Routine Run-time Code Generation

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

ABSTRACT

Run-time code generation (RTCG) would be used routinely if application programmers had a facility with which they could easily create their own run-time code generators, because it would offer benefits both in terms of the efficiency of the code that programmers would produce and the ease of producing it. Such a facility would necessarily have the following properties: it would not require that programmers know assembly language; programmers would have full control over the generated code; the code generator would operate entirely at the binary level. In this paper, we offer arguments and examples supporting these assertions. We briefly describe Jumbo, a system we have built for producing run-time code generators for Java..

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *code generation.*

General Terms

Languages.

Keywords

Run-time code generation; Java.

1. INTRODUCTION

A code generator is a program that produces other programs. Almost all code generators fall into one of two categories:

- *Binary-level, run-time (or load-time) code generators*. The best-known examples are the just-in-time compilers for languages like Java, C#, and Self [10,28]. These programs are in everyday use, but they are few in number. They are written by "systems programmers" possessing a deep knowledge of machine-level programming, rather than application programmers.
- *Source-level, compile-time code generators*. The best-known examples here are parser- and lexer-generators. But simple, *ad hoc* compile-time code generators are very common – far more than run-time code generators – because they can be written by a programmer knowing only the standard programming language. The scripts employed in source distribution of programs that modify the program source by, for example, inserting the correct local path names for libraries, are common examples. Templates and macros are also, in effect, simple program generators. Taking all forms of compile-time code generation together, it is a very common technique.

The argument of this paper can be summarized concisely: if the benefits of these two classes of code generators could be realized in a single system, the combination would be much more powerful than the sum of the two classes separately. The widespread employment of *run-time* code generators is constrained by the difficulties of writing them; the employment of *compile-time* code generators is constrained by the fact that they can be employed only where a compatible compilation environment is known to exist. A system which could be used by ordinary programmers to create run-time code generators would open up many possibilities. This is what we call *routine run-time code generation*.

We are further advocating that run-time code generators can be obtained by changing what is meant by "object code": instead of "executable machine language," we prefer the definition "executable program generator." That is, we are suggesting that, as a routine matter, programs might be delivered in the form of code generators. More generally, *components* might be delivered in the form of parameterized code generators. Taking a page from functional programming, we need merely extend this notion to allow for code generators parameterized by other code generators (themselves possibly parameterized by yet other code generators), and the result is a completely general, yet completely binary, RTCG facility. In other words, with traditional object files as the primitive types, run-time code generators of great variety can result from employing the full domain of higher-order values.

In summary, we propose that RTCG can and should be a routine tool used by application programmers. We are mainly concerned in this paper with arguing for the "should" part of that proposition. But the argument would be idle if we had no idea about the "can" part. Accordingly, we will briefly describe Jumbo, the system we have built for the creation of run-time code generators in Java.

The paper is organized as follows: We begin by discussing what we consider to be the most essential properties of a system to support routine RTCG, why routine RTCG is desirable, and how existing approaches succeed or fail in exhibiting the properties we have listed. The heart of the paper is sections 5-10, in which we discuss the potential applications of such a system. (Section 10, on using RTCG to provide higher-level programming facilities, is much the longest of these sections.) We follow this by describing

*Partial support received from NSF CCR-9619644.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

our approach to creating such a system, and give a brief description of the Jumbo system for creating run-time code generators in Java. A conclusions section completes the paper.

2. PROPERTIES OF A ROUTINE RTCG SYSTEM

Manifestly, run-time code generation is not routine. This despite the fact that its benefits have been touted for many years. We believe this is because the process of creating run-time code generators has never been properly supported in existing languages – or, more precisely, in existing *compiled* languages. Here, we discuss some key properties we believe a system to support creation of run-time code generators should possess. In section 4, we will review some of the prior art in this area, to see where existing systems succeed or fail along these lines.

In our view, then, a system to allow for routine run-time code generation would have these properties:

Programmer control: The programmer should be able to say exactly what program is to be created. Some restrictions to enforce safety or efficiency requirements may be necessary, but they should be minimized. (Actually, we would go further: We believe that in the current state of our knowledge of how RTCG might be used, it would be premature to place *any* restrictions on programmers' uses of it. But we acknowledge that such restrictions may, at length, turn out to be useful.)

Source-level specification: If programmers are to be in control, and if the system is to find routine use, programmers must be able to speak to the system in source language. Most current systems that perform run-time code generation – just-in-time compilers are the best known examples – cannot be expressed in source code; such code generators fall outside the scope of what we are advocating. (Of course, we are not saying that those code generators are bad, only that they are inherently exotic and unlikely to be routinely created by application programmers.) On the other hand, many kinds of code generation can be expressed in source. These include, obviously, anything currently done by compile-time code generators, as well as the generators produced using partial evaluation-based "meta-programming" techniques. It also includes transformations which are, for varying reasons, done at the binary level [15,19], but whose semantics can be explained at the source level.

Run-time code generation: The central argument for *run-time* code generation is that it presents opportunities for optimization that are not available at compile time, or even at load time [16]. We will shortly make another argument for this property from the notion of *deployability*.

Binary-level implementation: We mean by this that the inputs and outputs of the code generator should be binaries rather than source code. As stated earlier, source code generation is viable only when the compilation environment is certain to be appropriate for the generated code. Compilation environments are highly complex – requiring the correct version of the compiler and supporting libraries in the correct places on the compilation platform – and, more to the point, highly variable. Run-time environments, while also complex, are comparatively standardized. Another problem with generating source code is that the expense of compilation limits the range of applicability of the generator. Yet another is that many companies do not like to distribute source code, and there is no reason to think this reluctance would not extend to machine-generated code; limiting the commercial contribution is not a good way to encourage routine code generation.

(There is an ambiguity to the term "binary." Even traditional object files do not actually contain executable code – some translation is done in the linking process. In principle, one might argue that AST's are binaries, but just have a more complex translation process; [18] says essentially this. Indeed, one could even say the same about source code! In this sense, being "binary" is a matter of degree rather than kind. In any case, the essential point at issue is the complexity and, above all, the *standardization* of the translation process. Traditional binaries have to be standardized because the machine language is fixed by the hardware; virtual machine codes have to be standardized – thought not quite as much – because of the difficulty of distributing new virtual machines to all clients. ASTs and source programs could, *in principle*, be perfectly standardized; in practice, this is unlikely ever to happen.)

The last two properties are related to *deployability*, a notion that is considered critical in the software components community [29]. Indeed, deployability – broadly speaking, the ability to obtain and employ a software artifact easily – is generally considered part of the very definition of the term "software component." Examples include applets, COM components, and traditional subroutine and class libraries. It is also generally agreed that the intervention of a compiler when employing a piece of software renders that software inherently non-deployable. (In their survey [21], Luer and van der Hoek write, "the component must be prepackaged, which typically means that it is distributed in binary form rather than source code. Pre-packaging has the benefit that it avoids having to rely on a user to configure and compile source code, which, to date, remains an error-prone process that typically requires significant technical skills.") Thus, our insistence on binary-level operation is, in part, because we want code generators to be as deployable as other software components. Deployability is also enhanced when a component can be loaded dynamically (like applets and COM components). This provides another argument for insisting on run-time code generation: there would be no point in providing a code generator that could be *loaded* at run time if it could not be *used* at run time.

If a mechanism having these properties were available for mainstream, compiled languages, it would open up numerous opportunities for producing more adaptable – and therefore more useful – programs. Consider the example of Lisp. Lisp implementations are typically interpretive, which means that, broadly speaking, the compilation and execution environments are the same, eliminating the distinction between compile-time and run-time. A compile-time code generation facility *is* a run-time code generation facility; since programs are retained in source (that is, S-expression) form, a source-based code generation system *is* a "binary"-based system. Thus, Lisp should be a good test case for our claims. And, indeed, program generation has long been a standard tool in the Lisp programmer's toolkit. (Paul Graham [8] gives a lively and forceful discussion on this point.) Consider the example of encapsulated data types. Early versions of Lisp – like other languages of its era – did not have a built-in encapsulated data type facility. In the 1970's, a great variety of such facilities were introduced into Lisp. This was done by using the macro facility. Eventually, these features became "official."

Without the experimentation made possible by the macro facility, these might have taken a very different form and taken much longer to develop.

3. WHY SHOULD RTCG BE ROUTINE?

In the following sections, we will describe some of the ways in which routine run-time code generation might help solve computing problems. Again, we emphasize that the point here is not to encourage the occasional use of RTCG for specialized purposes; neither are we advocating any particular run-time code generator or area of potential application of RTCG. Rather, we want to give the ordinary application programmer the power to create run-time code generators.

Here, we state briefly some benefits that can result from supplying the programmer with this tool. Our list is merely a suggestion of what might be possible. In the heart of this paper – sections 6-10 – we elaborate on this list. However, as with any other programming facility, we would anticipate – and this is really the heart of the argument we are offering – that, given the power, programmers will make deeper and more imaginative uses of it than we can possibly anticipate.

Run-time efficiency for first-order programs: It is well known that many computations can be sped up by preprocessing some of the data (the static, or relatively static, data) before consuming the remainder of the data (the dynamic part).

Run-time efficiency for higher-order programs: Highly modular programs lose efficiency by virtue of the boundaries between components. The entire program is known by run-time, if not load-time. The case is similar to the first-order case except that the static data consists of the late-bound components with respect to which the program is, in effect, parameterized.

Code compaction: One benefit of generating code at run-time is that some parts of the code may become unnecessary, and other parts may be subject to substantial simplification. This can be particularly important when the target platform is resource-limited.

Code adaptation: Much has been said recently about the need for software to be highly adaptable in the coming world of ubiquitous embedded devices. The scenario often invoked is when an ad hoc community of small devices receives a new "visitor" who speaks a language never before heard. Assuming there is some primitive ground on which communication can begin, the devices must learn to adapt to this new interlocutor: they must determine what it wants, what it can offer, and how to cooperate to achieve the overall goals of the community. In more vulgar terms, they must all exchange drivers and adapt them for efficient communication.

Programming language expressiveness: A constant theme – one might even say *the* constant theme – of programming language and software engineering research is "raising the level" at which programmers express their solutions; another way of saying this is that the programmer should be able to express her design, not just its implementation. Each language attempts to do this, within its own domain and under its unique constraints. Beyond the development of a new language, one method of "raising the level" is to use macros. Another method is to embed a domain-specific language, admitting, within its scope, more concise expression or more efficient execution or both. Broader efforts with the same goals include various component methodologies [21], aspect-oriented programming [17], and intentional programming [26], all of which can be regarded – to a first order of approximation – as code generation systems. If these methods could be employed at the binary level, it would make them more portable and more deployable, and accordingly more routine.

In none of these areas does RTCG provide a "turn-key" solution. The difficult problems in adaptive computing, efficient component interaction, and so on, are deeper than any implementation technique can solve. But RTCG can be an enabling technology: it can permit programmers to create ad hoc solutions to these problems without incurring large infrastructure costs or overcoming a large learning curve.

4. RELATED WORK

Code generation is not a new concept, nor is the idea of creating tools to facilitate it. Our contribution is in emphasizing the importance of combining full programmer control, source-level specification, binary-level operation, and run-time code generation. Existing systems – with a few exceptions – fail to realize this combination. Of course, they have other properties that we have not emphasized, such as high efficiency, ease of programming, or static analyzability. Still, we would claim that missing any of the four properties we require precludes routine usage.

The biggest category of related systems is those based on partial evaluation [5]. In these systems, program inputs are classified as static or dynamic. When the static data arrive, a program is generated that is expected to process the dynamic data more efficiently than the original program would have done. Partial evaluation has some decided advantages over other approaches. In particular, it is easier to use because the programmer only writes a single program; the division between program generator and generated program (the "staging" of the computation) is automatic. However, it fails to have all the properties we have enumerated. Although most partial evaluation-based systems have some form of programmer annotation to allow for greater control of the staging, the degree of programmer control is inadequate (see section 6 for an example). Indeed, the approach is inherently limited, in this way: the generated program cannot contain anything that was not, in some sense, already contained in the generating program. For example, in typed languages [30], the generated code cannot contain new type declarations; since the original program, prior to staging, must have been complete and type-correct, it is impossible for new types to have a place in the generated code. However, the generation of types is frequently part of the program generation process; we view this as failing to give the programmer sufficient power over the generated code. Furthermore, these systems seem to be inherently compile-time, or, at best, load-time. (Hence, data are "static" and "dynamic" rather than, say, "earlier" and "later.") In particular, there is no facility in any of these systems, as far as we know, for generating code more than once during execution.

`C [6,24] comes close to possessing all four properties. For example, new functions can be created and compiled at any time during execution. However, like partial evaluation systems, `C does not permit the dynamic construction of new types. Also, for efficiency reasons, some restrictions are placed on the kinds of code that can be filled into holes, again limiting the programmer's freedom.

DynJava [22] is a system that resembles Jumbo, the system we have built to implement our ideas (see section 12). It is not our purpose here to give a technical comparison of these two systems. However, DynJava does have static type-checking and, as we have noted above, this restricts the types of code programmers can generate.

Intentional programming (IP) [26] has similar goals to ours: to create highly adaptable programs. It does this by representing programs as trees and giving programmers the opportunity to create tree transformers easily, thereby creating new language features. IP, however, operates at the level of abstract syntax trees, so is not binary-level. For the reasons given earlier, we believe this creates a significant barrier to widespread adoption.

5. SPECIFYING PROGRAM GENERATORS

Throughout the paper, we will use Java as the language in which to give examples. We propose to use a very simple method of constructing code generators: specify programs as strings. In other words, we will treat Java as if it were an interpreted language. As noted above, the disadvantage of this approach is that it is strictly compile-time – that is, it can be used only on systems known to have the correct compilation environment. For now, we are not concerned with how to create a mechanism that has all four properties we require; the first two properties suffice for what we want to illustrate.

We would, however, like to make a small notational change to Java strings to make them better applicable to our purposes. First, we will augment the normal double-quote notation for string constants with a "bracketing" notation: instead of **`"I'm a string"`**, we will write **`$<I'm a string>$`**. We will also employ an "anti-quotation" mechanism, as follows: Within a quoted string, the notation **`~(`*`expression`*`)`** will be used to indicate that the result of evaluating ***`expression`*** will be a string that is to be spliced into the middle of the containing string. In short, in this extension of Java:

```
$< … ~(---) … >$  ≡  "…" + (---) + "…"
```

The inner expression (`---`) is an arbitrary string-valued expression. In particular, it may contain strings as subexpressions, and these may use the new notation and may, in turn, contain anti-quoted parts. When the anti-quoted expression is a single identifier, we will omit the parentheses. We should add that, in contrast with the double-quote syntax, our brackets allow embedded line breaks.

The idea of quote/anti-quote is of considerable antiquity [1]. The specific notation used here is similar to that used in MetaML [30]. However, there is a significant difference in its use in, say, Lisp, and its use in MetaML. In Lisp, it is a mere abbreviation. The strings (actually, S-expressions) created using anti-quotation are just strings, subject to all the string operations provided by the language. In MetaML, the values inside the anti-quotation brackets are not strings at all, but values of type "code." These values have none of the string operations, just an implicit "compile" operation applied at some point later in the computation. *There is a bright line between these two notions of quotation*. For reasons to be given later (see section 11), we wish to state clearly on which side of this line we stand. *We will never perform ordinary string operations on strings constructed using our quotation syntax.* Aside from using them to construct more strings, the only thing we will do with them is to (implicitly) apply a compile operation. We give a number of examples in this paper, the primary purpose of which is to show that a wide variety of code generators can be written under this constraint.

There are no other restrictions on what code fragments can be created. In particular, variable capture is possible. Enforcing "hygienic" uses of variables is a feature – like type-checking – that arguably makes the use of program generation safer, but we repeat our claim that, in the current state of our knowledge, such restrictions are premature. Both the creation of new types and the capture of free variables occur naturally in program generation, and we have seen no compelling evidence that the safety gained by imposing restrictions is worth the power lost. (Concerning variable hygiene, Graham [7] says "[hygienic macros] are a classic example of the dangers of deciding what programmers are allowed to want." Of course, not everyone agrees.)

6. EFFICIENCY IN FIRST-ORDER PROGRAMS

The simplest applications of RTCG exploit static, or relatively static, data for efficiency [16]. Examples include propagation of run-time constants, loop unrolling, sparse array calculations, and recursion unfolding.

Although this idea has been advocated for many years, it is still only rarely used in practice. We believe the reason for this is that programmers have not been provided with a facility having the properties listed earlier. Above all, the available methods do not allow adequate programmer control. Absent programmer input, it is extremely difficult for a compiler or run-time system to know *when*, *where,* and *how* RTCG will be useful.

Consider the problem of unrolling a loop. Efficiency may be gained in this process both by eliminating the loop control overhead and by presenting a longer straight-line code sequence to the processor, allowing for better pipeline utilization. Although some optimizing compilers will do this for some loops, it is, by and large, impossible to do at compile time, as the iteration count for the loop is rarely known that early. Thus, a run-time code generation facility is essential. In fact, the iteration count of a particular loop need not be constant, but might change slowly enough that code can be generated for the loop each time the count changes. Since the rate of change of this iteration count is normally a function of the *expected use* of the program rather than any intrinsic properties of the program, it is impossible to have the unrolling done automatically. At the very least, we need a facility whereby the programmer can indicate exactly when to generate new, unrolled code for the loop. But now another problem intrudes: the unrolling process itself may be more subtle than simply repeating the loop body multiple times. For instance, it may be necessary to unroll the loop only partially, to avoid creating an overly long code sequence. The optimal unrolling may depend upon numerous factors, and it is unrealistic to expect the system to discover it automatically. Indeed, it may be difficult for the programmer to determine it as well, but with sufficient control over the outcome, the programmer is at least empowered to experiment with different arrangements. There is no reason to think that the efficiency issues arising in this context are so much more complex than those arising in ordinary programming that programmers should not even be allowed to deal with them.

No method in current use allows for sufficient programmer control to handle the various forms of loop unrolling in any easy-to-use package. Keppel et al. [16] allowed for complete programmer control but required that the dynamically generated parts of the program be written in machine language (or an abstract intermediate language), rather than in source. Lee and Leone [20] provide minimal control over the loop unrolling process. They do not offer a solution to the problem of over-unrolling. Partial evaluation systems generally provide little control over the ultimate form of the generated code. Some partial evaluation-based systems offer enough programmer control to solve the problem of *when* to unroll a loop [9,25], but we know of none where the user can direct the unrolling process to the extent of allowing for partial unrolling.

On the other hand, we would claim that the problem is not inherently that difficult. Imagine that a programmer has available the kind of simple, compile-time code generation facility we have described above. (Most of the code shown here is from [14].) In the simplest version of the problem, the programmer has a piece of code S that is to be repeated, say, 100 times. Let us stipulate that the code is in a critical spot, that loop control is a large fraction of the execution time, and that over-unrolling is not an issue; in other words, let us assume that completely unrolling the loop would be beneficial. Repeating the code 100 times – that is, manual unrolling – has obvious disadvantages from a software engineering perspective.

In a class `Unroll`, the programmer might define a method `unroll(String, int)`, where the first argument is the loop body and the second the number of repetitions:

```
static String unroll (String s, int n) {
  if (n==0) return $< >$ ;
      else return $< `s `(unroll(s, n-1)) >$ ;
}
```

For example, the expression `Unroll.unroll($< System.out.println(10); >$, 3)` yields

```
System.out.println(10);
System.out.println(10);
System.out.println(10);
```

Of course, the body of a loop normally makes some reference to the index variable in the loop. The simplest method is to have the user supply the name of the index variable and have the unroller set it correctly for each iteration (note that the inner call to `unroll` refers to the previous version):

```
static String unroll (String s, String i, int n)
{ return $< `i = 0;
           `(unroll($< `s `i++; >$, n)) >$;
}
```

Then, `Unroll.unroll($< System.out.println(v); >$, $< v >$, 3)` would produce

```
v = 0;
System.out.println(v); v++;
System.out.println(v); v++;
System.out.println(v); v++;
```

However, for reasons that will become clear in our next example, we prefer a different solution. In this approach, the client of the unroller supplies the loop body in the form of a function that, given a loop index, creates the loop body. Thus, the client supplies a function object implementing the interface

```
interface Stmtfun {
   public String iter (String i) ;
}
```

The unroller can now create an index variable to supply to the loop body, but we prefer to have the client supply it himself, just to avoid the logical complexities of using name generation (although it is certainly possible, and sometimes necessary, to use it). The new unroller is

```
static String unroll
          (Stmtfun F, String i, int n) {
  if (n==0)
  return $<`i = 0;>$ ;
  else return $< `(unroll(F, i, n-1))
                 `(F.iter(i))
                 `i++; >$ ;
}
```

We get the effect of the previous call by defining

```
Stmtfun F = new Stmtfun () {
    public String iter (String indx) {
      return $< System.out.println(`indx); >$ ;
    }
  };
```

and then calling `Unroll.unroll(F, $<v>$, 5)`.

These are, of course, simple examples of unrolling, and within the capabilities of numerous existing technologies (most obviously, partial evaluation). However, it is well known that, especially for large loops, unrolling may actually result in slower execution; the savings in loop control are overshadowed by the cost of loading the larger code and by the pressure it places on the instruction cache. Thus, the programmer may decide that efficiency is to be gained by *partially* unrolling the loop. The partial unroller creates a loop the body of which contains several copies of the original loop body in sequence. The number of copies is referred to as the *block size*. The loop repeats this sequence often enough to make up the total number of required iterations. If the block size does not divide the iteration count evenly, some extra copies of the body are added at the end. One final detail is that there is no point in including the "outer" loop if its iteration count would be less than two.

Here is our version of the partial unroller:

```
public static String unroll_part (
    String i, String init, int incr,
    int iterations, Stmtfun F, int BlockSize) {
  int loops = iterations/BlockSize,
      leftover = iterations%BlockSize;
  if (loops < 2)
    return unroll(i, init, incr, iterations, F);
  else
    return
  $< for (`i = `init;
          `i < `init+(loops*BlockSize*incr);) {
       `(unroll(i, i, incr, BlockSize, F))
     }
     `(unroll(i, i, incr, leftover, F))
  >$ ;
}
```

Note that in this form of unrolling it is not possible to know the exact initial value of the inner loop's index variable, since that

value changes at each iteration. Thus, the partial unroller uses a more complicated version of the full unroller:

```
public static String unroll (
     String i, String init, int incr,
     int iterations, Stmtfun F) {
  String C = $< >$ ;
  for (int x=0; x<iterations; x++)
    C = $< C
           F.iter(`init+(x*incr)) >$ ;
  C = $< C
         `i = `init+(iterations*incr); >$ ;
  return C;
}
```

Note, by the way, that in this case the function object F is not applied simply to an iteration variable, but rather to an expression whose value is equal to the value the iteration variable would have had in the normal loop. (This assumes, of course, that there is no assignment to the iteration variable within the loop.) This is why we made the decision earlier on that the loop body should be a function.

To complete this example, the call `Unroll.unroll_part($<i>$, $<0>$, 1, 500, F, 6);` yields

```
for (i=0; i<0+498;) {
  System.out.println(i+0);
  System.out.println(i+1);
  System.out.println(i+2);
  System.out.println(i+3);
  System.out.println(i+4);
  System.out.println(i+5);
  i = i+6;
}
System.out.println(i+0);
System.out.println(i+1);
i = i+2;
```

One can argue about whether this type of coding is overly complex or error-prone; it does not seem to us to exceed some threshold that is never approached in ordinary programming.

One can also argue about whether a truly *general-purpose* loop unroller can ever be constructed. In fact, we would say that it is unlikely. But that is all the more reason why the program generation capability should be put in the hands of ordinary programmers, so that they can deal with the complications (*and* simplifications) that arise in their particular application.

7. EFFICIENCY IN HIGHER-ORDER PROGRAMS

Higher-level abstractions usually introduce some run-time cost. Implementing the abstraction – that is, maintaining the illusion for the application programmer that the underlying system has properties that it does not, at its most primitive level, have – has an associated run-time cost. Since these abstractions are built on top of yet other abstractions, the inefficiencies multiply. In most cases, this is not a critical problem, but its hidden effect is to limit the kinds of abstractions that the system designer considers building and the programmer considers using. There are numerous examples:

Module boundaries. The effect of virtually all higher-level module systems is, in part, to increase the number of function calls made by the running system. The clearest example is class libraries, where even access to a field in an object requires a function call. Programmers and language designers engage in numerous machinations – often violating information-hiding principles – to eliminate this cost.

System layer boundaries. These are a notorious source of inefficiency in operating systems and middleware [CHL+98]. Programmers cope by, in effect, ignoring the system layer structure when possible; for example, efficient network protocol implementations integrate functions across layers [3] rather than following the layers as designed.

Virtual machines. This is one abstraction for which RTCG is in routine use, under the name "just-in-time compilation" [10,28].

Higher-order functions. Functional programming languages have often led the way in providing abstractions for programmers; the higher-order function is the typical abstraction here. Other languages have adopted the concept of higher-order functions; examples are the increasingly common use of function objects and callbacks in object-oriented languages, and such features as iterators (which provide a kind of "map" operation in imperative languages). Again, increased function calling imposes an efficiency penalty.

Polymorphic functions. This is another example of a useful abstraction in higher-order languages. It reduces the programmer's burden by allowing a single copy of a function or class to be used in differing circumstances. In most languages, it is implemented by "boxing" all values, imposing an otherwise unnecessary cost on computations that involve primitive values.

Memory management. A crucial implementation technology pioneered in higher-level languages (such as Lisp), automatic memory management has now reached the mainstream. It comes with a high run-time cost, which extensive research has significantly reduced but not eliminated.

For most of these cases, the principal cost of the abstraction is in the form of extra function calls. For virtual machines, the problem is sufficiently constrained that significant benefits are obtained from *automatic* RTCG. However, that does not always work. For example, consider *polymorphic collections* – that is, collections of `Object` instances – in Java. These entail what may be a substantial cost in casting (boxing and unboxing) primitive values.[1] The run-time system cannot know that a certain collection is bound to contain, say, integers for its entire life, so it cannot optimize that collection. The programmer's only recourse is to write his own, monomorphic, collection class.

In C++ this cost is avoided by using templates, a compile-time mechanism. Here we show how to accomplish the same effect using strings (see also [14]). Note that in generating a new collection type in Java, one must create two classes: the collection class itself and a class of iterators for that class. Here we present a generator for monomorphic vectors. The method `String makeVectorName (String)` produces the name of the new vector class, and `String makeIteratorName (String)` produces the name of the iterator class for that new vector class.

```
String vname = makeVectorName(type);
```

[1] Parametric polymorphism eliminates the explicit casts, but not the boxing and unboxing.

```
String itname = makeIteratorName(type);

String vectorClassDefs =
   $<public class `vname { // vector class
       `elttype[] elements;
       int numelements;

       public `vname() { // constructor
         elements = new `elttype[10];
       }

       public void add(`elttype o) { ... }
         ...
    }

    public class `itname { // iterator class
        ...
    }
  >$;
```

Some additional, but routine, code is needed to insure that no particular instance of this generic code is generated more than once. This is contained in the method `String newVector (String);` the latter returns the code needed to create a vector of the given type. Use of this generator is not much different from the use of templates in C++; this code is assumed to appear within a larger quoted fragment:

```
`(vector($<int>$)) v = `(newVector($<int>$));

for (int i = 0; i < vlen; i++) v.add(i);

int sum1 = 0;
for (`(iterator($<int>$)) i =
          v.iterator(); i.hasNext(); ) {
   sum1 += i.next();
}
```

This provides no obvious advantages over templates (aside from the fact that Java does not have templates). We believe it *would be* advantageous if the collection-generating code could be provided in binary and could generate binaries. It would make the facility both simpler and less expensive to use – as simple and inexpensive as using the current APIs.[2]

8. CODE COMPACTION

As computers become ubiquitous, an old concern is reasserting itself: code size. Anticipated memory sizes, for all but the tiniest devices, are much larger than the memories into which the original "hero programmers" squeezed their code. Nonetheless, relative to the kind of functionality they are expected to support, these new devices are very small. There are still hero programmers, of course, but they are in short supply. The question is how to make programming small devices much more convenient. Ideally, code to run in embedded devices should be, to the greatest extent possible, obtained directly from a single code base that covers larger devices as well. (Here, we are laying aside real-time issues that are common in embedded devices and considering only the effect of miniaturization.)

Consider just one example: the problem of "feature loading." Programs often have a multiplicity of features from which a client can choose; the choice depends upon the client's needs and budget. The result of allowing this choice is that the programmer must be capable of producing, easily, any one of countless configurations of a single program. Many companies do just this, often with the help of sophisticated compile time tools written in-house. But what if the company wanted to distribute the entire program as an adaptable component? With current technologies, they could distribute the largest, most full-featured, version of the program, with versions for every platform (a really fat binary), or they could distribute the code itself, along with the entire compilation environment. Either option has obvious and severe disadvantages. Using RTCG, they would be able to distribute code capable of producing the right version of the program for any client.

9. ADAPTIVE PROGRAMS

Many writers have predicted a future in which numerous devices cooperate to serve humans. These devices are of varying size, levels of intelligence, reliability, and mobility; moreover, they may or may not be familiar with, or even trusted by, one another; yet they must cooperate. This is the general aspect of those scenarios that fall under the heading of ubiquitous computing.

All the ways in which RTCG might be used come into play when considering programs that need to survive in a dynamic environment with constantly changing interlocutors. Adaptive software is software that can respond gracefully to dramatic changes in the computational environment. All software is adaptive to some extent; the difference is the kind and quantity of environmental changes the software can handle, and the resource constraints under which the software labors.

A simple version of adaptation is adjustment to the target architecture and operating system, which is frequently accomplished in C/C++ programs using long sequences of "ifdef's." Obviously, this could be accomplished using quote/anti-quote syntax just as easily:

```
$< class GenericCode {
      ...
      ... `((current_os == "Linux")
           ? $<Linux  code>$
           : (current_os == "Windows XP")
           ? $<Windows XP code>$
           :  ...
           : $< … default or error code … >$ ...
       ...
   } >$
```

Equally obviously, this method is much more powerful than "ifdefs" in that the code associated with any particular configuration can be obtained by any means available in the language, including calculating it from a variety of static parameters, or obtaining it from a website. It also provides the opportunity to give more structure to these configuration calculations by, for example, creating classes corresponding to

[2] Frankly, it is not clear to us why a preprocessor is needed in languages like C and C++ at all. The facility used here is nearly as convenient and much more powerful. We suspect the perceived need arises from the lack of a standardized, easy-to-use string facility in those languages. The Java designers evidently felt that a preprocessor was simply unnecessary, and so provided neither it nor a quotation facility like ours. The inclusion of a preprocessor in C#, where a standard string type exists, is a mystery.

each platform. We might end up with something more concise and more extensible:

```
interface MachineType {
  String getPlatformSpecificCode () ;
  String getOtherPlatformSpecificCode () ;
      ...
}

MachineType currentPlatform =
                    getCurrentPlatform();

$< class GenericCode {
     ...
     ... `(currentPlatform.
              getPlaftormSpecificCode()) ...
     ...
   } >$
```

10. LANGUAGE EXPRESSIVENESS

RTCG is usually regarded as a low-level implementation technique. It is transparent to the programmer in the sense that it does not alter the programming language. We believe that this view of RTCG is far too strained. Indeed, we believe that the potential to provide high-level abstractions not previously available to programmers is the most exciting – as well as the least explored – area of application of RTCG.

Our evidence for this contention comes from experience with other systems. Consider the use of macros, which are a (more or less general, depending upon the language) form of compile-time program generator. Macros serve both to improve efficiency *and* increase expressiveness of the underlying language. The latter advantage has been realized most notably in Lisp systems, where macros have been used extensively for years. The template facility in C++ has also been used widely to raise the level of abstraction of programming, while at the same time – and this point has been emphasized by Bjarne Stroustrup frequently in public comments – offering the efficiency of non-generic code.

Facilities that increase programming ease are – by definition, one might say – static facilities. That is, they are known to the programmer and employed at compile time. Why then would we be interested in RTCG in this context? We return to the notion of deployability. We can make an analogy with ordinary procedures. They are present at compile-time, it is true, but not as source code. If they did have to be available as source code, this would severely constrain their use. (C++ templates offer an example [27].) Thus, even if we allow that the use of these features is inherently compile-time, we would still claim that having them in binary rather than source form would make a great difference: they would be easier to deploy and less subject to piracy, and therefore more likely to be widely developed and distributed.

We offer two examples: an implementation of the programming idiom of "state machines" and an implementation of a domain-specific language.

Programming idioms. In functional and object-oriented languages, idioms like "divide and conquer" can be programmed directly. But there is a cost in efficiency. As normally conceived, such idioms represent a way to write programs. From this point of view, the ability to express the general idea of the idiom is only a partial solution. Using run-time code generation, the precise program implied by the idiom can be expressed; this is a direct implementation of the idiom.

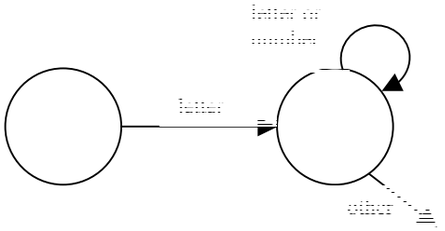
An example is the implementation of state machines. The ability to express the state transitions and actions – say, as function objects – is quite different from the ability to produce a program in which, following the standard idiom, states are program labels and actions are statements. The efficiency with which the idiom can be realized is one of its central attractions.

We present part of a simple implementation of finite-state machines. In this implementation, a finite-state machine consists of an array of states (numbered from zero), each containing an array of transitions; each transition consists of a *predicate* to test whether that transition should be taken, an *integer* giving the target state of that transition, and an *action* to be taken when that transition occurs. Predicates and actions are pieces of parameterized code; specifically, a predicate is a function from a string (an expression denoting the input character) to a string (the condition to be tested), and an action is a function from an integer (the target state) and a string (the input character) to a statement. Rendering all of this into Java is a bit cumbersome:

```
interface Predicate {
  String pred (String inputcharvar);
}

interface Action {
  String action (int nextstate,
                 String inputcharvar);
}

public class Transition {
  Predicate pred;
  int nextstate;
  Action act;

  Transition (Predicate p, int s, Action a) {
    pred = p;  nextstate = s; act = a;
  }
}

public class State {
  Transition[] transfun;

  State (Transition[] tf) { transfun = tf; }
}

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

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

  String genFSMCode () { ... }
}
```

We have omitted the definition of `genFSMCode()`, the code-generating function itself, to save space, but we will show an example of its output shortly.

With these definitions, we can define this finite-state machine:



with this code (**mkLetterPred** and **mkAlphanumPred** generate the expressions to test for letters and alphanumeric characters, respectively):

```
Transition[] s0 =
  {new Transition(
      new Predicate () {
        public String pred (String ch) {
          return mkLetterPred(ch);
        }
      },
      1,
      new Action () {
        public String action(int s, String ch) {
          return $<addToBuffer(`ch);>$ ;
      }
     })
  };
State st0 = new State(s0);

Transition[] s1 =
  {new Transition(
      new Predicate () {
        public String pred (String ch) {
         return mkAlphanumPred(ch);
       }
      },
      1,
     new Action () {
       public String action(int s, String ch) {
         return $<addToBuffer(`ch);>$ ;
       }
      }),
    new Transition(
      new Predicate () {
        public String pred (String ch) {
          return $<true>$ ;
       }
      },
      2,
     new Action () {
       public String action(int s, String ch) {
         return $<emitbuffer();>$ ;
       }
     })
  };
State st1 = new State(s1);
State[] Mstates = {st0, st1};
FSM M = new FSM($<Ident>$, Mstates);
```

Because this specification is rather cluttered, we have underlined the semantically meaningful parts; the rest is pure boiler-plate required by Java for defining function objects and initializing arrays. (In earlier work [12,13], we advocated the use of functional languages to define program generators for Java, and the results were, unsurprisingly, notationally neater.)

When the finite-state machine `M` defined in the last line gets the message `genFSMCode`, it generates this code:

```
class Ident {
  static void runFSM (InputSource in) {
  int theState = 0;
    while (true) {
      if (in.empty()) return;
      char ch = in.next();
      switch (theState) {
   case 0: if (('a' <= ch && 'z' >= ch)
              || ('A' <= ch' && Z' >= ch)) {
            addToBuffer(ch);
            theState = 1;
          }
          else ;
          break;
   case 1: if (('0' <= ch'9' >= ch)
               || ('a' <= ch'z' >= ch)
               || ('A' <= ch'Z' >= ch)) {
            addToBuffer(ch);
            theState = 1;
          }
          else if (true) {
            emitbuffer();
            theState = 2;
          }
          else ;
          break;
   default: return;
   }
  }
  return;
 }
}
```

Domain-specific languages. DSL's can be thought of as specialized compilers for a subset of a given base language. For example, an array-processing language might be built on top of a general-purpose language by defining special data types and operators. Given the understanding that only these prescribed features will be used, more efficient object code could be produced. This efficient code cannot be produced by the regular compiler, because the latter is not privy to this understanding; therefore, a separate program generator is needed.

We present as an example the "message specification language" [2,23]. This domain comes from a military application in which electronic message formats, described informally in terms of bit fields and their allowable values, are to be translated to functions to decode and encode such messages. Our presentation here is based on an earlier code generator written in ML [11]; we refer the reader to that paper for a more expansive explanation. We give only the generator for the "decoding" function.

The goal of this code generator is to take descriptions like this one:

Field Name	Size	Range
Course	3	001-360
Separator 1	/	
Speed	4	0000-5110
Separator 2	/	
Time (group)		
	2	00-23
	2	00-59

and turn them into code to read messages. If the input is in a byte array **A** indexed by **inptr** (counting in bits) and the output is to go into a record called **store**, this should be translated to code like this (assuming, for simplicity, some library operations like **outOfRange** and **getint**):

```
if (outOfRange(getint(A[inptr/8], 3), 0, 360))
{ abort(); }
else {
  store.course = getint(A[inptr/8], 3);
  inptr = inptr - (inptr%8) + (8*3);
}
if (A[inptr/8]!="/") { abort(); }
else {
  inptr = inptr - (inptr%8) + (8*1);
}
if (outOfRange(getint(A[inptr/8], 4), 0, 5110))
{ abort(); }
else {
  store.speed = getint(A[inptr/8], 4);
  inptr = inptr - (inptr%8) + (8*4);
}
if (A[inptr/8]!="/") { abort(); }
else {
  inptr = inptr - (inptr%8) + (8*1);
}
if (outOfRange(getint(A[inptr/8], 2), 0, 23))
{ abort(); }
else {
  store.hour = getint(A[inptr/8], 2);
  inptr = inptr - (inptr%8) + (8*2);
}
if (outOfRange(getint(A[inptr/8], 2), 0, 59))
{ abort(); }
else {
  store.minute = getint(A[inptr/8], 2);
  inptr = inptr - (inptr%8) + (8*2);
}
```

As in [2], we do not attempt to map directly from the tabular format, but instead embed operations in Java that provide a facsimile of the table. In our implementation, this code is produced as the output of the following Java code:

```
Message course =
  MessageOps.infield($<course>$,
         MessageOps.asc2int(3, 0, 360));
Message slash = MessageOps.delim("/");
Message speed =
  MessageOps.infield($<speed>$,
         MessageOps.asc2int(4, 0, 5110));
Message time =
   MessageOps.seq(
     MessageOps.infield($<hour>$,
       MessageOps.asc2int(2, 0, 23)),
     MessageOps.infield($<minute>$,
       MessageOps.asc2int(2, 0, 59)));

Message fullmsg =
  MessageOps.seq(course,
  MessageOps.seq(slash,
  MessageOps.seq(speed,
  MessageOps.seq(slash, time))));

Bitsource b = new Bitsource($<A>$, $<inptr>$);
Recordfield r = new Recordfield ($<store>$);
String msg = fullmsg.genmsg(b, r,
                            $<abort();>$);
```

The string **msg** contains the code shown above (modulo grooming). We now show how to implement the operations in class **MessageOps**.

The central type here is **Message**, which is an interface type for function objects:

```
public interface Message {
  String genmsg (Bitsource bs,
                 Recordfield r,
                 String stmt);
}
```

A **Message** is a function that takes (an expression denoting) a location in the bit source **bs** and returns code to write appropriate values into the record **r**, invoking **stmt** in case of an error. **MessageOps** defines a collection of static methods on messages: **asc2int** takes a certain number of bytes out of **bs**, makes sure they represent a number in ASCII that falls in the proper range, and puts them into **r**; **delim** checks for a delimiter in the input stream and skips over it; **infield** is a message transformer, taking a message and transforming it into a message that is nearly identical except that it places its results into a given subrecord of **r**; finally, **seq** performs two message extractions in sequence. Here are their definitions (**If** is an auxiliary method):

```
static Message delim (final String exp) {
  return new Message () {
    public String genmsg(Bitsource bs,
              Recordfield r, String stmt) {
      return If($< `(bs.getByte()) != `exp >$,
                stmt, bs.advanceByte()) ;
    }
  };
}

static Message infield (final String fname,
                        final Message m) {
  return new Message () {
    public String genmsg(Bitsource bs,
                 Recordfield r, String stmt) {
      return m.genmsg(bs, r.subfield(fname),
                      stmt);
    }
  };
}

static Message asc2int (final int width,
           final int lo, final int hi) {
  return new Message () {
    public String genmsg(Bitsource bs,
                 Recordfield r, String stmt) {
      return
        If( $< outOfRange(getint(`(bs.getByte()),
                          width), `lo, `hi) >$,
              stmt,
              $< `(r.deref()) = getint(
                       `(bs.getByte()), width);
                 `(bs.advanceNBytes(width));
              >$ );
    }
  };
}

static Message seq (final Message m1,
                    final Message m2) {
  return new Message () {
```

```
  public String genmsg(Bitsource bs,
              Recordfield r, String stmt) {
    return $< `(m1.genmsg(bs, r,
                          $< abort(); >$))
              `(m2.genmsg(bs, r, stmt)) >$;
    }
  };
}

static String If (String c, String t,
                  String f) {
  return $< if (`c) `t else `f >$;
}
```

This kind of coding is familiar to functional programmers, though perhaps tricky for those not used to it. We remind the reader that function objects are used here to provide the required operations without violating our rule: strings representing programs are never subjected to any string operations except inclusion in larger strings via anti-quotation.

11. WRITING RUN-TIME CODE GENERATORS

The method we have used for giving examples in this paper can be used to produce run-time code generators easily. After producing the desired string, one has simply to invoke the compiler on the execution platform and load the compiled code. Why, in practice, is this so rarely done?

Programmers might blanch at the cost of invoking a compiler while a program is executing, and certainly this is an important consideration. Still, there are many long-running programs for which the cost of compilation would almost certainly be paid off with interest. The idea of producing such a run-time code generator is not so much dismissed on efficiency grounds as, by and large, never even considered. In our view, the underlying concern is *portability*: for compiled languages, the simple method described above requires that the target machine provide the appropriate compilation environment – the correct version of the compiler and libraries, all in the expected places. In fact, *most machines don't even have compilers*.[3] Programmers are only willing to use facilities that are present on almost all machines, or, lacking that, are portable and simple to install. Compilers don't match either description.

Which brings us back to a point we made at the start of this paper: routine code generation can be achieved by viewing "code" as a primitive type and employing the full range of higher-order types constructed from it. We claim that the use of such higher-order code values is essential to realizing the promise of RTCG. It permits us to make highly adaptable programs without dealing in source code explicitly, or invoking a compiler.

The first remark we wish to make is an obvious one: Code generators are functions, meaning that they need arguments before they can produce code. Less widely appreciated is that those arguments may themselves be code generators, which need their own arguments. If *Code* is the type of primitive code generators, a user might be asked to provide a function of type, say, $int \rightarrow Code$, to the run-time code generator, which will in turn produce some code. The run-time code generator therefore has the type $(int \rightarrow Code) \rightarrow Code$. We wish to convince the reader that this, and more complex, types are useful for the goal of making RTCG routine.

There are two halves of the argument we need to make: First, that higher-order functions arise naturally when RTCG is used for the kinds of applications we have listed. We have been making this argument implicitly throughout the paper; we make it more explicit below. Second, and more subtly, we need to show that it is worthwhile – even necessary – to make the use of higher-order functions in this context *explicit*.

For the first prong of the argument, we needn't do more than look at some of our examples. Starting with the first, and simplest, example – loop unrolling – we see immediately that the loop unroller takes a statement-returning function as one of its arguments; specifically, its type (treating function objects as actual functions) is

$$(Code \rightarrow Code) \times Code \times int \rightarrow Code$$

Indeed, every one of the examples in this paper includes higher-order functions over *Code*.

To understand the second point – why higher-order functions have to be used explicitly – consider the alternative: In traditional macro systems like Lisp, code generation is accomplished by the manipulation of explicit program representations. What might be regarded in the abstract as higher-order operations on code (as discussed above) are encoded as first-order functions over program representations. For example, when constructing a loop body for the loop unrolling code generator, instead of using a function taking an identifier to a statement (that is, taking the loop index variable to the loop body), we could use a function `String subst(String indexvar, String dummyvar, String loopbody)` that substitutes `indexvar` for occurrences of `dummyvar` in `loopbody`.

Thus, a traditional source-level code generation system is based upon a concrete (source code or abstract syntax tree) representation of programs. When a complete program has been constructed, a compiler is applied to produce executable code; until then, the concrete representation is open to whatever manipulations are permissible on such data. Our system is also based on a concrete representation, but one that is restricted in the operations that can be performed on it: only string concatenation is permitted. To put it differently, holes can be filled in, but existing code fragments cannot be modified.

The choice we are discussing, then, is nothing other than the choice between using an abstract, encapsulated value (a *Code*-producing function) and using a particular concrete representation of that value (a program). The trade-off is, in large measure, the familiar one: The concrete representation is more intuitive and more flexible. At the same time, it is more dangerous to use, both because the "invariants" can be violated and because the representation is likely to change over time, invalidating programs that depend upon it (even as its abstract meaning remains the same). When the values in question are programs, additional concerns arise: The concrete representation is, in effect, source code, so the technique cannot be used without revealing source code and requiring the presence of a compiler on the run-time platform. Code-producing functions are a lot like concrete representations of code, except that they can only do one thing:

[3] Academic computer scientists may forget this, since they rarely see such machines in their own work.

generate code when supplied with appropriate arguments. This reduction in flexibility is just the same as what happens when a data representation is hidden in a class. In this case, one benefit is that it allows the supplier to provide machine code (namely, the code for the generator that will produce the desired machine code for execution) rather than a concrete program representation. Another, more fundamental, benefit is that it allows for code-producing functions to be optimized in ways that the concrete program representation cannot be. If a particular string (or AST) may be altered by using destructor operations – as any string may, in principle, be – then it is not possible to compile it ahead of time. The commitment to leave each particular string alone until it gets compiled allows the string to be partially compiled statically, thereby optimizing the final, run-time code generation process.

In short, our view is that the best way to promote a routine RTCG facility is not to pass source code (or abstract syntax trees) among computers, but rather to pass parameterized code generators (possibly parameterized on other code generators).

Before ending this discussion, we wish to make one point of technical clarification. In the above, we have used the type *Code* as if it were synonymous with the type of machine language programs (or, in the Java context, virtual machine programs). For the idea we are espousing to work, *Code* cannot be simply machine language. It must be a richer type, but one from which machine language can be (efficiently) obtained. Details can be found in the references [4,14].

12. JUMBO

We have developed a compiler for Java, called Jumbo [14], which incorporates a quotation mechanism like the one employed in this paper. The compiler works in conjunction with a run-time compilation API that, like any Java API, is portable and easy to install. The Jumbo compiler produces code that can be run on any JVM system and, given the Jumbo API, can perform run-time code generation. The client machine need not have a compiler installed; as with any Java program, the API will be loaded when it is first used.

Jumbo has all four properties we have insisted upon. As we have seen in the examples of this paper, the programmer has complete control over the construction of programs. The programmer does not actually manipulate strings; the Java type system enforces this, as the quoted program fragments are assigned type `Code` rather than `String`. However, the specification of code generators, as we have seen, feels very much like manipulating strings; the crucial difference is that ordinary string operations are not available. (In Jumbo, quoted program fragments must be parsed, which raises some issues that do not come up when manipulating strings; for that reason, the examples given in this paper will not work in Jumbo as is, but must be modified slightly.)

Jumbo is a complete implementation of Java. It can be used as an alternative to `javac` for ordinary Java programs (and produces virtually identical output), and it can be used to produce run-time code generators. Aside from some restrictions imposed by the parser, arbitrary Java code can be enclosed in quotes for run-time execution.

Jumbo is described in [14] and can be obtained at shasta.cs.uiuc.edu/Jumbo. The Jumbo versions of the examples from this paper are also provided there.

13. CONCLUSIONS

Run-time code generation has many potential applications. Despite this, for compiled languages, the use of run-time code generation has never been popular. If the production of run-time code generators were easier, clever programmers would likely exploit that potential and, indeed, find many applications not yet conceived of.

Furthermore, the production of run-time code generators is not that difficult. Generating a program and invoking a compiler at run-time are well within the capabilities of average programmers. Still, the practice has not been adopted.

We believe that the difficulty is not technical, but, so to say, bureaucratic. As long as "code" is considered to be synonymous with "concrete program representation," RTCG will entail difficulties associated with the distribution of source code and the vagaries of compilers (specifically, their tendency not to exist when or where they are needed). We propose a new paradigm, in which *Code* is a first-class value and run-time code generators are regarded as higher-order values involving *Code*. No compiler is needed at run time, and no source code is ever created at run-time, much less revealed to the client.

14. ACKNOWLEDGMENTS

Many of these ideas were formulated during the development of the Jumbo system and its predecessor; Lars Clausen was the chief programmer of both systems, with Miranda Callahan and Ava Jarvis making important contributions as well. Ralph Johnson made helpful comments on a previous draft of this paper. Miranda Callahan made many useful comments. I would also like to thank the members of the program committee – particularly Geoff Cohen, the paper's "shepherd" – for their extremely useful feedback.

15. REFERENCES

[1] A. Bawden. Quasiquotation in Lisp. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-99). San Antonio, Texas. January 1999. 22-23.

[2] J. Bell, F. Bellegarde, J. Hook. R.B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D.P. Oliva, T. Sheard, L. Tong, L. Walton, T. Zhou. Software design for reliability and reuse: A proof-of-concept demonstration. TRI-Ada '94. 1994.

[3] D.D Clark, D.L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. ACM SIGCOMM. 1990. 200-208.

[4] L. Clausen. Optimizations in Distributed Run-time Compilation. Univ. of Illinois PhD thesis. 2003 (forthcoming).

[5] O. Danvy, R. Glück, P. Thiemann, (eds.). Partial Evaluation. Lecture Notes in Computer Science 1110. Springer-Verlag. Heidelberg. 1996.

[6] D. Engler, W. Hsieh, M. Kaashoek. `C: A language for high-level, efficient, and machine-independent dynamic code

generation. In 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96). St. Petersburg Beach, Florida. January 1996. 131-144.

[7] P. Graham. Being Popular. On-line article at www.paulgraham.com/popular.html. May 1991.

[8] P. Graham. Revenge of the Nerds. Intl. ICAD Users Group Annual Conference. Boston. May 2002. Expanded version at www.paulgraham.com/icad.html.

[9] B. Grant, M. Mock. M. Philipose, C. Chambers, S.J. Eggers. Annotation-directed run-time specialization in C. In Proc. Conf. on Partial Evaluation and Program Manipulation (PEPM). 1997. 163-178.

[10] U. Holzle. Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming. Stanford Univ. CS Dept. Tech. Rpt. STAN-CS-TR-94-1520. 1994.

[11] S. Kamin. Standard ML as a meta-programming language. Univ. of Illinois Computer Science Dept. September, 1996. Available at www-faculty.cs.uiuc.edu/~kamin/pubs/.

[12] S. Kamin, M. Callahan, L.R. Clausen. Lightweight and generative components I: Source-level components. In Proc. First International Symposium on Generative and Component-Based Software Engineering (GCSE'99), September 28-30, 1999. 49-64.

[13] S. Kamin, M. Callahan, L.R. Clausen. Lightweight and generative components II: Binary-level components. In International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2000). Montreal, Canada. September 20, 2000. Lecture Notes in Computer Science 1924. Springer. 2000. 28-50.

[14] S. Kamin, L.R. Clausen, A. Jarvis. Jumbo: Run-time code generation for Java, and its applications. Conf. on Code Generation and Optimization (CGO). San Francisco. March 2003. 48-58.

[15] R. Keller, U. Holzle. Binary Component Adaptation. UC Santa Barbara Dept. of Computer Science. Tech. Rpt. TRCS97-20. Dec. 1997.

[16] D. Keppel, S. J. Eggers, R. R. Henry. A Case for Runtime Code Generation. Univ. of Washington Dept. of Computer Science and Engineering Tech. Rpt. 91-11-04. November 1991.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. Proc. European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[18] T. Kistler, M. Franz. A Tree-Based Alternative to Java Byte-Codes. International Journal of Parallel Programming 27:1. February 1999. 21-34.

[19] G. Kniesel, P. Costanza, M. Austermann. JMangler – A framework for load-time transformation of Java class files. Proc. IEEE International Workshop on Source Code Analysis and Manipulation. IEEE Computer Society Press. 2001.

[20] P. Lee, M. Leone. Optimizing ML with Run-Time Code Generation. SIGPLAN Conference on Programming Language Design and Implementation. 1996. 137-148

[21] C. Luer, A. van der Hoek. Composition environments for deployable software components. UC Irvine Dept. of Information and Computer Science Tech. Rpt. 02-18. April 2002.

[22] 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.

[23] C. Plinta, K. Lee, M. Rissman. A model solution for C^3I message translation and validation. Software Engineering Inst. Carnegie-Mellon Univ. Tech. Rpt. CMU/SEI-89-TR-12. December 1989.

[24] M. Poletto, W.C. Hsieh, D.R. Engler, D. R., M.F. Kaashoek. `C and tcc: a Language and Compiler for Dynamic Code Generation. Transactions on Programming Languages and Systems 21:2. March 1999. 324-369.

[25] U. Schultz, J. L. Lawall, C. Consel, G. Muller. Towards Automatic Specialization of Java Programs. Lecture Notes in Computer Science 1628. Springer. 1999.

[26] C. Simonyi. The death of computer languages, the birth of Intentional Programming. Microsoft Research Tech. Rpt. MSR-TR-95-52. 1995.

[27] B. Stroustrup. Separate compilation must stay! AT&T Tech. Rpt. 1996.

[28] Sun Microsystems Incorporated. The Java hotspot performance engine architecture: A white paper about Sun's second generation performance technology. Technical report. April 1999.

[29] C. Szyperski. *Component Software.* ACM. New York. 1997.

[30] W. Taha, T. Sheard. Multi-stage programming with explicit annotations. In Proc. ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). ACM SIGPLAN Notices 32: 12. New York. June 12-13, 1997. 203-217.