Lightweight and Generative Components II: Binary-level Components

Sam Kamin*, Miranda Callahan, Lars Clausen

Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{s-kamin,lrclause,mcallaha}@uiuc.edu

Abstract. Most software component technologies fail to account for *lightweight* components (those for which a function call is too inefficient or semantically inappropriate) or *generative* components (those in which the component embodies a *method* of constructing code rather than actual code). Macro-based systems such as the C++ Standard Template Library are exceptions. They, however, have the disadvantage that components must be delivered largely in source form. In this paper, we present a component technology in which lightweight and generative components can be delivered in binary form. The technology is conceptually simple and is easily implemented with existing programming languages. Our basic idea was explained in part I of this paper: By giving a *compositional* semantics for a source language in a domain of meanings *Code*, components can be *written* in the form of macros, but *communicated* in terms of meanings. In the companion paper, we showed how higher-order values over *Code* can be used to write lightweight, generative components. There, we took *Code* to be `string`, so our components amounted to higher-order macros. In this paper, we define *Code* more abstractly, allowing components to be delivered in a form that does not resemble syntax, yet allows for them to be loaded dynamically and execute efficiently.

1 Introduction

The ideal software component technology would automate the use of common programming idioms. It would admit both lightweight components — those for which a function call is too inefficient or semantically inappropriate — and *generative* components — those in which the component embodies a *method* of constructing code rather than actual code. At the same time, its use would be as efficient and non-bureaucratic as subroutine libraries.

Perhaps the closest approach to this ideal is the C++ standard template library (STL), which "provides reusable, interchangeable, components adaptable to many different uses without sacrificing efficiency" [13, back cover text]. The

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

remarkable feature of the STL is that a single client program, with no change except to invoke one or another implementation of a given type of component, can yield two very different binaries: for one implementation, it might invoke a set of subroutines, as in traditional component technologies; for another, it might create in-line code with no subroutine calls at all (what we have referred to as "lightweight" components). The present paper describes a component technology that has this same feature, plus one more: components are delivered as binaries.

The ability to deliver components in binary is important for several reasons. One is the well-known unwillingness of many individuals and organizations to deliver source code. Another is that source code components introduce extra compilation cost (STL is an example of this). The most important reason is that binaries tend to be simpler and less bureaucratic to use, probably because their execution environment is more stable. The STL is the exception that proves the rule: After the STL was first introduced, it took several years before all the major C++ compilers were able to compile it. The problem is fundamental because it is *non*-technical. In principle, we could all agree once and for all on the definition of C++ and its pre-processor, and write components to that definition. In practice, we never can. However, we can — or rather, must — agree on basic conventions for using machine-language components.

Note that in referring to "source code," we include abstract syntax trees. These are more abstract than source code itself, but still subject to the objections raised in the previous paragraph. From our point of view, AST representations are still too concrete.

How, then, can we write generative components without manipulating any concrete version of the source code? How, for example, can we substitute arguments to functions into the functions themselves (perform "inlining")? The answer is that, by using a *compositional* semantic map — in which the meaning of any structured fragment is a function of the meanings of its constituents — we can replace *textual substitution* by *abstraction* and *application*. Suppose component C needs to obtain an integer n from the client, after which it can generate code that is especially efficient for problems of size n. If C were a macro, we would substitute a number, say `100`, into C and compile the result. On the other hand, if the language has a compositional semantic function $[\![\cdot]\!]$, then we can create a representation of the function $F = \lambda n.[\![C[n]]\!]$ and send that representation to the client. The client would then perform the application $F\,[\![100]\!]$, obtaining a value that can be compiled and executed. In fact, if we choose the semantic domain carefully, this compilation process can be made very simple.

All that is required to make this work is an appropriate semantic domain, which we will call *Code*, a compositional semantics $[\![\cdot]\!]$:*Source* $\rightarrow$ *Code*, and a function *compile* from *Code* to machine language. In this framework, a *component* is any value that somehow includes *Code*; we have found that *higher-order values*, such as functions from *Code* to *Code*, are particularly useful. A *client* is simply a function from a component to *Code*. In a companion paper [11], we have argued that a functional *meta-language* augmented with a *Code* type representing the meanings of program fragments in a conventional *object language*, provides a convenient notation for expressing component and client values. The component is written as a higher-order value over *Code*; the client is a function of type $ComponentType \rightarrow Code$; the loader applies the client to the component, converts the resulting value of type *Code* to machine language, and executes that machine language. That is — and this is the fundamental idea of this approach — *the loader is essentially an evaluator of meta-language expressions of type Code*.

What, then, is an appropriate definition of the domain *Code*? Roughly speaking, in this paper we use

$$Code \ = \ Env \ \rightarrow \ (MachineLanguage \ \times \ Env)$$

where *Env* provides the types and locations of variables. The conversion from *Code* to executable is now very simple: apply the *Code* value to an initial environment and then extract the first element of the resulting pair. Components are compiled to *Code* values on the server — that is, they are the binaries compiled from meta-language expressions of type *Code* — and the conversion to machine language is performed by the loader after applying the client to the component. Figure 1 illustrates this idea for a component consisting of a pair of functions, `f` and `g`. The meta-language is called JR, a functional language similar to ML,[1] and the object language is Java.

Note that we cannot take *Code* to be simply *MachineLanguage*. In a phenomenon familiar from denotational semantics, that type is not rich enough to produce a *compositional* semantics of components.

In this paper, we describe and illustrate a proof-of-concept implementation of this idea. The next section gives a set of introductory examples. Section 3 gives the technical details; it begins by discussing a hypothetical simplified language and target machine, and then goes into our prototype. Section 4 gives another example, and section 5 discusses the relationship of this approach to previous work. We have implemented the key aspects of Figure 1, but many technical problems remain to be solved before we have a complete and robust implementation of our ideas; some of these are mentioned in the conclusions.

2 An introductory example

In this section, we build a sorting component to illustrate our approach. This example is virtually identical to the one presented in [11].

Before presenting it, however, we need to give a brief introduction to the meta-language and discuss a few other details relevant to the examples.

All syntactic fragments of Java correspond to values of type *Code*. There is no distinction made between different kinds of values: expressions, statements, etc. However, for clarity of exposition, we will use subscripts to indicate the

[1] One syntactic difference that appears in this figure is that square brackets are used for tupling.

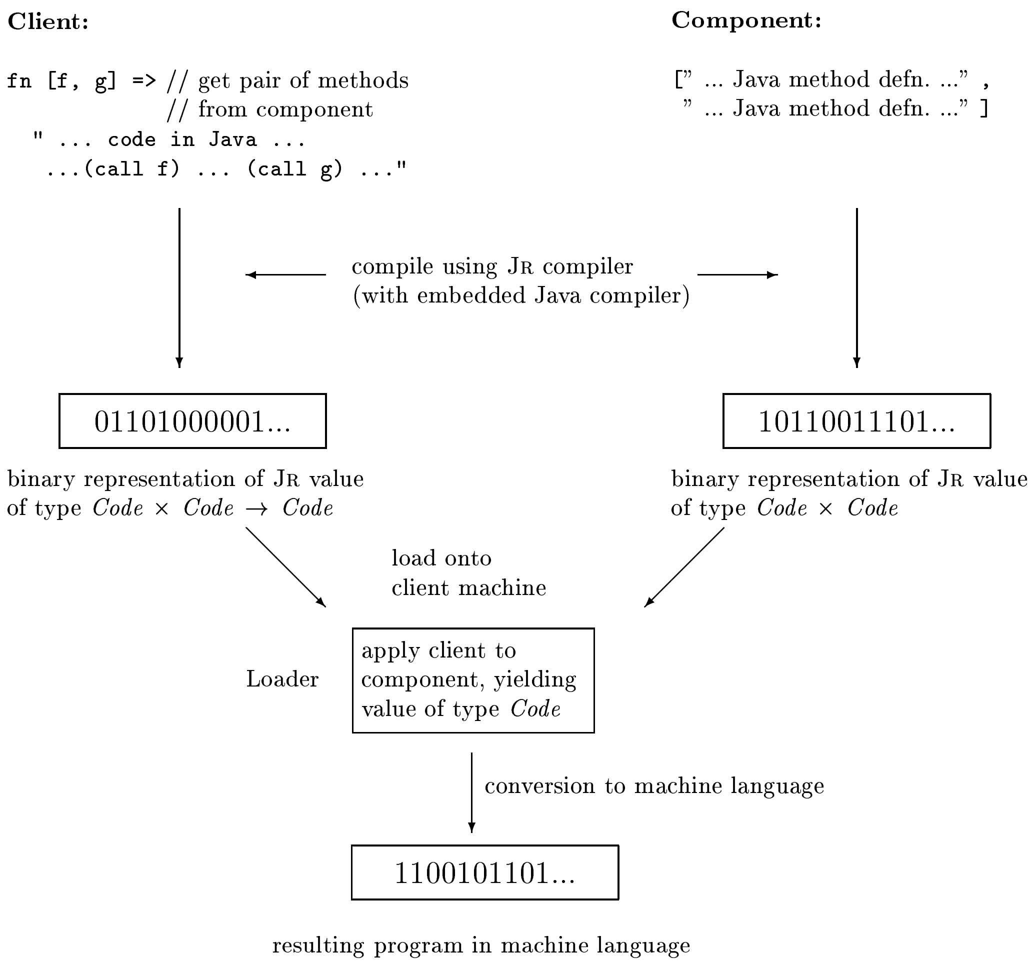

Fig. 1. Combining clients with components

intended kind of value. Bear in mind that *there is only one type of Code*; the subscripts are merely comments.

In this implementation, the meta-language is JR. and the object language is Java. Since we have only a bytecode interpreter for JR, and not a compiler, the "binary representation of JR values" in Figure 1 is actually a program in JR bytecode form. The programs produced by the "conversion" step are in assembly language rather than binary form. And the object language is actually a subset of Java, containing only simple types and arrays (whose semantics is actually closer to that of C arrays), with the usual expressions, statements, and function definitions, but no objects. We trust the reader will have no trouble envisioning how this implementation could be made to match Figure 1 precisely.

JR is similar to Standard ML. There are some minor syntactic differences, but the main one is JR's anti-quotation mechanism, inspired by the one in MetaML [16]. Within JR programs, items occurring in double angle brackets `<<` $\cdots$ `>>` are fragments in the object language (Java); these fragments are JR expressions of type *Code*. They may in turn contain variable, "anti-quoted" parts, introduced by a backquote (`‘`) followed by a syntax category (required for the multiple-entry parser); these anti-quoted parts are JR expressions of type *Code*, which are spliced into the larger *Code* value.

A simple example is this "code template:"

```
fn whatToPrint =>
   << public static void main (String[] args) {
         System.out.println(‘Exp(whatToPrint));
      }
   >>
end
```

This is a function from *Code* (specifically, the code for a Java expression of type `String`) to *Code* (a Java function definition). In other words, it has type $Code_{expr\ of\ type\ \texttt{String}} \rightarrow Code_{fundef}$. We could apply it to an expression like `<<"Hello, world.">>`, or something more elaborate, like `<<args.length>0 ? args[0] : "">>`.

To get back to the sorting component, the simplest version consists of a single procedure:

```
// Component:
val sortcomp =
<<class sortClass {
    static void sort (int[] A, int length) {
      int i=1;
      while (i<A.length) {
         int temp = A[i], j = i-1;
         while (j >= 0)
             if (temp < A[j]) { A[j+1] = A[j]; j--; } else break;
         A[j+1] = temp;
         i++;
      }
```

```
    }
  }
  >>;
```

A client obtains this component (we have nothing to say about *how* it is obtained — possibly it is in a local library, or perhaps it is on a remote component server), and converts it to machine language by calling `make_dll`. `make_dll` converts *Code* values to machine language, and places externally-visible names into a table from which other object code can access them, as usual. Thus, the component can now be used like any other class:

```
// Client:
let val _ = make_dll sortcomp
in
  <<class SortClient {
      static void useSortComponent () {
        int[] keys; ...  sortClass.sort(keys); ...
      }
    }
  >>
end;
```

One way in which the component needs to be more general is in allowing for the specification of different sort orderings. The conventional solution is for the client to provide the ordering as an argument to the sort function. But this solution is verbose and inefficient. We need to define a new class containing the sort function, pass an object to the sort function each time it is called, and invoke an instance method for each comparison. Instead, we can define the component in such a way that the comparison function can be in-lined. Since the client cannot know what the arguments to the comparison function will be, it must provide a function from arguments to *Code*:

$$SortComponentType = (Code_{arg1} \times Code_{arg2} \rightarrow Code)_{comparison} \rightarrow Code_{sort\ function}$$

and, as always, $ClientType = SortComponentType \rightarrow Code$.
The component looks like (note the comparison `temp < A[j]` in the sort method above, from which we are abstracting):

```
fun sortcomp comparisonFun =
   << ...
        if (‘Exp(comparisonFun <<temp>> <<A[j]>>))
      ...
   >>;
```

and the client:

```
let val _ = mk_dll (sortcomp (fn e1 e2 => <<‘Exp(e1) < ‘Exp(e2)>> end))
in ...
end
```

As delivered to the client, the component is an executable that represents the compiled version of the JR function. When provided with the code for the comparison function, it generates the binary for the Java function just as if the "less than" function had appeared in the body of the sort routine all along. This solution is both less verbose (the client does not need to define any new classes or methods) and more efficient (the comparison is inlined) than the conventional one.

This approach can also accommodate extremely lightweight uses, as when the user is sorting only a few elements and doesn't want to pay the cost of a function call. To do this, the user provides the same arguments as above, but the component returns two things: a sort procedure (optionally) and a piece of code to be placed at the point of the sort procedure call. The latter may be replaced either by an actual call or by code to do the sorting in-line. Just as the comparison function provided by the client must take two arguments, so the calling code provided by the component must take an argument, namely the array to be sorted. Thus,

$$
\begin{aligned}
&SortComponentType = \\
&\qquad (Code \times Code \rightarrow Code)_{comparison\ function} \rightarrow int_{size} \\
&\qquad \rightarrow (Code_{optional\ sort\ method} \times (Code_{array\ arg} \rightarrow Code_{call\ to\ sort}))
\end{aligned}
$$

and, as always, $ClientType = SortComponentType \rightarrow Code.$

```
// Component:
  fun sortcomp comparefun size =
  if (size == 2) // place in-line
  then [[],       // no auxiliary class in this case
        fn A => <<if (!(`Exp(comparefun <<`Exp(A)[0]>> <<`Exp(A)[1]>>))) {
                    int temp = `Exp(A)[0];
                    `Exp(A)[0] = `Exp(A)[1];
                    `Exp(A)[1] = temp;
                  }>>
       ]
  else
    [<<class sortClass { ... as above ... } >>,
     fn arg => << sort(`Exp(arg)); >> end ];
```

We note that the use of higher-order functions — in particular, the ability of the component to return a function to the client — is crucial. It is the main reason why we believe the meta-language should be a higher-order language.

3 What is *Code*?

The quoted Java syntax in our examples is simply syntactic sugar for expressions built from abstract syntax operators. These operators are given definitions relative to the chosen *Code* type. For example, **while** has type $Code_{Expr}$ $\times$

$Code_{Stmt} \rightarrow Code_{Stmt}$ and **plus** has type $Code_{Expr} \times Code_{Expr} \rightarrow Code_{Expr}$. These define a compositional semantics for the language, in which *Code* is the domain of meanings.

We have argued that any compositional semantics can be used as the basis for a component system. The components and clients look like macros, but they can be communicated in terms of representations of their *meanings*, not their syntactic representations.

In this view, the key technical decision to be made to create a component system is the definition of *Code*. In this section, we describe our prototype implementation. This definition of *Code* is useful for delivering Java components to Sparc machines. It is by no means the final word on this topic, but it demonstrates the feasibility of the approach.

The prototype definition is, however, necessarily quite complicated. Therefore, we first explain the idea by way of a greatly simplified language and target architecture. This is the topic of the following section; the explanation of the prototype follows it.

3.1 A simple language and architecture

Consider a language consisting only of simple variables, assignment, and the addition operator; the abstract syntax operators are **ident**: $string \rightarrow Code_{Expr}$, **asgn**: $string \times Code_{Expr} \rightarrow Code_{Expr}$, and **plus**: $Code_{Expr} \times Code_{Expr} \rightarrow Code_{Expr}$. Furthermore, the target architecture is a stack machine with instructions `LOAD`, `STORE`, and `PLUS`.

Given a client or component written in this language, we first eliminate the quotes and antiquotes by a using these transformation rules repeatedly (the basic idea is standard [2]):

$\texttt{<<}\ x = e\ \texttt{>>} \rightarrow \textbf{asgn}\ \texttt{"}x\texttt{"}\ \texttt{<<}\ e\ \texttt{>>}$
$\texttt{<<}\ e + e'\ \texttt{>>} \rightarrow \textbf{plus}\ \texttt{<<}\ e\ \texttt{>>}\ \texttt{<<}\ e'\ \texttt{>>}$
$\texttt{<<}\ x\ \texttt{>>} \rightarrow \textbf{ident}\ \texttt{"}x\texttt{"}$
$\texttt{<<}\ \text{‘}e\ \texttt{>>} \rightarrow e$

For example, suppose we have the following component, of type $Code \rightarrow Code$:

```
Component = fn arg => <<‘arg + ‘arg>>
```

and this client of type $(Code \rightarrow Code) \rightarrow Code$:

```
Client = fn comp => <<x = ‘(comp <<y>>)>>
```

Eliminating the syntactic sugar by the above transformation rules, these become:

```
val Component = fn arg => plus arg arg
val Client = fn comp => asgn "x" (comp (ident "y"))
```

We can now define the abstract syntax operators using our chosen definition of *Code*. Since we are compiling for a stack machine, and will ignore the issue of declarations (assuming all identifiers to be declared by some other mechanism), we can use a particularly simple definition:

$$\begin{aligned} Code &= Environment \rightarrow MachineLang \\ Environment &= string \rightarrow string \\ MachineLang &= string \end{aligned}$$

The environment maps identifiers to the names of memory locations. A *MachineLang* value is simply a sequence of machine language instructions, regarded as a string. Again, our machine has three instructions: `PLUS` takes two arguments off the stack and puts one result back on it, `LOAD` places a value on the stack, and `STORE` removes a value.

Now we can define the semantics of our language (carat is the string concatenation operator):

```
fun ident s // : string -> Code
       = fn rho => "LOAD " ^ (rho s) ^ "\n" end
fun asgn s C // : string x Code -> Code
       = fn rho => (C rho) ^ "STORE " ^ (rho s) ^ "\n" end
fun plus C1 C2 // : Code x Code -> Code
       = fn rho => (C1 rho) ^ (C2 rho) ^ "PLUS\n" end
```

Applying these definitions to our example, we have:

```
val Component = fn arg => fn rho =>
         (arg rho) ^ (arg rho) ^ "PLUS\n" end end
val Client = fn comp => fn rho =>
         (comp (ident "y") rho)
           ^ "STORE " ^ (rho "x") ^ "\n" end end
```

The component and client are not *transformed* into this form. Like any other expression, they are *compiled* into machine language code that has these meanings. The loader loads them both and applies the client to the component; that is, it sets up the run-time state that JR code needs for an application.

When complete, this application will leave the (binary representation of the) value:

```
fn rho => "LOAD " ^ (rho "y") ^ "\n" ^ "LOAD " ^ (rho "y")
          ^ "PLUS\n" ^ "STORE " ^ (rho "x") ^ "\n"
```

Finally, the loader may supply an initial environment to transform this value of type *Code* to a value of type *MachineLang*. Let us finish the example by assuming the loader provides the environment { "x" → "Lx", "y" → "Ly" }. The resulting machine language code:

```
LOAD Ly
LOAD Ly
PLUS
STORE Lx
```

can now be executed by the machine.

3.2 Java components for the Sparc architecture

We have implemented a prototype for writing components for (a compiled version of a subset of) Java, to run on Sparc architectures. The set of abstract syntax operations and the translation from the antiquote notation to these constructors is much more complicated than in the simple example, but it is conceptually similar and we will not go into detail. More to the point, the definition of *Code* and the definitions of the operations are also much more complicated, and on these we will elaborate.

Evaluating expressions of type *Code* should be an efficient process and should produce efficient code. Both forms of efficiency depend crucially on the exact definition of the *Code* type. The meanings of syntactic phrases — that is, values of type *Code* — can carry a great deal of structure, allowing the relatively slow generation of efficient code, or very little (as in the simple example above), allowing the rapid generation of inefficient code. Another dimension of variability is the amount of preprocessing performed on components: one extreme example is to let $Code = string$, so that components are just macros and code generation is compilation. At the other extreme is the work we present here, in which components are fully compiled to target machine language.

We have designed our definition of *Code* to allow for reasonably good code generation. The definition below makes use of a "register status" value that allows for a simple but useful register allocation process during the conversion from *Code* to machine language. For simplicity, this single *Code* type represents the meaning of any kind of syntactic fragment — expression, statement, declaration, etc. This means that some parts of each value may be irrelevant, such as the return location of a value representing the meaning of a statement.

$$\begin{aligned}
&\mathit{Code} &&= \mathit{Env} \rightarrow (\mathit{MachLang} \times \mathit{Locator} \times \mathit{Type}) \times \mathit{Env} \\
&\mathit{MachLang} &&= \mathit{MachLangFrag}_{\textit{l-value}} \times \mathit{MachLangFrag}_{\textit{r-value}} \times \mathit{MachLangFrag}_{\mathit{decl}} \\
&\mathit{Env} &&= (\mathit{Var} \rightarrow \mathit{Type} \times \mathit{Size} \times \mathit{Locator}) \times \mathit{StackOffset} \times \mathit{RegStatus} \\
&\mathit{Locator} &&= \textit{temp-addr} + \textit{live-addr} + \textit{synch-addr} \\
&\mathit{RegStatus} &&= (\mathit{RegNum} \cup \{\mathit{Nowhere}\}) \leftrightarrow (\mathit{Locator} \cup \{\mathit{Empty}\})
\end{aligned}$$

(The double arrow denotes a relation.)

Given an environment, each *Code* value produces some machine language and a "locator" and type, which give the location and type of the value returned by this bit of machine language (if any); for declarations, the piece of code might alter the environment, so a new environment is passed back as well. The machine language part of the value is actually three sequences of machine instructions, one to calculate the l-value, one for the r-value, and one for constants that are to be placed into the data segment. (In our prototype, we represent the machine language sequences by strings, in the form of assembler input.)

The environment has three parts: a table giving information about the variables, the next available stack location for future variable declarations, and the current availability of registers. For each variable, the provided information is its type, size, and location. Locations are of three types: live, temporary, and synchronized. These distinctions are important to the register allocator, as we will explain.

The environment provided to each *Code* value includes a table, called *RegStatus*, giving the contents of the registers. This table relates register numbers to locations: a given register number can contain a value that is also in a location, or it may contain nothing; similarly, the value contained in a location may also be in a register, or not. The allocation of registers depends upon the register status and also on the type of locators used in the code being compiled. A register containing a temporary value can be recycled once that value is used; a live value needs to be stored back on the stack if its register is to be used for anything else; and a synchronized location is one which is currently stored in a register, so that spilling that register is a no-op.

To illustrate the effect of this definition, consider this client and component:

```
fun client exp = <<int x = 2; int y = x+5; `Exp(exp);>>;
val component = <<(x+y)*(x-y)>>;
```

The application of the client to the component yields the following machine language:

```
setsw 2, %i0
setsw 5, %i1
add %i0, %i1, %i1
add %i0, %i1, %i2
sub %i0, %i1, %i3
mulx %i2, %i3, %i2
```

This code uses an efficient register allocation scheme. It is the kind of code that might have been produced by a compiler that was given the entire, non-componentized, statement.

Figures 2 and 3 contain representative examples of combinator definitions: addition[2] and sequencing. In this implementation, "machine language" means "assembly language." Fragments of assembly language are represented by strings, so that constructing code from fragments involves a lot of string concatenation; carat (^) is the string concatenation operator.

Consider the addition operator. Its left operand produces some machine code (only the r-value, `rcf1`, is of interest), and may modify the environment (in particular, the stack configuration and register status). The right operand does likewise. After the right operand has been compiled, the triple `[e,s,R]` represents the environment, that is, the variable map, stack offset, and register status. The register allocator uses the register status, the stack offset, and the locations of the two operands to find a location for the sum; this is `r3`, and `R'` is the register status after it has been used. The register allocator may also need to generate some spill code; this is `cf3`. The code generated for the addition is the r-value

[2] restricted to integers. The actual definition handles floats as well, using the *Type* of its arguments to determine the type of op-code to emit.

code for both operands, the spill code (if any), and then the add instruction. The result has locator `TEMP s`, indicating that its home, if it needs to be spilled, is at that offset of the stack, and that it is temporary. The stack top is updated and `R'` is returned as the new register status.

```
fun add opnd1 opnd2 =
  fn E =>
    let val [[[lcf1, rcf1, df1], l1, t1], E'] = opnd1 E
        val [[[lcf2, rcf2, df2], l2, t2], [e,s,R]] = opnd2 E'
        val [[r1, r2], r3, cf3, R'] = getreg R [(TEMP s)] [l1, l2]
    in
        [[["", rcf1^rcf2^cf3^"      add "^r1^","^r2^","^r3, df1^df2],
         TEMP s, ["int", 0]], [e, s+(stacksizeof "int"), R']]
    end
  end;
```

Fig. 2. The addition combinator (restricted to integers)

```
fun seq op1 op2 =
  fn E =>
    let val dummy = if trace then print "seq" else ""
        val [[[lcf1, rcf1, df1], l1, t1], [e,s,[locnos, regnos]]] =
            op1 E
        val R' = if (istemp l1)
                 then freereg (regof (lookup regnos (locof l1))) [locnos, regnos]
                 else [locnos,regnos]
        val [[[lcf2, rcf2, df2], l2, t2], E''] = op2 [e,s,R']
    in
      [[["", rcf1^rcf2, df1^df2], l2, t2], E'']
    end
  end;
```

Fig. 3. The sequencing (“semicolon”) combinator

The total cost of using a component consists of the cost of code generation and the cost of executing the generated code. To be worthwhile, these costs must be lower than the cost of using a non-generative component. In our prototype, the cost of code generation is unrealistically high, because we have only an interpretive implementation of JR, and because *Code* uses assembly language programs, represented as strings, for the generated machine language. A compiled meta-language directly generating machine code would be more realistic. Thus, we cannot give meaningful results concerning the combined costs. Just to illustrate the potential gain, we give an example whose execution-time speed-up is unusually great. This example is a component that adds a cache to a function [11]. The trick to doing this is that the function may contain *recursive* calls that need to be changed to call the new, caching version of the function. For this reason, a caching component cannot have type $Code \to Code$, but must instead have type $(string \to Code) \to Code$. That is, the client presents a function with a missing function call. The component fills it in with a call to the caching version of the function.

The code is given in the appendix. The function computes the ith Fibonacci number using the simple recursive definition. We give two different components. One incorporates a fixed-size cache of ten integers; this should give a significant speed-up for calculating Fibonacci numbers from 0 to 10, after which the exponential nature of the algorithm will reappear. The other incorporates a two-element cache, one for even arguments and one for odd arguments; by the nature of the algorithm, this cache is effective for all arguments. We also give a control example that uses a non-caching component.

The results show that the caching components work as we would expect. Times are in seconds, and include code generation and a single execution of the indicated function call:

Function call	**No cache**	**10-element cache**	**Even-odd cache**
`fib(10)`	.02	0.180	0.180
`fib(20)`	.02	0.180	0.200
`fib(30)`	.29	0.250	0.170
`fib(40)`	33.47	6.99	0.210

As just mentioned, this implementation is very inefficient in producing code. Code generation overhead dominates the times for the first three rows; we can see this because the completely linear version — the one with the even-odd cache — shows no consistent change in execution speed. However, the times for `fib(30)` and `fib(40)` show the time increase for either the uncached or the 10-element cached version. In short, even counting the overheads of our prototype — overheads which could be reduced to a fraction of their current values — we can still see a speed-up relative to the uncached version of the code.

In one sense, this example is unfair: few components will turn an exponential algorithm into a linear one. On the other hand, the example does illustrate a genuine advantage of our method: no non-generative technology can possibly produce such a speed-up. This points toward the possibility of constructing programs out of programming *methods* that are supplied as components.

4 A search-and-replace component

Many editors contain a regular-expression matching facility with a "submatch" facility. That is, the pattern can contain sub-patterns, usually within escaped parentheses ("`\(`" and "`\)`"). When the pattern has matched an object string, the parts of that string that were matched by these subpatterns can be referred to by sequence number. For example, in the Unix `ex` editor, the command `s/\(a*\)b\(c*\)/\2b\1/` finds a sequence of `a`'s followed by a `b` followed by a sequence of `c`'s, and swaps the `a`'s and `c`'s; thus, if the object string is `aaaaabccdef`, this command will change it to `ccbaaaaadef`.

Patterns like this are among the features that make languages like Awk and Perl preferable to Java for simple text-processing. Although C has a library subroutine that does regular expression matching, it is awkward to use because there is no simple way to define actions that can be attached to the regular expressions.

The component shown in Figures 4 and 5 offers such a facility to the Java programmer. In addition to the usual pattern-forming operations like repetition (here represented by the function `star`) and concatenation (the infix operator `**`), the (infix) operator `==>` adds an action to any pattern, which is to be executed when the pattern is matched. `==>` has type

$$Pattern \times (Code_{start\ of\ match} \rightarrow Code_{end\ of\ match} \rightarrow Code_{action}) \rightarrow Pattern$$

As an example, if a client were to include this expression:

```
‘((star ((plus (oneof <<"0123456789">>) ==>
           (fn s e => <<sum = sum+convertdec(input,‘s,‘e-1);>> end))
         ** (star (oneof <<" ">>))))
     <<input>> <<0>> <<z>>)
```

it would do the following: Within the string `input`, starting at position 0, it would find runs of integers separated by spaces, and add their values to the variable `sum`; then it would assign to `z` the position just past the end of the match.

5 Related work

Efforts to componentize software in non-traditional ways include Kiczales's aspect-oriented programming, Engler's ‘C and Magik systems, and the C++ template-based approaches, including the STL and the template meta-programming method of Veldhuizen, Czarnecki, and Eisenecker. Also relevant is the work on the "explicitly staged" language MetaML.

The idea behind aspect-oriented programming (AOP) [12] is to separate *algorithms* from certain details of their implementation — like data layout, exception behavior, synchronization — that notoriously complicate programs. Each such detail, or *aspect*, is described in a specification distinct from the algorithm. An aspect *weaver* combines the algorithm and the various aspect specifications into an efficient program. Our approach can account for these details as well, but it is not as convenient notationally because the client programmer has to know about, and plan for, the use of components. On the positive side, our approach is fundamentally simpler in that it requires little new technology. The most serious

```
val namecnt = 0;
fun newname x = (namecnt := namecnt+1; x^(tostring namecnt));

fun str S =
  fn buff startpos endpos =>
     let val i = newname "i"
     in <<int `i = 0;
          while (`S[`i] != '\0' &&
                 `buff[`startpos+`i] != '\0' &&
                 `S[`i] == `buff[`startpos+`i])
            `i++;
          if (`S[`i] == '\0') // match succeeded
            `endpos = `startpos+`i;
          else
            `endpos = -1;
        >>
     end
  end;

fun oneof S =
  fn buff startpos endpos =>
     let val i = newname "i"
     in <<int `i = 0;
          if (`buff[`startpos] == '\0')
            `endpos = -1;
          else {
            while (`S[`i] != '\0' &&
                   `S[`i] != `buff[`startpos])
              `i++;
            if (`S[`i] == `buff[`startpos]) // match succeeded
              `endpos = `startpos+1;
            else
              `endpos = -1;
          }
        >>
     end
  end;
```

Fig. 4. A search-and-replace component (part 1)

```
infixl 4 "**"
fun ** r1 r2 =
  fn buff startpos endpos =>
     let val i = newname "i"
     in <<int `i;
          `(r1 buff startpos endpos)
          if (`endpos != -1) {
            `i = `endpos;
            `(r2 buff i endpos)
          }
        >>
     end
  end;

fun star r =
  fn buff startpos endpos =>
     let val i = newname "i"
     in <<int `i = `startpos;
          while (`i != -1) {
            `endpos = `i;
            `(r buff endpos i)
          }
        >>
     end
  end;

fun plus r = r ** (star r);

infixl 5 "==>"
fun ==> r S =
  fn buff startpos endpos =>
    <<{`(r buff startpos endpos)
       if (`endpos != -1)
         `(S startpos endpos)
      }
    >>
  end;
```

Fig. 5. A search-and-replace component (part 2)

problem with AOP is that it provides only a conceptual framework. Each set of aspect languages and their associated weaver is *sui generis*. Our method uses conventional functional languages to write all components.

Engler's `'C` ("tick-see") [7] is a C extension that includes a *Code* type and run-time macros. Engler's goal differs from ours — in his examples, the extension is mainly used to achieve increased efficiency, à la partial evaluation. The main technical difference from the current work is that the meta-language is C rather than a functional language; one consequence is the absence of higher-order functions, which are fundamental in our approach. Engler has also written a system called Magik [6], in which one can create language extensions by programming transformations of abstract syntax trees. Magik is extremely powerful, but is inherently a compile-time system, and is in general much harder to use than a system that can be programmed at the source level (such as `'C` or ours).

Most generative component systems are compile-time. Simonyi's Intentional Programming [14] approach is another. Sirkin et al's Predator [15] data structure precompiler is specialized to the production of efficient data structure-manipulating code, and as such goes beyond anything we have attempted. Batory's Jakarta Tool Suite [1] is a set of tools for the generation of domain-specific languages; domain-specific optimizations are performed at the AST level. Access to concrete representations of programs always provides for greater power and flexibility. The extent to which the power of these other component systems could be realized at the binary level is an interesting question for further study.

The approaches based on aggressive use of the C++ template mechanism are an interesting contrast to ours. Stepanov's Standard Template Library (STL) [13] accounts for lightweight components, based on the `infix` mechanism of C++. Veldhuizen [17] and Czarnecki and Eisenecker [4, 5] take this idea even further. Observing that the template mechanism actually provides a Turing-complete *compile-time* language embedded in C++, they show how to create "template metaprograms," thereby accounting for generative components as well. Furthermore, the integration of the static and dynamic languages has definite benefits, including a cleaner syntax in many cases. However, the static language provided by the template mechanism is a rather awkward one, and writing template metaprograms is a tour-de-force. Furthermore, the entire template approach is subject to the criticism that all components must be provided in source form. Our approach yields many of the benefits of these approaches, uses a more powerful meta-language, and permits component distribution in binary form.

In MetaML [16], one writes in a single language, but explicitly indicates where partial evaluation can be performed; that is, the dynamic parts of a program are specially bracketed, just as the Java parts of our components and clients are. This approach is simpler for the programmer, especially inasmuch as the language permits some type-checking of *generated* code to be performed at compile-time. Such a feature would make our system much easier to use, but we have not yet explored this possibility. As a general-purpose component mechanism, our system offers two advantages. First, we have greater control over the representation of the generated programs. Second, by distinguishing the meta-language from the object language, we can choose an object language that reflects a simpler, more realistic, machine model than the meta-language, so that it can be compiled more efficiently. For example, when using Java, the absence of "upward function values" allows us to place on the stack some values that, if we remained in MetaML, would have to go into the heap. In other words, programs in MetaML are made more efficient by eliminating static computations, but what remains are still programs in MetaML and can only be compiled as efficiently as MetaML. Therefore, the differing compilation models of the two languages is, for us, a crucial distinction.

6 Conclusions

To us, a "software component" is simply a value — usually, a higher-order value — that contains *Code*. This definition is both general and conceptually simple. It admits conventional "ADT-style" components, but also includes lightweight (inlined) and generative components. Based on ideas that are well-known in the functional programming community, it is relatively easy to implement because the basics are found in existing functional languages. All that is needed is an implementation of *Code* constructors, plus, in practice, a multiple-entry parser for the quotation mechanism. Our prototype implementation demonstrates that this notion is workable.

The major problem we have observed is the difficulty of writing components and clients. The semantic mismatch between meta-language and object language is the primary culprit. In staged languages like MetaML [16] where the languages at every stage are identical, there is no such problem. On a superficial level, however, it may be possible to simplify the uses of quotation and anti-quotation by, to some extent, inferring their proper placement. Since the object language and meta-languages are distinct, the "stage" of any subexpression can be determined syntactically. Nonetheless, we see no way to fundamentally simplify the construction of components. One possibility is to type-check components as is done in MetaML. This appears to be much more difficult to do in our setting, because the program-constructing operations (in the meta-language) have no semantic connection to the constructed programs (in the object language). However, we have not attempted a systematic study of this problem.

The heart of the matter is the definition of *Code* and the implementation of the component system. How efficiently can a well-engineered loader evaluate *Code* expressions? Should *Code* values carry more information than the register state to allow for higher quality code to be generated, or will the increase in *Code* evaluation time increase overall system response time? Would it be better to eliminate the register state entirely, relying on run-time code generation techniques to improve code efficiency? Although our principal goal is increased programmer productivity rather than efficiency, the method cannot achieve widespread use if it is overly costly in execution speed.

The cost of evaluation of *Code* values also depends upon how much of the value can be pre-computed. This is a challenging partial evaluation problem. In a client `fn e => <<... ‘Exp(e)...>> end`, it is likely that much of the code surrounding the use of `e` could be pre-calculated. However, in `fn t => <<‘Id(t) x; int y; ...x...y...>> end`, where the parameter is a type, it may be difficult to pre-compile very much of the program at all, because the location of `y` is not known. Register allocation adds considerably to the difficulties: a statement or expression abstracted from the program makes it difficult to do register allocation for the code that follows. Our prototype implementation does no such pre-compilation, and we consider this a particularly important area for study. (It is worth emphasizing that we do not advocate passing abstract syntax trees as components. In addition to the disadvantages mentioned in the introduction, there is another crucial problem with this representation: AST's are, in principle, impossible to pre-compile, because there is always the possibility that they will be modified before conversion to machine language.)

Our components are in no sense portable across architectures. However, there is an alternative approach: instead of taking the *MachLang* type to be Sparc code, we could have it be code for some virtual machine. The conversion from *Code* to machine language would then be more involved: the loader would provide some initial environment and extract the *MachLang* portion of the result (as now), and then compile the virtual machine code to actual machine code. Obviously, this is being done now on a daily basis by the "just-in-time" compilers for Java [9]. We would only remark that the Java virtual machine is perhaps not the best intermediate language for our purpose — a three-address code would seem more appropriate — and that the efforts to add annotations to Java virtual machine code to help the code generator [10, 8] seem particularly apposite here.

Another key area for study is integration between the meta-language and object language. As discussed in the related work section, C++'s standard template library is notationally simpler to use, in part because of such integration. A multi-level language such as MetaML, in which both levels are the same, also achieves such integration, but it does not strike us as an ideal setting for a general component framework, for reasons we have stated earlier. On the other hand, the "Turing tarpit" looms here: the combination of two entirely different, and independently complex, languages may push the entire package beyond the realm of practical usability. Integrating the two levels could go some a considerable distance toward alleviating this problem. It would also allow us to address a problem that we have not mentioned: capture of variables. In Scheme, this problem has led to the development of "hygienic macros" [3]; since we have a similar combination of features — open terms and higher-order functions — we may be able to benefit from this technology.

Acknowledgements

Uday Reddy and An Le provided useful feedback on an earlier draft of this paper. We also appreciate the help of Joel Jones, whose compilation expertise helped us implement the prototype. The anonymous SAIG referees made many useful suggestions, for which we are most grateful.

References

1. Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Fifth Intl. Conf. on Software Reuse*, June 1998.
2. A. Bawden. Quasiquotation in lisp. In *In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-99)*, pages 18–42, January 22–23 1999.
3. William Clinger and Jonathan Rees. Macros that work. In *Proc. of the Conf. on Principles of Programming Languages (POPL)*, pages 155–160, 1991.
4. Krzysztof Czarnecki and Ulrich W. Eisenecker. Synthesizing objects. In Rachid Guerraoui, editor, *13th European Conference on Object-Oriented Programming (ECOOP '99)*, pages 18–42, June 1999.
5. Krzysztof Czarnecki and Ulrich W. Eisenecker. Static metaprogramming in C++. In *Generative Programming: Methods, Techniques, and Applications*, chapter 8, pages 251–279. Addison-Wesley, 2000.
6. Dawson Engler. Incorporating applications semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, USA, October 15–17 1997.
7. Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynaic code generation. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg Beach, Florida, 21–24 January 1996.
8. J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the java byte codes in support of optimization. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.
9. Sun Microsystems Incorporated. The java hotspot performance engine architecture: A white paper about sun's second generation performance technology. Technical report, April 1999.
10. J. Jones and S. Kamin. Annotating java class files with virtual registers for performance. *Concurrency: Practice and Experience*, to appear.
11. Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components I: Source-level components. In *First International Symposium on Generative and Component-Based Software Engineering (GCSE'99)*, September 28–30 1999.
12. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 1241. Springer-Verlag, June 1997.
13. David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley Professional Computing Series, 1996.
14. Charles Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, 1995.
15. Marty Sirkin, Don Batory, and Vivek Singhal. Software components in a data structure precompiler. In *Intl. Conf. on Software Eng.*, pages 437–446, 1993.
16. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 203–217, New York, June 12–13 1997. ACM Press.

17. Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

Appendix

This appendix contains the code for the caching example discussed in section 3.2. We give the client first, followed by two caching components. Each component creates a class called `cacheComponent` which contains two methods, `cacher` and `setupcache`. Each of these methods has a single integer argument and returns the Fibonacci number of that argument; the difference is that `setupcache` initializes the cache before calling `cacher`. For our experiments, we timed a single call to `setupcache`, with various arguments. `quickCacheComponent` is the component that uses the simple 10–element cache. `moduloCacheComponent` uses the 2–element cache. The timings shown in the second and third columns of the table in section 3.2 were obtained from running `(theClient quickCacheComponent)` and `(theClient moduloCacheComponent)`, respectively.

```
fun fibmaker thename =
 <<if (x < 3)
      return x;
   else
      return (`Method(thename) (x-1) + `Method(thename) (x-2)); >>;

fun theClient theComponent =
    let val code = load (theComponent fibmaker)
    in (seq code <<cacheComponent.setupcache(5);>>)
    end;

fun quickCacheComponent thecode =
   <<class cacheComponent {
      static int cachesize;
      static int[] ncache;

      private static int original (int x) {
          `Stt(thecode "cacher")
      }

      private static int cacher (int x) {
        if ((x < cachesize) && (ncache[x] != -1))
          return ncache[x];
        else {
            int newres;
            newres = original (x);
            if (x < cachesize)
              ncache[x] = newres;
            return newres;
        }
      }
```

```
    public static int setupcache (int x) {
      int i = 0;
      cachesize = 10;
      ncache = new int[cachesize];
      while (i < cachesize) {
        ncache[i] = -1;
        i++;
      }
      return cacher (x);
    }
  }>>;

fun moduloCacheComponent thecode =
  <<class cacheComponent {
      static int cachesize;
      static int[] cachekeys;
      static int[] cachevalues;

      private static int original (int x) {
        `Stt(thecode "cacher")
      }

      private static int cacher (int x) {
        if (cachekeys[x%cachesize] == x)
          return cachevalues[x%cachesize];
        else {
          int newres;
          newres = original (x);
          cachekeys[x%cachesize] = x;
          cachevalues[x%cachesize] = newres;
          return newres;
        }
      }

      public static int setupcache (int x) {
        int i = 0;
        cachesize = 2;
        cachekeys = new int[cachesize];
        cachevalues = new int[cachesize];
        while (i < cachesize) {
          cachekeys[i] = -1;
          i++;
        }
        return cacher (x);
      }
   }>>;
```