Program Generation Considered Easy

Invited Talk

Sam Kamin*
Computer Science Department
University of Illinois at Urbana-Champaign
+1 (217) 333-7505
kamin@cs.uiuc.edu

ABSTRACT

Programmers frequently write program generators using the simple model of programs as text. The essence of this approach is its lack of structure. For this reason, it gets no respect from academic researchers. But the flip side of lacking structure is freedom from restrictions. We argue that the latter is important, and perhaps essential, in finding a willing audience for program generation among working programmers. Jumbo is a system for producing run-time program generators, which is designed to offer the programmer a "programs as strings" model to as great an extent as possible, though some constraints are inevitable. We show by several examples that these constraints still allow for both a natural and a powerful program generation model. We then discuss how the approach taken by Jumbo, though possessing less structure than some competing methods, still raises scientific problems that ought to be of interest to researchers in this area.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *code generation.*

General Terms
Languages.

Keywords
Run-time code generation; Java.

1. INTRODUCTION

Programmers frequently write program generators using the simple model of programs as text. Much experience shows that this method is often efficacious and is not overly difficult. However, it gets no respect from academic researchers. It is perhaps a perfect example of the academic's nemesis: something that works in practice but doesn't work in theory.

In truth, virtually all program generation systems appeal, at some intuitive level, to this model. It is difficult to imagine how a programmer would build or understand a program generator without being able to picture the programs it generates. However, the systems are always enriched in some way, with some structure and, accordingly, some restrictions. The essence of the "programs as strings" model is freedom from constraints.

*This work was partially supported by NSF grant CCR-0306221.

Jumbo is a Java compiler incorporating a code-quotation mechanism. It allows programmers to easily create both compile-time and run-time program generators. Its code quotation mechanism is unusual in its generality: virtually any Java program or program fragment can be quoted, and virtually any part of a program can be left as a parameter for the program generator. This makes for a natural, easy-to-learn, and powerful mechanism.

Jumbo attempts to adhere to the programs-as-strings model as much as possible, because of its simplicity. But it cannot follow it perfectly because the pure programs-as-strings model is inherently compile-time. Jumbo's purpose is to allow for the creation of *run-time* program generators. Thus, there are restrictions: For one, only syntactically coherent parts of a program can be abstracted. More significantly, only string *construction* is permitted, not string *destruction*. This restriction seems to us essential if the generated programs are to be compiled in advance, rather than simply invoking a compiler at run time.

In this paper, we demonstrate first that Jumbo still follows the string model closely enough that it is easy to write program generators with little training. We show this by giving several examples, each consisting of a sequence of program generators for a single domain. The first element of each sequence is so simple that a Java programmer could write it in just a few minutes; the most complex might take a couple of hours. The domains are: literate programming; object-oriented programming patterns; and Huffman coding.

Following these examples, we contrast the Jumbo approach with other program generation methods and discuss some areas of application that we think are of potential interest for the program generation community. Lastly, we make the case that the Jumbo approach to program generation, while perhaps lacking some of the interesting theoretical properties of other approaches, nonetheless provides enough structure to raise interesting scientific questions; in this way, we hope to interest other researchers in our methodology.

2. JUMBO

Jumbo is a compiler for a "two-level" version of Java. It permits the programmer to specify code to be generated using a quote/anti-quote syntax similar to that of MetaML [18]. A fragment of Java code within brackets `$<` and `>$` represents a value of type `Code`. Within those quotations, an expression of the form `` `(*expr*) `` or `` `syntax-category(*expr*) `` causes the *`expr`* to be evaluated and its value, which must be of type `Code`, to be spliced into the quoted code. (These antiquated expressions are sometimes called "holes;" they are the parts of the code that will not be known until program generation time.) The syntax category, which is needed just for parsing purposes, represents the type of fragment that is expected at that position in the code; `Expr` and `Stmt` are the most common cases. (Occasionally, syntax categories are used in the quotes themselves: `$syntax-category< … >$`.)

As a simple example, here is the famous power function example:

```
interface PowerFun { int pow (int x) ; }

static Code genPowerBody (int n) {
  return (n==0) ? $< 1 >$
       : (n==1) ? $< x >$
       : even(n)?
           $< sqr(`Expr(genPowerBody(n/2))) >$
       : $< x * `Expr(genPowerBody(n-1)) >$;
}

static Code genPowerClass (int n) {
  return $< public class PowerClass
                          implements PowerFun {
             public int pow(int x) {
               return `Expr(genPowerBody(n));
           }} >$;
}
```

A client calls `genPowerClass(k)` and gets back a Code object, say `powCode`. It can then do one of the following

```
   powCode.generate();

or   PowerFun p =
         (PowerFun)powCode.create("PowerClass");
     … p.pow(arg) …
```

The `generate` call creates the `PowerClass` .class file and leaves it for later use. The `create` call does this, but then creates an object of the generated class. Generally speaking, `generate` is for compile-time uses and `create` is for run-time uses. (The `PowerFun` interface is needed only for run-time uses.)

The quotation syntax works, intentionally, much like ordinary string quotation. Any Java code can be quoted in this way. In addition to the syntax categories like `Expr` and `Stmt` that can be used in antiquotes, categories `Int`, `Double`, `Char`, and so on, are used to "lift" values of the corresponding type into generated code. That the result is a value of type `Code` rather than `String` should be transparent to the programmer, except in that it implies the restriction mentioned in the introduction: `Code` values can be constructed by quotation and splicing, but cannot be destructed.

The basic idea behind Jumbo is *compositional compilation*. 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 language construct is a function only of the compilation of its sub-constructs, which 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.

3. LITERATE PROGRAMMING

Literate programming is an idea introduced by Donald Knuth [11] in 1984. He argued that programs should be readable as text, and then famously backed up the argument by publishing the implementations of TEX and Metafont [12,13]. Unable to effectively present his programs within the constraints of ordinary program structure, he wrote his programs in a flattened format, called a *web*, using links to connect composite program parts to their constituents. Within this flattened format, sections could be defined and comments written, making it read like a sequential narrative. A program called a "weaver" turned the web into a nicely typeset document, with table of contents, index, cross-references, and section headings. Another program, called a "tangler," extracted the program text from the web and put all the fragments in their correct places so that the program could be compiled. (See `www.literateprogramming.com` for the latest on literate programming.)

This may seem an odd place to begin a discussion about program generation, but one of the primary goals of program generation – compile-time program generation, at least – is to *improve program structure*. This goal is sometimes implicit, but it is always present, even when improvements in efficiency appear to be the driving motivation. After all, the program being generated *could be* produced by hand. But this would involve extra work and, more importantly, would obfuscate the program.

Thus, program generation is, in large measure, an approach to the age-old software engineering problem of making programs more transparent. These were also Knuth's goals in the invention of literate programming.

In any event, the role of the tangler in literate programming is fundamentally program generation. Accordingly, we produce a version of literate programming in Jumbo. We cannot produce a weaver, as we have no way to produce anything but compiled programs. On the other hand, the generality of Jumbo will allow us to go beyond standard literate programming in the presentation of program structure, as we will see.

We illustrate with an example by Cordes and Brown [4]: a program to print the first 25 Fibonacci numbers. Figure 1 is the web for this program[1]; it is drawn from Figure 8 of [4], but modified to produce Java instead of C code (some minor modifications, such as shortening the comments, were made to save space). The tangled program is shown in Figure 2. (The woven – i.e. typeset – version of the program is not shown here, as we are not attempting to produce a weaver.)

We trust that Figures 1 and 2 need no extensive explanation. There is perhaps one notation that is not obvious: on lines 14 and 17 we see "`@<Main variable declarations@> =`" and "`@<Main variable declarations@> +=`", respectively. In all other cases, we just see "=", meaning that the earlier appearance of the placeholder is to be replaced by the code that follows. The "+=" notation indicates that we are adding code to that already substituted for the placeholder.

[1] All figures appear at the end of the paper.

The Jumbo version of this web is shown in Figure 3. Each fragment of code is represented by a function. (We cannot simply assign these code fragments to variables, as every reference to a variable would appear before its definition.) Aside from some additional boiler-plate in each code section, we have fairly reproduced the structure of the original web. (The careful reader will note that we have not reproduced the behavior of the "+=" operator. We will address this at the end of this section.)

Figure 3 is the entire program. Jumbo itself acts as the tangler. Jumbo does not produce source code, but if it did, it would be exactly the same source code as is shown in Figure 2. That is, the JVM code it produces is just what a Java compiler, such as `javac`, would produce from the code in Figure 2.

In Jumbo, we can give another version of the program, one we consider even more "literate" than the first. Literate programming permits the *physical* structure of the program to be presented in a readable form, by flattening it into a sequential narrative. However, the *logical* structure – the way the programmer would *explain* the program – may be quite different. In this program, one could argue that the real heart of the matter is the body of the loop. This suggests an entirely different "layout" of the program. This alternative presentation is given in Figure 4. We again trust that this "web" is sufficiently clear to require no lengthy explanation. The program that Jumbo produces from this input is nearly identical to that generated by the previous examples: we moved the initializations of `fib1` and `fib2`, as their new positions seem to follow the logic of this presentation more closely, and used a generated name instead of `count`, so as to properly implement a "repeat" construct. Note how this presentation highlights the loop body, and also keeps all the termination logic together. (We have also abstracted out the print function, making it easier to modify. This could always be done by procedural abstraction, but the idea here is to preserve the program while changing only its presentation; adding another procedure would be changing the program.)

We return now to the problem of implementing the += operator. The problem is to add new declarations to the declaration section at will, without having an explicit hole for each declaration. This cannot be done directly in Jumbo, as all abstractions of code are explicit. However, we can accomplish this by ordinary programming: declare a `Code` variable `decls`, and whenever a new declaration is needed, at it to `decls` by side effect:

```
  decls = $< `Stmt(decls) … new declaration … >$;
```

In the end, put `decls` into the (single) hole created for it. For space reasons, we cannot show this entire version. But the point is that this problem can be handled by the ordinary programming facilities available in Java.

Aside from some notational awkwardness, we have a system that emulates literate programming, and even generalizes it. We do not have a weaver (although the main contribution of the weaver – providing cross-referencing and indexing – could be provided by a Java cross-referencing tool, of which several exist). Nonetheless, for a quick and easy – and extensible – implementation of literate programming, it's not bad!

4. OBJECT-ORIENTED PATTERNS

Object-oriented patterns [7] are useful programming idioms related to the structure of classes and class hierarchies. Patterns are not formally defined, like programs, or even very *closely* defined, like algorithms. Rather, they are fairly general ideas about how to write programs. Thus, for any given pattern, the variety of programs that can be said, correctly, to employ that pattern is very large. Any attempt to program patterns as program generators must therefore reckon with the impossibility of writing a "complete" implementation of the pattern – if that phrase even has any meaning.

Nonetheless, implementing patterns is a natural and desirable thing to do, for the usual reasons: avoiding duplication of effort, ensuring high quality, enforcing standards, etc. Little program generators are one solution to the quandary. Instead of attempting to write a program generator that handles every possible instance of the pattern, we can write a version of the pattern for our particular problem. If it turns out that our version is not general enough, we have the same recourse as with any other program: fix it. This is practical because most uses of patterns are *not hard to program*; it is only the attempt to program them in great generality that is difficult.

We illustrate with a sequence of implementations of the Proxy pattern [7]. This pattern is used when an object needs to have its behavior altered in some way; another object, or *proxy*, can be used in its place, delegating operations to the original object (called the *subject*) as necessary. Subclassing is one way of implementing the Proxy pattern, but it is not applicable in all cases, and it is generally assumed in discussions of this pattern that subclassing is, for whatever reason, not an option.

The implementation of Proxy can be addressed at two levels of generality: We can create methods of conveniently generating proxies for classes with a given, fixed interface; or we can create methods to generate proxies for any interface. In the latter case, the operations of that interface have to be provided as an argument to the generator, either explicitly by the programmer, or by the programming environment, or by reflection.

We will start our sequence of implementations by implementing proxies in the less general sense. The example is from [8]. The `RequestInt` interface consists of three methods: `safeRequest`, `regularRequest`, and `unsafeRequest`. The proxies "advise" these methods; that is, they use delegation, but perform actions before or after the delegation call. Note that there may be several different proxy implementations of interest – i.e. different kinds of advice – and the subjects that we want to advise may come from different classes (all sharing the same interface, of course).

To give the idea of what we're getting at, we present a proxy in pure Java in Figure 5. For any object implementing the `RequestInt` interface, it adds a counter on calls to `unsafeRequest`. The client uses it by calling

```
  CountingProxy proxy1 =
           new CountingProxy(new Subject());
```

Whatever methods we write to create proxy implementations, we should be able to produce a class like this one as one example.

Our first Jumbo implementation provides the proxy builder with a single method, genProxy, to create the proxy class:

```
static Code genProxy (String proxyname,
      MonoList decls, Codefun safeFun,
      Codefun regularFun, Codefun unsafeFun)
```

The first argument is the name of the proxy class to be generated and the second is a list containing any new variable declarations needed by the proxy code. The third, fourth, and fifth arguments are function objects implementing this interface:

```
interface Codefun {public Code apply (Code c);}
```

These supply the advice for each method – safeRequest, regularRequest, and unsafeRequest – in that order, by giving a code-generating function to be applied to the delegation call. For example, the identity function means the proxy uses pure delegation and provides no advice. Thus, the code in Figure 5 is produced by these statements:

```
Codefun ID = new Codefun () {
  public Code apply(Code call) {return call;}};
Codefun COUNT = new Codefun () {
  public Code apply (Code call) {
    return $< count++;
              `Stmt(call)
              System.out.println(count); >$;
    }
  };
Code c = RequestProxyGen.genProxy (
                 "CountingProxy",
                 $Field< int count = 0; >$,
                 ID, ID, COUNT);
c.generate();
```

The genProxy method is shown in Figure 6.

Our next version makes this just a little easier, especially in those cases where most of the methods are to get the same advice. Here, we put the code functions into a table. The improvement is that the table has a default value, so that we needn't mention the methods that get the standard treatment. In this version, the code of Figure 5 is produced by these statements:

```
CodeMap mods = new CodeMap(ID);
mods.put("unsafeRequest", COUNT);
Code c = RequestProxyGen.genProxy(
                 "CountingProxy",
                 $Field< int count = 0; >$,
                 mods);
```

The function objects ID and COUNT are as above. CodeMap is a collection class based on Map but allowing for a default value in the constructor; to save space, its definition is not shown. This definition of genProxy is given in Figure 7.

Generating proxies for arbitrary interfaces is considerably more complex. In Java, we can obtain the interface specification by reflection. In Figure 8, we present a program generator that does this. To save space, we have omitted the definition of cvtToType, an uninteresting method that converts Class objects to objects of the Jumbo class Type.

The proxy writer's view of the new version of genProxy is almost the same as above. The code just given is altered only to the extent of changing the assignment to c:

```
Code c = ProxyGenerator.genProxy(
          "CountingProxy", “RequestInt”,
          $Field< int count = 0; >$, mods);
```

That is, the generator is a generic “ProxyGenerator” instead of RequestProxyGen, and the interface name is passed as an argument.

That is as far as we will go with this example. As mentioned at the start of the section, any implementation of a pattern will necessarily leave room for improvement. In Jumbo, it was easy to get started, and what we've done can be easily extended.

5. HUFFMAN CODES

Creating run-time program generators[2] in Jumbo is just as easy as creating compile-time generators. Indeed, the program generators we have presented up to now, though intuitively compile-time, could be used at run-time (replacing the generate calls with create calls and adding interfaces as needed). They could even be used on remote machines. The latter might well be useful. For example, one could imagine creating and distributing a library of Jumbo code for literate programming. Run-time use and remote use are essentially the same: both require the ability to generate programs *without* generating source and invoking a compiler.

We present in this section an example of a clear-cut use of *run-time* program generation: Huffman encoding. More precisely, we use program generation to create an efficient *decoder* for a given code. The idea is that the code for a particular file or set of files may be determined dynamically (from a sample of the input), but, once determined, the decoding process can potentially be optimized by generating a decoder specifically for that input.

First, a brief primer on Huffman encoding. A Huffman code is a binary tree whose leaf nodes are labeled with characters, every character appearing exactly once. The code for a character is just the path from the root to that character's node (using 0 for left and 1 for right). To decode a bit string, therefore, is just to follow the bit string from the root to a leaf note, emit the character, and then continue from the root again. The construction of the code is the interesting part, but it is not our concern here.

Assume a type HuffmanTree representing a binary tree, with fields c (for the character), left, and right, the code to decode a string of zeroes and ones (for simplicity, we represent this as a Java string, using the characters ‘0’ and ‘1’ instead of actual bits).

[2] We use the term “run-time program generation” (RTPG) in preference to the more common “run-time code generation” (RTCG) advisedly. “Program generation” has the connotation of source-level specification of generated code, under programmer control, whereas “code generation carries the implication of low-level programming done by “systems programmers.” We believe code/program generation should be done at run time, but want to emphasize that the programmer is fully in control of the process and determines exactly what will be generated; hence “run-time program generation.”

Then the following will output the characters encoded in that string according to the `HuffmanTree tree`:

```
public void decode(String s) {
  HuffmanTree x = tree;
  for (int i=0; i<s.length(); i++) {
    char bit = s.charAt(i);
    if (bit == '0') x = x.left;
    else if (bit == '1') x = x.right;
    if (x.left == null && x.right == null) {
        System.out.print(x.c);
        x = tree;
    }
}
```

(This code was obtained from [16].)

Generating the decoder for a given tree is straightforward:

```
Code makeDecodeBody () {
  if (left == null || right == null)
    return $< `Char(c) >$;
  else return $< (s.charAt(i++) == '0')
                ? `Expr(left.makeDecodeBody())
                : `Expr(right.makeDecodeBody())
              >$ ;
}

Code makeDecodeClass () {
  return
    $< public class Decode
                implements DecodeInt {
         int i=0;  String s; char c;
         public void decode (String s) {
           int l = s.length();
           while (i < l) {
             c = `Expr(tree.makeDecodeBody());
              System.out.print(c);
           }
         System.out.println();
       }} >$ ;
}
```

The method `makeDecodeClass` is used as follows:

```
Code c = tree.makeDecodeClass();
DecodeInt d = (DecodeInt)(c.create("Decode"));
```

In our experiments, the generated code ran an order of magnitude faster than the original code. We are reluctant to report specific numbers, both because we have not done extensive testing and because the Sun Hotspot run-time system is somewhat unpredictable; thus, our numbers might not be reproducible. In any case, the point is that run-time generation is easy to do, and *may* produce speed-ups. (A more careful performance analysis is done in [1], where we generate code for serialization.) Furthermore, *re*generating code is also easy: in some circumstances, it may be advantageous to continue sampling the input, periodically recomputing the code and regenerating the decoder; in others, it may be worthwhile to generate a variety of codes and dynamically choose the one that produces the most efficient decoder (as is done, for example, for FFT codes in FFTW [6]). Though still considered exotic, the use of code generation to gain efficiency in this way could be routine if appropriate tools existed.

6. PROGRAM GEN. CONSIDERED EASY

We hope the examples we've given here convince the reader that program generation in the Jumbo model – basically, programs as strings without destructor operations – is straightforward. The string-based model has advantages that have a direct impact on the practicality – and popularity – of program generation, and which we have attempted to preserve in Jumbo:

The entire language is covered. Programmers can have confidence that any Java program can be quoted and, when generated, will provide the expected results. It would be much easier to implement a "sufficiently large" subset of the language, but that is not enough. Programmers sometimes have idiosyncratic styles based on the use of some unusual idioms; having to change that style would be an undue burden. They also frequently use code that they have imported – and may not even fully understand – that makes use of a feature not covered in the subset. (For Java, the obvious example is inner classes; though exotic, they are used often enough that failure to implement them would render a Java system unusable for most large Java programs.)

Holes are permitted nearly anywhere. In a pure string model, abstraction on any substring in the source code is possible. In Jumbo, abstraction is confined to syntactically sensible locations (and some restrictions are imposed as a result of the complexity of the mapping from concrete to abstract syntax). However, we avoid other restrictions, such as disallowing abstraction on type names or declarations. As another example, Smaragdakis and Batory [17] show why it is useful to abstract on the superclass of a class definition.

The lack of structure in the string model leads to a variety of potential problems. For example, one cannot even guarantee that the constructed program is syntactically correct. (In Jumbo, fragments must be parsable, but it is possible to build an invalid abstract syntax tree by inserting the wrong type of subtree under a node.) But it seems to us that the difficulties do not fundamentally exceed what one encounters in ordinary programming; or, to put it another way, the solution to the possibility of error is careful debugging.

By contrast, other approaches limit the programmer's control over the program construction process in fundamental ways. They do this either to provide programmer support or for efficiency or both. When used to aid the programmer, such limits are often, in our opinion, counter-productive, as they force the programmer to find ways to work around them. When used for efficiency, they preclude the use of the system for many important applications, and therefore offer the programmer no help for those applications. Such restrictions appear in both partial evaluation-based systems and "heterogeneous" – that is, Jumbo-like – systems:

Partial evaluation systems. We refer here to systems based on the idea that the programmer writes one program, paying no mind to staging, and by indicating which of the inputs to the program are static and which dynamic, enables the system to produce a staged program automatically. This is the ultimate simplification of the staging process. However, since the very earliest work on partial evaluation, it has been clear that "paying no mind to staging" is impractical. Much of the research in this area has been about precisely how the programmer can communicate staging information to the partial evaluator. In the end, massaging the input to the partial evaluator so as to produce the desired residual program becomes an art in itself, and we are bound to inquire whether it is actually easier than specifying the residual program directly. More fundamentally, partial evaluation systems are restrictive in the kinds of holes they allow: The only abstractions available are those already present in the host language, so that many useful abstractions – on types, class names, and declarations, for example – are precluded.

`C [5], DynJava [15], CodeBricks [2], et al. These are systems in which the program explicitly constructs the generated program. Like Jumbo, all can be regarded as supporting an explicitly string-based model, up to a point. However, each imposes restrictions on what can be generated and what holes can be left, primarily to allow for efficient code generation at run time. We believe that, while undoubtedly achieving the desired goal of efficient run-time code generation, these restrictions render these systems inapplicable for a variety of very important applications; we take up this theme in the next section.

Aspect-oriented programming as embodied in AspectJ [10] is often considered a program generation system. Like those systems, it allows the programmer to divide his program into small pieces that are assembled prior to execution. However, it seems to us more helpful to think of AspectJ as a system that *uses* program generation rather than one that creates program generators. The AspectJ weaver incorporates a fixed repertoire of program generating methods, and does not give the programmer the capability to produce new program generating methods at all. As an example of this, the Proxy implementation of section 4 was inspired by the presentation in [8]; the AspectJ implementation can be found at www.cs.ubc.ca/~jan/AODPs/. It is interesting to see the program that is woven from the AspectJ specification. Although we have no reason to doubt that its behavior is just as the pattern requires, the program itself bears little resemblance to the one we produced, which is, we believe, what a programmer would have expected. This remark is not intended as a criticism of AspectJ, but simply to draw a contrast between it and the kind of program generation systems to which Jumbo ought, in our view, to be compared.

7. USES OF PROGRAM GENERATION

One of the most vexing questions about program generation is: why isn't there more of it? Our view is that the tools have not been available that are both easy enough to use and general enough to produce a wide range of examples. We do not need a "killer app" for program generation, because any particular program generator can be produced by a variety of methods, if the programmer is determined enough. Rather, we need a facility allowing program generation to be used routinely by ordinary programmers [9].

We have emphasized generality in this paper because we believe that the compelling applications for program generation are not likely to be the first-order, "value-based optimization" applications which most systems can accommodate well. Indeed, in our experience, it is difficult to find examples for which such optimizations are really compelling. There is almost always a way to preprocess the static data that does not involve program generation; being able to employ a regular compiler that can optimize code at leisure gives a huge advantage over code generated at run time under severe time constraints.

In our view, the important applications for program generation – especially, run-time program generation – will be those in which it is used to achieve modularity. The combination of *mobile computing* – in which different parts of a program are obtained from different locations at run time or load time – and *heterogeneous devices* – in which a given program can run on a variety of different hardware/software platforms – provides an environment for such applications. Heterogeneity implies the need for adaptability in the construction of the code for each device, and mobility implies that the components of the application come from diverse sources. Program generation will be needed to control not the cost of a single algorithm, but the gradual accretion of inefficiencies from the protocols through which the parts of the application communicate.

Like many advances in programming languages and software engineering, program generation will show its real value in "programming in the large." This makes it especially difficult to prove that value. The program generation community can help by implementing compelling applications – and making sure that program generation is seen to be essential to them – and by providing programmers with tools to do the job themselves.

8. PROGRAM GEN. CONSIDERED HARD

We have argued that the string-oriented model of code generation used by practitioners for compile-time program generation deserves more respect from academic researchers. In Jumbo, we have attempted to preserve the model while producing run-time program generators, by using the concept of compositional compilation. This entails certain compromises – most significantly, prohibiting the use of string destructor operations – that seem to us inescapable.

In our opinion, one rarely-spoken reason that academics disdain the string model is that it leaves nothing to do. It has so little structure that there is no apparent role for language or run-time support, and thus no employment for theoreticians. However, compositional compilation adds enough structure that some interesting problems do arise.

Jumbo is actually a rather superficial implementation of the idea of compositional compilation. Not to be misunderstood: it is a substantial, and useful, piece of software implementing the entire Java language. But it is only a first implementation. There are difficult problems remaining to be solved both to improve Jumbo and to apply the idea to other languages.

Lars Clausen's PhD dissertation [3] on the design and use of Jumbo describes a proof-of-concept implementation of one such improvement. It includes a set of source-level rewriting rules for Java that could be used to optimize the code generation process of Jumbo. This is, in effect, a partial evaluation problem, but a hard one. Clausen implemented these rules and applied them to show that performance improvements were possible. However, the application of the rules was carefully guided by hand, and results were obtained for only a small set of simple cases. No partial evaluation is incorporated in the running system. Making one work is a difficult research problem.

Another deep problem is reasoning about fragments in isolation. This is the problem that the partial evaluation-based systems avoid, but it is common to other program generation systems, as well as aspect-oriented programming. It has been suggested that JML [14] can form the basis of a reasoning system for fragments. In any case, at this stage, it is difficult even to state the requirements on fragments, much less prove them.

There is a host of broader questions concerning the applicability of our approach. In principle, any language can be implemented by a compositional compiler, but that is in part because the notion of compositionality is rather flexible. There are difficulties on both ends: some language features – notably, those whose implementation involves syntactic manipulation, like macros – and many back-end optimizations are difficult to define compositionally. (Java is an easy target because it has no preprocessor, and the translation to JVM code essentially admits no role for global optimizations.) If run-time program generation becomes widely used, it will be reasonable to consider, as a language design criterion, the ability of the language to be compiled compositionally.

9. CONCLUSIONS

Our view of program generation – this is true especially of run-time program generation – is simply that the lack of appropriate tools makes it too hard for the ordinary programmer. Some general-purpose tools for producing program generators are easy to use, but implement only subsets of languages, or impose other restrictions that programmers may find frustrating. But program generation has great potential that will only be realized when ordinary programmers can use them for *routine* tasks – which means, when programmers can write them themselves.

Jumbo is a tool that is easily learned, implements all of Java, and places few restrictions on the programmer. We have given some examples to show both its ease of use and its range of applications (more can be found in the references). We believe these are key features of any system that will achieve widespread use.

10. ACKNOWLEDGEMENTS

The Jumbo system was written by Lars Clausen, with help from Ava Jarvis. The author received help in preparing this paper from T. Baris Aktemur and Philip Morton. John Hatcliff provided very helpful feedback on an earlier version of the paper.

11. REFERENCES

[1] T.B. Aktemur, J. Jones, S. Kamin, L. Clausen. Optimizing marshalling by run time program generation. In preparation. 2004.

[2] G. Attardi, A. Cisternino, A. Kennedy. CodeBricks: code fragments as building blocks. PEPM 2003. San Diego, CA. 66-74.

[3] L. Clausen. Optimizations in Distributed Run-time Compilation. Univ. of Illinois PhD thesis. 2003.

[4] D. Cordes, M. Brown. The Literate-Programming Paradigm. IEEE Computer 24:6. June 1991. 52-61.

[5] D. Engler, W. Hsieh, M. Kaashoek. `C: A language for high-level, efficient, and machine-independent dynamic code generation. Proc. 23rd POPL. St. Petersburg Beach, Florida. January 1996. 131-144.

[6] M. Frigo, S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP). 1998. 1381-1384.

[7] E. Gamma, R. Helm. R. Johnson, J. Vlissides. Design Patterns. Addison-Wesley. Reading, Mass. 1995.

[8] J.Hannemann, G. Kiczales. Design Pattern Implementation in Java and AspectJ. Proc. 17th OOPSLA. November 2002. 161-173.

[9] S. Kamin. Routine Run-time Code Generation. Proc. 18th OOPSLA (Onward! Track). November 2003. 208-220.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. An Overview of AspectJ. Proc. ECOOP. Springer-Verlag. 2001.

[11] D.E. Knuth. Literate Programming. Computer J. 27:2. May 1984. 97-111.

[12] D.E. Knuth, Computers & Typesetting, Volume B: TeX: The Program. Addison-Wesley. Reading, Massachusetts. 1986.

[13] D.E. Knuth, Computers & Typesetting, Volume D: METAFONT: The Program. Addison-Wesley. Reading, Massachusetts. 1986.

[14] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University, TR #98-06x. November 2003.

[15] Y. Oiwa, H. Masuhara, A. Yonezawa. DynJava: Type Safe Dynamic Code Generation in Java. 3rd JSSST Workshop on Programming and Programming Languages (PPL2001). March 2001.

[16] R. Sedgewick, K. Wayne. Introduction to Computer Science. Online textbook available at www.cs.princeton.edu/introcs/home/. 2004.

[17] Y. Smaragdakis, D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. On Software Eng. And Methodology 11(2). April 2002. 215-255.

[18] W. Taha, T. Sheard. Multi-stage programming with explicit annotations. In Proc. PEPM '97. ACM SIGPLAN Notices 32: 12. New York. June 12-13, 1997. 203-217.

```
@* Fibonacci numbers.
Program to compute the first 25 Fibonacci numbers.

@* Main program.
This is a skeletal program body around which the program is constructed.

public class Fib25 {
  public static void main (String[] args) {
    final int MAX = 25;
    @<Main variable declarations@>
    @<Generate Fibonacci@>
  }}

@* Local (main) variables.
Right now, we know we need a counter to count the first 25 numbers.
@<Main variable declarations@> =
  int count;

@ Generation of Fibonacci numbers requires initial priming with two values.
@<Main variable declarations@> +=
  int fib1 = 0, fib2 = 1, newfib;

@* Generation of Fibonacci numbers.
@<Generate Fibonacci@> =
  @<Print titles and first two@>
  count = MAX-2;
  @<Loop to print remainder@>

@* Printing titles.
@<Print titles and first two@> =
  System.out.println("Fibonacci numbers");
  System.out.println(fib1);
  System.out.println(fib2);

@* Generate the remainder of the list.
@<Loop to print remainder@> =
  while (count-- != 0) {
    @<Compute new Fibonacci@>
    System.out.println(newfib);
    @<Reset last two Fibonacci values@> }

@* Computing the next Fibonacci.
Next Fibonacci number is fib_n = fib_n-1 + fib_n-2.
@<Compute new Fibonacci@> =
  newfib = fib1 + fib2;

@* Reset the last two values.
@<Reset last two Fibonacci values@> =
  fib1 = fib2;
  fib2 = newfib;
```

Figure 1: Web for Fibonacci program (after Figure 8 from [4])

```
public class Fib25 {
  public static void main (String[] args) {
    final int MAX = 25;
    int count;
    int fib1 = 0, fib2 = 1, newfib;
    System.out.println("Fibonacci numbers");
    System.out.println(fib1);
    System.out.println(fib2);
    count = MAX-2;
    while (count-- != 0) {
      newfib = fib1 + fib2;
      System.out.println(newfib);
      fib1 = fib2;
      fib2 = newfib;
    }}}
```

Figure 2: Tangled program from Figure 1

```
public class Fib1 {

  public static void main (String[] args) {
    new Fib1().genfib().generate();
  }

  Code genfib () {
  // Program to compute the first 25 Fibonacci numbers.
    return $< public class Fib25_1 {
                public static void main (String[] args) {
                  final int MAX = 25;
                  `Stmt(vardecls());
                  `Stmt(gen25fibs());
                }
              } >$;
  }

  Code vardecls () {
  // Right now, we know we need a counter to count the first 25.
    return $< int count; >$;
  }

  Code gen25fibs () {
  // Declare and initialize variables, then enter loop.
    return $< `Stmt(initial_decls())
              `Stmt(generate_fibonacci()) >$;
  }

  Code initial_decls () {
  // Generation of Fibonacci numbers requires initial priming with two values.
    return $< int fib1 = 0, fib2 = 1, newfib; >$;
  }

  Code generate_fibonacci () {
    return $< `Stmt(print_title_and_first_two());
              count = MAX-2;
              `Stmt(loop_to_print_remainder()) >$;
  }

  Code print_title_and_first_two () {
    return $< System.out.println("Fibonacci numbers");
              System.out.println(fib1);
              System.out.println(fib2); >$;
  }

  Code loop_to_print_remainder () {
  // The loop will run until count is zero.
    return $< while (count-- != 0) {
                `Stmt(compute_new_fibonacci());
                System.out.println(newfib);
                `Stmt(reset_last_two_Fibonacci_values());
            } >$;
  }

  Code compute_new_fibonacci() {
  // Next Fibonacci number is fib_n = fib_n-1 + fib_n-2.
    return $< newfib = fib1 + fib2; >$;
  }

  Code reset_last_two_Fibonacci_values() {
    return $< fib1 = fib2;
              fib2 = newfib; >$;
  }
}
```

Figure 3: Jumbo version of web in Figure 1

```
public class Fib2 {
  public static void main (String[] args) {
    new Fib2().genfib().generate();
  }

  // Compute first 25 Fibonacci numbers.
  Code genfib () {
  // The key part of this algorithm is the body of the loop, which maintains
  // the invariant: for some n >= 0, fib1 = F_n and fib2 = F_n+1.
    return context($< newfib = fib1 + fib2;
                      fib1 = fib2;
                      fib2 = newfib;
                      `Stmt(print($<fib2>$)) >$);
  }

  Code context (Code inner_loop_body) {
  // Declare local variables, then establish invariant and maintain it in loop.
  // Note that termination code is independent of the computation.
    return class_container("Fib25_2", declare_vars(), establish_invariant(),
             repeat(23, inner_loop_body));
  }

  Code class_container(String class_name, Code decls, Code pre_loop, Code loop) {
  // Class = declarations, code to establish invariant, and main loop
    return $< public class `class_name {
                public static void main (String[] args) {
                  `Stmt(decls)
                  `Stmt(pre_loop)
                  `Stmt(loop)
                }
              } >$;
  }

  Code declare_vars () {
  // These are the variables used in the loop body above
    return $< int fib1, fib2, newfib; >$;
  }

  Code establish_invariant () {
  // Set fib1 and fib2 so as to satisfy invariant
    return $< fib1 = 0;  fib2 = 1;
              `Stmt(print($<"Fibonacci Numbers">$))
              `Stmt(print($<0>$))
              `Stmt(print($<1>$)) >$;
  }

  Code repeat (int n, Code body) {
  // Repeating by a fixed amount is easy
    Name c = new Name("count");
    return $< int `c = `Int(n);
              while (`c-- != 0) `Stmt(body) >$;
  }
     }

    // Auxiliary methods
  Code print (Code subject) {
    return $< System.out.println(`Expr(subject)); >$;
  }
}
```

Figure 4: Second Jumbo version of Fibonacci web

```
class CountingProxy implements RequestInt
{
  RequestInt subject;
  int count = 0;

  public CountingProxy (RequestInt subject) {
    this.subject = subject;
  }

  public void safeRequest(String s) {
    subject.safeRequest(s);
  }

  public void regularRequest(String s) {
    subject.regularRequest(s);
  }

  public void unsafeRequest(String s) {
    count++;
    subject.unsafeRequest(s);
    System.out.println(count);
  }
}
```

Figure 5: Proxy implementation in Java

```
static Code genProxy (String proxyname, MonoList decls,
      Codefun safeFun, Codefun regularFun, Codefun unsafeFun) {
  return $<
    public class `proxyname implements RequestInt {
      RequestInt subject;

      `Field(decls);

      public `proxyname (RequestInt subject) { this.subject = subject; }

      public void safeRequest(String s) {
        `Stmt(safeFun.apply($<subject.safeRequest(s);>$));
      }

      public void regularRequest(String s) {
        `Stmt(regularFun.apply($<subject.regularRequest(s);>$));
      }

      public void unsafeRequest(String s) {
        `Stmt(unsafeFun.apply($<subject.unsafeRequest(s);>$));
      }

    } >$;
  }
```

Figure 6: Simple Proxy Generator in Jumbo

```
    static Code genProxy (String proxyname,
                          MonoList decls,
                          CodeMap methodMods) {
```

Same as previous version, but use lookup in methodMods, e.g.

```
          public void safeRequest(String s) {
            `Stmt(methodMods.get("safeRequest").apply(
                                      $<subject.safeRequest(s);>$));
          }
```

Figure 7: Second Simple Proxy Generator in Jumbo

```
public static Code genProxy( String proxyType, String interfaceName,
                             MonoList newFields, CodeMap mods){
  MonoList body = new ArrayMonoList();
  Type subjectType = new Type(interfaceName);
  body.add($Field< `Type(subjectType) subject; >$);
  body.add(newFields);
  body.add($< public `proxyClassName (`Type(subjectType) s) { subject = s; } >$);
  body.addAll(genClassBody(interfaceName, mods));

  Code c = $< public class `proxyType implements `interfaceName {
              `Body(body) } >$;
  return c;
}

static MonoList genClassBody(String interfaceName, CodeMap mods){
  MonoList body = new ArrayMonoList();
  Method[] methods = Class.forName(interfaceName).getMethods();
  for(int i=0; i<methods.length; i++)
    if(! Modifier.isFinal(methods[i].getModifiers()))
      body.add(genMethod(methods[i], mods));
  return body;
}

static Code genMethod(Method method, CodeMap mods){
  String methodName = method.getName();
  Codefun cf = mods.get(methodName);
  Class retType = method.getReturnType();
  Class[] parameters = method.getParameterTypes();
  return genMethod(methodName, cvtToType(retType), cvtToType(parameters), cf);
}

static Code genMethod(String methodName, Type retType,
                      Type[] parameterTypes, Codefun cf){
  Code delegation;
  MonoList paramList = new ArrayMonoList();
  MonoList argList = new ArrayMonoList();
  for(int i=0; i<parameterTypes.length; i++){
    String argName = "__arg"+i;
    paramList.add($Param< `Type(parameterTypes[i]) `argName >$);
    argList.add($< `argName >$);
  }

  delegation = $<subject.`Name(methodName) ( `Args(argList) ); >$;
  return $< public `Type(retType) `Name(methodName) ( `Params(paramList) ){
             `Stmt(cf.apply(delegation)) } >$;
}

static Type[] cvtToType(Class[] classes){
  Type[] types = new Type[classes.length];
  for(int i=0; i<types.length; i++)
    types[i] = cvtToType(classes[i]);
  return types;
}

static Type cvtToType(Class aClass){
  if(aClass.isPrimitive()){
    if(aClass == Boolean.TYPE)   return Type.boolean_type;
    if(aClass == Character.TYPE) return Type.char_type;
    // … seven more similar lines omitted to save space …
  }else{
    return new Type(aClass.getName());
  }
}
```

Figure 8: Generalized Proxy Generator in Jumbo