the method — of which only the class is known at compile time. It is quite plausible that we might apply the compiler to the class and obtain a "residual compiler" that will receive the method and complete the compilation at run time. (The situation is symmetric in the arguments. When the compiler sees the quoted method, it has a similar problem, with two inputs — the method and its surrounding context — of which it sees only one.)

How can we partially evaluate a compiler applied to an incomplete fragment? The first point to note is that an ordinary compiler handles only compilation units; when presented with a smaller fragment, it gives a syntax error; partial evaluation cannot overcome that. The second point is that, given a compiler for a real language, even if we provided a compilation unit (with holes), it would be a practical impossibility to partially evaluate it mechanically.

In Jumbo, we address these problems in two ways. First, the Jumbo compiler is *compositional*. This means it is structured in such a way that small fragments are still meaningful to the compiler; they can be *partially* compiled, to an intermediate representation we call *Code*. The *Code* value of a compound fragment is a function solely of the *Code* values of its subfragments. Thus, in the example above, when the method definition is supplied at run time, it is supplied in its partially compiled form — not as source code or an abstract syntax tree (AST). This *Code* value is somehow placed inside the *Code* value for the class, and the result is compiled to JVM code.

Second, we have written a set of source-level transformations to optimize the compilation of fragments. These are the subject of this paper. In pursuing this strategy, we have also found that the compiler may need to be massaged to make it more susceptible to transformations, though we have yet much to learn about that process. The work is on-going; the results given here represent the current state of our compiler.

In the paper, we elaborate on each of the themes mentioned above. Section 2 explains what we wish to achieve with our system; to give a preview, it argues that the primary reason to insist on a single compiler is *not* to save development time, but rather to ensure a high level of *generality* in the tool we produce. Sections 3 and 4 discuss compositional compilation in general, and its use in Jumbo, respectively. Section 5 describes the analyses and transformations we have implemented and Section 6 gives examples and timing results. In Section 7, we discuss some of the difficulties presented by Java which have limited our success in optimization, and ways to overcome them. Section 8 gives our conclusions. Related work is noted throughout the paper, and is not segregated.

2 Trade-offs in RTPG systems

In previous work [3], we have argued that, in view of the many possible uses of program generation and our relatively modest understanding of those uses, RTPG systems ought to be as general and flexible as possible. Generality has two meanings here: First, it refers to the richness of the language in which generated programs are expressed — the language inside quotations. Second, it refers to the programmer's freedom in dividing her program into fragments.

Is it legal to fill the hole in `$<int m () {‘(hole) return x;} >$` with the declaration `$<int x=10;>$`? How about filling `$<if (y==x) ‘(hole) else ... >$` with `$<break L;>$`? Is the position of the hole in this fragment legal: `$<try ... catch (‘(hole)) { ... }>$`? Can the hole in `$<‘(hole) class C { ... }>$` be filled with `$<import java.util.*;>$`? These are the kinds of questions we would ask to probe the generality, in the second sense, of an RTPG system.

Different systems make different trade-offs among the values of *generality*, *safety*, and *efficiency*. On the whole, safety and efficiency compete with generality. Disallowing the insertion of variable declarations, for example, allows the types of all variables used in a fragment to be known, and thereby promotes safety and efficiency, but it certainly constrains the programmer's ability to structure the program-generating process.

In most work on run-time code generation, speed and safety are primary concerns. We know of no system, other than Jumbo, in which the answers to all the questions asked above would be "yes." Consider the question of whether to allow a fragment to contain a declaration whose scope extends beyond that fragment. Partial evaluation-based systems [5, 6, 11, 12] possess the "erasure property" — erasing quotation marks leaves a valid program which is equivalent to the original but is not staged. Thus, they follow ordinary scoping rules for declarations, and the generation process cannot introduce new declarations. Template-based approaches [11, 13], which construct programs at run-time by combining precompiled fragments, are inherently limited to fragments that generate machine code; declarations produce no machine code. Other non-partial evaluation-based system [7, 10] restrict the introduction of declarations to permit faster code generation.

The design of Jumbo gives precedence to generality, in both its meanings. By using the same compiler statically and dynamically, we ensure that the dynamic language is the same as the static one. And by giving each node in the abstract syntax tree its own semantics, and insisting that *any* node — even a declaration — can be left as a hole to be filled in at run time, we ensure the ability to divide up the program into almost arbitrary fragments.

Thus, by using a single compiler for both static and dynamic compilation, we lower development cost — there is no extra work beyond writing the one compiler — and get a system of great generality. On the other hand, we can then offer no safety guarantees, and suffer from inefficiency at run time. The latter is the problem we address in this paper.

3 Compositional compilation

There would be nothing for us to do — we could achieve our goal trivially — if we were willing to carry the compiler with us wherever code was to be generated, or to assume it existed everywhere. We could just emit source code and invoke the compiler from the running program. However, this approach is inherently inefficient, and more importantly, is extremely difficult to use in practice because of portability issues and the fundamental reliance on exporting source, which many organizations will not do.

Instead, Jumbo works by *partially* compiling each quoted fragment, producing a value of type *Code*. We will give the precise definition of *Code* shortly. First, we discuss the structure of our compiler.

The idea of compositionality is that the *Code* value to which any AST translates is a function only of the abstract syntax operator at its root and the values of each of its children. Thus, the compiler is really just a set of definitions of abstract syntax operators, but with arguments of type *Code* rather than *AST*. Examples are:

```
Code makeIfThen (Code cond, Code truebranch)
Code makeVariable(int flags, Type type, String name)
Code makeClass(int flags, String name, String supername,
               StringList implementees, CodeList members)
```

This is the essential difference between this structure and a conventional compilation structure: Instead of creating an AST and then generating JVM code while traversing it, the abstract syntax operators themselves contain the code to compile that syntactic construct.

A preprocessing step translates quoted fragments to abstract syntax operators, in the usual way. For example,[1]

```
Code safePointer (Code ptr, Code computation) {
   return $< if (‘Expr(ptr) == null)
               throw error();
             else ‘Stmt(computation) >$;
}
```

becomes (`0` is the code for binary operator "`==`")

```
Code safePointer (Code ptr, Code computation) {
   return makeStatements(
            makeIfThenElse(
              makeBinOp(0, ptr, nullConstant()),
              makeThrow(makeSelfInvocation("", "error", new List())),
              computation));
}
```

[1] The syntactic category names are needed to allow parsing of holes within fragments. The holes are replaced by special names — `unknownExpr`, `unknownStmt`, etc. — before parsing. Zook et al. [14] describe a way to eliminate these using context-sensitive parsing, but we have not yet implemented their technique in Jumbo.

This program is now statically compiled — that is, as an ordinary Java program. The calls to the abstract syntax operations are part of the program and will be elaborated at run time, after the holes have been filled in. In particular, at run time, *Code* values will be provided for the arguments to `safePointer`, and the returned expression will be evaluated.

Eventually, this *Code* value will be placed inside the *Code* value for a compilation unit, and be ready for the final step of compilation — generating Java .class files containing JVM code. The method `void generate ()` performs this final step. Alternatively, `Object create (String classname)` calls `generate`, and then loads the class file and returns an object of the class. `generate` is for when Jumbo is used for off-line program generation, and `create` for true run-time program generation.

We have achieved our goal of allowing for partial compilation even for fragments of programs: the compilation of any fragment, *compile*(A), is its *Code* value, obtained by evaluating the expression to which the fragment is translated by the preprocessor. Holes are handled with no special effort — they are just expressions within this larger expression which do not happen to be explicit calls to abstract syntax operations. Mathematically, we can regard a fragment with a hole, $P[\cdot]$, as being compiled to a function from *Code* to *Code*:

$$\begin{aligned} &\mathit{compileWithHole}(\mathcal{P}[\cdot]) = \lambda C : \mathit{Code}.\ \mathit{compile}(\mathcal{P}[A]) \\ &\qquad \text{where } A \text{ is } \textit{any} \text{ fragment such that } \mathit{compile}(A) = C. \end{aligned}$$

Compositionality ensures that this function is well defined.

4 Jumbo

As discussed in [15, 16], there are many choices for the *Code* type. A degenerate version of compositional compilation is to make *Code* be AST's, and let `generate` do all the work. In Jumbo, our goal is to put as much meaning as possible into *Code*, leaving `generate` very simple. The most natural way to do this is to make each *Code* value a function taking the compilation context (or "environment") to JVM code. This is how compositionality is achieved in defining abstract meanings of programs in denotational semantics [17], and it works just as well when "abstract meaning" is replaced by "compilation."

In Java, the situation is a bit more complicated, but for the most part we follow this idea. *Code* values are represented by objects having a single method, plus some additional information:

$$\mathit{Code} = \mathit{ExportedDefinitions} \times (\mathit{Environment} \to \mathit{ClosedCode})$$

$$\mathit{ExportedDefinitions} = (\mathit{ClassInfo} + \mathit{MethodInfo} + \mathit{FieldInfo})$$

$$\mathit{Environment} = \text{stack of } (\mathit{ClassInfo} + \mathit{MethodInfo} + \mathit{LocalInfo})$$

$$\mathit{ClosedCode} = \mathit{JVM\ code} \times \mathit{integer} \times \mathit{integer} \times \mathit{VarDecls} \times \mathit{Value}$$

The first component of *Code* is the declarations exported from the code fragment. The second is the function we have been referring to above, which we call *eval*; it does the actual translation to JVM code. Exported declarations are just that: declarations that are in scope outside of this fragment. Based on the exported declarations of a class's members, the class can create a fairly complete record of its contents, and that record (a *ClassInfo*) will be its exported declaration. The *eval* method is given an environment containing all enclosing classes, methods, and variables, and then generates code. The two integers in *ClosedCode* give the next available location for local variables and the gensym seed, needed to assign unique names to anonymous classes. The *VarDecls* value carries the local variable declarations of that code fragment. The *Value* field gives the constant value of an expression, if it has one; the Java language definition [18, section 15.27] requires this.

In the implementation, `Code` is an abstract class with two methods: `DeclarationList getDecls ()`, and `ClosedCode eval (Environment)`.

The definition of *Code* is quite a delicate one, and we have gone through several iterations. The current definition is parsiminious in the sense of having as few components as we think is possible. We now explain briefly why this definition works. In Java, names fall under two scope rules: names defined within a method — local variables and inner classes — are in scope in statements that follow the declaration ("left-to-right" scope), while names defined in a class — fields and inner classes — are in scope everywhere within the class (with the exception that fields are not in scope in their own initializers). The exported definitions in *Code* are used to create the latter part of the environment; the environment passed into the *eval* function of the methods of a class contains all the fields and inner classes of that class. Names with left-to-right scope are passed along in the environment from one statement to the next, using the *VarDecls* in *ClosedCode*. Thus, the *eval* function for each statement gets an environment containing all the names in scope at that statement.

(Aside to Java experts: This definition is actually a little bit too parsiminous, in that it does not allow a proper treatment of free variables in inner classes. The rule about inner classes is that each variable captured by an inner class becomes a field of the inner class, and the constructors of the inner class must assign the variable to its corresponding field. The question is, how do we know which variables are actually used in an inner class? This information does not come from the exported definitions of the inner class, since references are not definitions, nor is it passed "left-to-right." We finesse this problem by assuming that all variables in scope in an inner class are referenced in that class. This gives a correct, but obviously non-optimal, implementation of inner classes. An earlier version of Jumbo had an additional "pass" — that is, another method in *Code* — whose purpose was to gather free variable references in classes; we removed it for reasons discussed in Section 7.)

So, our task comes down to this: In a Jumbo program, sections of quoted code become expressions of type *Code*. At run time, these expressions will be evaluated, producing a *Code* object whose `getDecls` and `eval` functions will

then be invoked. We wish to optimize this entire process, but mainly the `eval` function of each *Code* value, since this is where most of the compilation occurs.

5 Source-level optimization of Java

In this section, we describe the optimizations we apply. These take the form of source-level transformations, including method inlining, constant propagation, and various simplifications.

At present, these optimizations are not all applied automatically. A number of transformations are "contractive" — simply put, they never make things worse — and they are applied repeatedly in a "clean up" process. Others — such as inlining — are potentially dangerous, in that they can lead to code expansion, and the system must be told to perform them. (The user interface highlights all inlinable methods and constructors, and the user clicks on the method name to inline it.) We intend to explore methods of automating the entire process in future research.

The transformations are mainly standard and will be described only briefly. We emphasize that all are valid transformations in Java. The idea is not to build an optimizer specific to our compiler, but to use the logic of Java to optimize it. On the other hand, the specific choice of analyses and transformations was made with knowledge of the compiler.

To simplify rewriting, we first normalize the code. There are three main parts of the normalization step:

FQCN: Converts every name to its fully qualified version. For instance, a field access `x` becomes `this.x`, and a field declaration `Code c;` becomes `uiuc.Jumbo.Compiler.Code c;`. (`uiuc` is our university's domain name, so it is the root of package names that reside here.)

For-While: Converts for-loops to while-loops. Also, each while-loop's condition is replaced by `true` and taken inside the loop. The flow goes out of the loop with a break-statement.

Flattening: Breaks complex expressions into simpler expressions. For instance, after this step, all the arguments going into a method call will be simple variables.

We can then apply the following rewrites. All must be applied "manually" — that is, by explicitly requesting the rewriting engine to apply them. However, Cleanup incorporates many of them in a fixpoint iteration; those are not normally invoked manually.

Inlining: Inlines a method invocation. Replaces return-statements of the inlined method with break-statements.

WhileUnroll: Unrolls the first iteration of a specified while loop.

AnonClassConvert: Converts anonymous classes to non-anonymous inner classes.

ConstructorInlining: This transformation is described below.

Unflatten: Transforms the flattened program to a form that is more readable.

Cleanup: Runs the following rewrites in a fixpoint iteration. Each can be invoked manually, but there is little reason to do so.
- **Untupling:** Extracts a field from a newly created object.
- **UnusedDecl:** Removes declarations that are never used.
- **UnusedScope:** Removes scopes that have no semantic significance.
- **UnusedDef:** Removes variable definitions that are not used.
- **UnusedReturn:** Eliminates assignment of a method call when the assigned variable is not used. The method call must still be executed for its side effects.
- **IfReduction:** Simplifies if-statements whose condition is a constant boolean.
- **Arithmetic:** Simplifies constant-valued arithmetic and logic expressions.
- **UnusedBreak:** Removes break statements that make no difference to the flow.
- **ConstantPropagation:** Moves constant values through local variables.
- **CollapseSystemCalls:** Collapses `intern` and `equals` calls made on `String`s.
- **ArrayLength:** Replaces `array.length` expressions with the length, if available.
- **Switch:** Reduces constant switch statements to the match.
- **CopyAssignment:** Propagates redundant assignments of variables and literals.
- **UnusedObject:** Removes object creation statements if they are never used and side-effect-free.
- **FieldValue:** Propagates values through object fields assigned directly.
- **TightenType:** Makes types more specific, if possible.
- **UnusedFieldAssign:** Removes unused assignments to fields.
- **UnreachableCode:** Removes code which is indicated to be unreachable by the flow analysis.
- **ObjectEquality:** Replaces `(obj1 == obj2)` with `true`, and `(obj1 != obj2)` with `false`, if it can determine whether the two objects point to the same location; and vice versa.
- **PointlessCast:** Removes cast expressions where the target of the cast is already the right type.
- **WhileReduction:** Removes while statements which only have a `break` as the body and/or `false` as the condition.
- **InstanceOf:** Attempts to resolve `instanceOf` expressions.
- **NullCheck:** If it can prove that an object `o` is not null, then replaces `o != null` with `true` and `o == null` with `false`; and vice versa.

These rewriters use the information obtained from program analyses. The analyses are **Dominator**, **Flow**, **Use-Def** and **Alias**. The first three are standard. Our alias analysis is described in [19].

5.1 Constructor Inlining

Most of our transformations and analyses are strictly intra-procedural. This makes inlining very important for exposing opportunities for optimization. Constructors cannot be inlined like methods, because there is no notation to create

an uninitialized object in Java; this is an implicit effect of each constructor. (If we were optimizing JVM code instead of source, this would not be a problem.) We might try to use the zero-argument constructor for this purpose, but it might have an explicit definition that conflicts with the definition of the constructor we are attempting to inline. We solve this problem by adding *annotations*, containing the statements of the constructor, to object creation sites. Other rewriters then see the constructor code as though it was just an inlined method. The constructor itself, which resides in a separate class, cannot be optimized, but values propagated out of it can be used in the calling program. The annotations must be removed before the optimized program is written; for this reason, the annotations must have the property that they can be removed at any time and leave a program with the same meaning as when they were there.

6 Examples

We demonstrate the effect of our optimizations via three examples. The first is a complete (but small) class, without holes. The other two are the classic (in the field of program generation) exponentiation function, and a program to generate finite-state machines.

For each example, we show the original program, with quoted fragments. The latter will be preprocessed away and transformed to calls to abstract syntax operators, as described in Section 3. The resulting program is an ordinary Java program that will be compiled into JVM code and executed. At run time, the various *Code* values produced by these expressions will be brought together to form a *Code* value representing a class. A call to `generate` or `create` will turn this `Code` value into a Java .class file. In our examples, we are not executing the generated programs, since we are interested only in code generation time. In each test, we let the virtual machine "warm up" — load the Jumbo API, `java.lang`, and other classes — before executing the programs, then run each test 500 times. Our measurements exclude I/O time for outputting the .class file.

To obtain the optimized versions of the programs, each quoted fragment is optimized, in isolation, after it is preprocessed, using the rewritings described in the Section 5.

For each run — original or optimized — we measure the overall time, and we also measure the time spent in the method `Class.forName`. This method does the run-time look-up for names used but not defined in the program (for example, classes defined in imported packages). It consumes such a large portion of run-time compilation time — more than 50% in most cases — that its effect on speed-up is often substantial. Furthermore, these calls are impossible to eliminate by any static optimization, since the imports must be elaborated on the target machine (i.e. at run time). Since this cost is specific to Java, it is interesting to see what speed-up we would be getting if this cost could be ignored.

The tables in this section have two columns for each of three different Java virtual machines: Sun's HotSpot, Kaffe (an open source VM), and IBM's pro-

duction VM. (HotSpot is included because it is the the most widely used virtual machine, but none of the three is distinctly better than the others, nor is any of them the "definitive" virtual machine.) For each VM, we give the overall execution time and the execution time excluding `forName` calls; these are the "w" and "w/o" columns, respectively. The tables have three rows: unoptimized time, optimized time, and speed-up ((unoptimized time - optimized time) / unoptimized time).

The timings are in seconds. Tests were run on an AMD Duron 1GHz processor, with 790 MB of memory, running Debian Linux.

6.1 Simple class

To get a kind of baseline, we show the results of optimizing a complete, but simple, class. The tests just invoke `generate` on this code:

```
$<
public class Temp {
  int x;

  int id() {
    return 12;
  }
}
>$
```

When presented with a complete class without holes, the rewriters ought to be able to reduce it to a very efficient form. However, the speedups are not as great as we would hope. (In the case of the IBM JVM, the rewriting actually produced a slow-down.) Reasons for this are discussed in Section 7.

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	0.46	0.44	0.79	0.76	0.42	0.41
Rewritten	0.40	0.38	0.66	0.64	0.47	0.46
Speed-up	13.0%	13.6%	15.2%	15.8%	-11.9%	-12.2%

Table 1. Run-time generation performance for the simple example.

6.2 Exponent

The exponentiation function generator creates a function that computes x^n for given value of n. Table 2 gives the performance of the original and rewritten programs.

```
interface ExpClass
{ public int exponent(int x); }

public class Power {
  public static ExpClass getExp(int n) {
        Code r = $<1>$;
        for(int i = 0; i < n; i++){
            r = $<‘Expr(r) * x>$;
        }
        String cname = "Power"+n;
        Code expcl = $<
            public class ‘cname implements ExpClass {
                public int exponent(int x) {
                    return ‘Expr(r);
                }
            }
        >$;
        return (ExpClass)expcl.create(cname);
    }
}
```

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	2.79	0.98	2.45	1.29	4.55	2.98
Rewritten	2.70	0.89	2.36	1.09	4.19	2.63
Speed-up	3.2%	9.2%	3.7%	15.5%	7.9%	11.7%

Table 2. Run-time generation performance for the Exponentiation example.

6.3 FSM

Another application of RTPG is generation of finite state machines (FSM). Table 3 gives the program generation timings for this example.[2]

The example is discussed in [3] and here we give its main class. Due to space considerations, we do not give the source of other classes. (`ArrayMonoList` is just a type of list; here it is used to collect all the cases in the switch statement that is the heart of the FSM implementation.)

[2] It is notoriously difficult to understand the performance of Java virtual machines, and Table 3 is an example. The calls to `forName` are a large percentage of the execution time on all VMs. Furthermore, these calls are identical in optimized and unoptimized code. Yet speed-ups in two cases actually *decline* when `forName` is discounted. This is because, even though optimizations do not touch this method, it runs faster in the optimized than in the original code. We have, at present, no explanation for this behavior.

```
public class FSM {
    String FSMclassname;
    State[] theFSM;

    FSM (String c, State[] M) { FSMclassname = c;  theFSM = M; }

    Code genFSMCode () {
        ArrayMonoList body = new ArrayMonoList();

        // Each state corresponds to a case in the switch statement
        for (int i=0; i<theFSM.length; i=i+1){
            body.addAll($<case ‘Int(i):
                          ‘Stmt(theFSM[i].genStateCode("ch"))
                          break; >$);
        }

        Code result =$<
            import java.util.*;

            public class ‘FSMclassname {
                static void runFSM (StringTokenizer in) {
                    int theState = 0;
                    while (true) {
                        char ch;
                        if (!in.hasMoreTokens()) return;
                        ch = in.nextToken().charAt(0);
                        switch (theState) {
                            ‘Case(body)
                            default: return;
                        }
                    }
                    return;
                }
                static void addToBuffer(char ch){ ...  }
                static void emitbuffer(){ ...  }
                public static void main (String[] args) {
                    String input = ...; // obtain input from console
                    runFSM(new StringTokenizer(input));
                }
            }>$;
        return result;
    }
}
```

The constructor of this class takes a finite-state machine description in the form of an array of states; the client sends the `genFSMCode` message to that object and then invokes `generate` on the result. The created class contains a main method that reads a string from the console and runs the client's FSM on it.

We haven't shown an FSM description due to space limitations, but to give a general idea, an FSM description is a set of states, and each state is a set of transitions. Here is the definition of a single transition.

```
new Transition(new Predicate1 (), 1, new Action2 ())
```

where

```
class Predicate1 implements Predicate {
    public Code pred (String ch) {
      return   $<('a' <= `ch && 'z' >= `ch)
          || ('A' <= `ch && 'Z' >= `ch )>$ ;
    }
}

class Action2 implements Action {
    public Code action(int s, String ch) {
        return $<addToBuffer(`ch);>$;
    }
}
```

This transition, if it sees a letter, goes from its current state to state 1 and puts the letter into the buffer.

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	13.10	4.93	14.01	8.82	8.92	3.89
Rewritten	12.25	4.76	13.48	7.78	8.37	3.70
Speed-up	6.5%	2.9%	3.9%	11.8%	6.2%	4.9%

Table 3. Run-time generation performance for the FSM example.

7 Lessons learned and future work

Compositional compilation can be applied to any language, yielding a compiler that supports run-time program generation (once the quotation/anti-quotation syntax is added). Each language will present different issues, both in construction of the compiler and in optimizing run-time program generation. Java is in some ways highly suitable for this treatment. Because it has no preprocessor and no optimization pass to speak of, most of the compiler consists of a translator from AST's to low-level code — the process to which compositionality applies most naturally. But in another sense, Java is *too dynamic*; some compilation steps must be performed dynamically that, in other languages, can be performed statically. Obviously, anything the must be done at run time cannot be optimized away. In this section we discuss why we have not gotten better speed-ups, and our future plans.

7.1 Optimization Problems

The major issue blocking rewriting is resolution of class names. The Java definition requires that these names be resolved on the target machine. Thus, for example, the test to determine if a method override is legal — which must be done for every method — cannot be eliminated, because the superclass is available statically only in the rare case when it is defined in the quoted fragment itself.

Similarly, the normalization of class names (conversion of a short class name to a fully qualified class name) for variable, field, and method declarations must be done dynamically. This necessitates that the fields keeping track of type information be mutable: The objects containing those fields are the class and method objects created by `getDecls`, but normalized class names cannot be filled in until `eval` is called. Moreover, these objects are returned from `getDecls` to `generate`, so unless the fragment being optimized is in a place where the `generate` call can be inlined — which it usually is not — the class and method information have to be considered to have "escaped." Propagating information through mutable fields of objects that escape is very difficult.

One result is that the optimized code generator still contains type checks which we would initially have expected could be eliminated, such as a check for the validity of the return statement in `$<int foo() { return 5; }>$`.

Even if the fragment being optimized consists of a complete class, it is possible that the consumer of the fragment will compile it in a larger context: adding import statements, adding sibling classes, or making it an inner class. Not knowing this context causes more class name resolution problems. For example, if an enclosing class contains a field named "`java`", then "`java.lang.Object`" represents a series of field lookups, not a fully qualified class name, Having an explicit `create` or `generate` call available in the code being optimized resolves this difficulty, because it tells us that the fragment we see will not be placed in any larger context. However, as noted above, this will not normally be the case.

7.2 Next steps

We have continually refined our compiler in two ways. One is reducing the number of "passes" — that is, the number of functions in *Code*. The idea is that putting more work in a single pass makes more information available locally; with multiple passes, each called from `generate`, the connection from one pass to another cannot be inferred except in those cases where we can see the `generate` call and inline it. As mentioned in Section 4, the current structure is as compact as we think is possible.

The other refinement is making the fields in the compiler's classes final. There is a bit more we can do along these lines.

More broadly, however, Java fundamentally limits optimizations because of the requirement to locate classes dynamically. This entails run-time calls to `forName`; in one case — the exponentiation example in HotSpot — `forName` consumes 65% of run-time compilation time. We have also noted above how

dynamic class locating has a cascading effect: it requires that certain fields be mutable, which in turn diminishes our ability to statically determine their values.

It would be an interesting exercise to see what we could achieve if we assumed that imported classes could be looked up at compile time. But Jumbo is a compiler for Java, not an idealized or subsetted version of Java, and we do not want to change that. Nor would this be a simple experiment: we have pointed out in this section how this property of Java has pervasive effects in the compiler; vacating this property would have correspondingly pervasive effects. So, our current thinking is that it may be time to apply our approach to a more conventional, less dynamic, language like C, and this is an avenue we are actively exploring.

8 Conclusions

We have shown how source-level optimizations can improve the performance of a program generation system based on the principle of compositional compilation.

The Jumbo compiler was first publicly released in 2003. We began the current study from (the newest version of) that compiler, but found that compositionality alone was not enough to permit optimization. We rewrote the compiler to be (a) *more* compositional — where the first definition of *Code* contained four functions, the current one has two — and (b) more functional in style, making greater use of final fields. It seems reasonable to us that, since RTPG can offer very significant performance advantages, the compilers to support it might be written so as to allow for more efficient code generation. In any case, in our future development of Jumbo, we will think of it as a process of co-design: writing optimizations that apply to the compiler, and modifying the compiler to make the optimizations applicable.

Though our speed-ups are still modest, we consider our results encouraging. There remain possibilities for further rewriting, even in Java, and we fully expect to see better results as we develop both the optimizations and the compiler. We are also exploring the application of our ideas to other, more static, languages.

9 Acknowledgements

We thank the reviewers, who provided numerous helpful comments on this paper.

References

1. Kamin, S., Clausen, L., Jarvis, A.: Jumbo: Run-time Code Generation for Java and its Applications. In: Proc. of the Intl. Symp. on Code Generation and Optimization (CGO '03), IEEE Computer Society (2003) 48–56
2. Clausen, L.: Optimizations In Distributed Run-time Compilation. PhD thesis, University of Illinois at Urbana-Champaign (2004)

3. Kamin, S.: Routine Run-time Code Generation. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03), ACM Press (2003) 208–220 Also appeared in: SIGPLAN Notices, vol. 38 (2003), pp. 44-56.
4. Kamin, S.: Program Generation Considered Easy. In: Proc. of the 2004 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM '04), ACM Press (2004) 68–79
5. Taha, W., Sheard, T.: MetaML and Multi-stage Programming with Explicit Annotations. Theoretical Computer Science **248** (2000) 211–242
6. Taha, W., Calcagno, C., Leroy, X., Pizzi, E.: MetaOCaml. (`http://www.metaocaml.org/`)
7. Engler, D.R., Hsieh, W.C., Kaashoek, M.F.: 'C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation. In: Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '96), ACM Press (1996) 131–144
8. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, M.F.: 'C and tcc: A Language and Compiler for Dynamic Code Generation. ACM Transactions on Programming Languages and Systems **21** (1999) 324–369
9. Poletto, M., Engler, D.R., Kaashoek, M.F.: tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In: Proc. of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97), ACM Press (1997) 109–121
10. Oiwa, Y., Masuhara, H., Yonezawa, A.: DynJava: Type Safe Dynamic Code Generation in Java. In: The 3rd JSSST Workshop on Programming and Programming Languages (PPL2001). (2001)
11. Consel, C., Lawall, J.L., Meur, A.F.L.: A Tour of Tempo: A Program Specializer for the C Language. Sci. Comput. Program. **52** (2004) 341–370
12. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: an expressive annotation-directed dynamic compiler for C. Theoretical Computer Science **248** (2000) 147–199
13. Hornof, L., Jim, T.: Certifying compilation and run-time code generation. In: Partial Evaluation and Semantic-Based Program Manipulation. (1999) 60–74
14. Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ Programs with Meta-AspectJ. In Karsai, G., Visser, E., eds.: Proc. of the Third Intl. Conf. on Generative Programming and Component Engineering (GPCE 2004). Volume 3286 of Lecture Notes in Computer Science., Vancouver, Canada, Springer (2004) 1–18
15. Kamin, S., Callahan, M., Clausen, L.: Lightweight and Generative Components-1: Source-Level Components. In: Proc. of the First Intl. Symp. on Generative and Component-Based Software Engineering (GCSE '99), Springer-Verlag (2000) 49–64
16. Kamin, S., Callahan, M., Clausen, L.: Lightweight and Generative Components-2: Binary-Level Components. In: Proc. of the Intl. Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '00), Springer-Verlag (2000) 28–50
17. Stoy, J.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Press (1977)
18. Gosling, J., Joy, B., Steele, G.: The Java Language Definition. Addison-Wesley (1996)
19. Morton, P.: Analyses and Rewrites for Optimizing Jumbo. Master's thesis, University of Illinois at Urbana-Champaign (2005)