Lightweight and Generative Components II: Binary-Level Components

Sam Kamin\*, Miranda Callahan, and 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 [·], 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 \([·]:\text{Source} \rightarrow \text{Code}\), 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,