Lambda Functions for C++0x

Jaakko Järvi
Texas A&M University
College Station, Texas, U.S.A
jarvi@cs.tamu.edu

John Freeman
Texas A&M University
College Station, Texas, U.S.A
jfreeman@cs.tamu.edu

ABSTRACT

A "functional" style of programming has become common in C++, following the introduction of the "Standard Template Library" (STL) into C++'s standard library. C++ is, however, notably lacking in its support for this style and thus cannot take full advantage of its own standard libraries. C++'s mechanisms for defining functions or objects to pass to STL algorithms are overly verbose. The effective use of modern C++ libraries calls for *lambda functions* in the language.

This paper describes a design and implementation of built-in lambda functions for C++. C++'s compilation model, where activation records are maintained in a stack, and the lack of automatic object lifetime management, make safe lambda functions and closures challenging: if a closure outlives its scope of definition, references stored in a closure dangle. Our design is careful to balance between conciseness of syntax and explicit annotations to guarantee safety.

Lambda functions can be declared without annotating their parameter types. C++0x, the forthcoming revision of standard C++, supports constrained templates and modular type checking. We describe how to infer parameter types of lambda functions from the constraints of generic functions in order to support modular type checking in the presence of lambda functions. Our design is currently under consideration for adoption to C++0x.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions, and subroutines; polymorphism*

General Terms

Design, languages

Keywords

C++, STL, closures, lambda functions

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

1. INTRODUCTION

Many programming languages offer support for defining local unnamed functions "on-the-fly", within another function or expression. These languages include practically all functional programming languages and also a growing number of imperative or object-oriented mainstream languages, C# 3.0 [20, §26.3], Python [7, §5.11], and ECMAScript [5, §13], to name a few. Local unnamed functions, often called *lambda functions* or *lambda expressions*, have many uses in day-to-day programming: as arguments to functions that implement various traversals, as callbacks triggered by I/O events in GUI objects, and so forth. Even outside of primarily functional programming languages, lambda functions can be considered part of the (desired) toolbox of mainstream programming.

Lambda functions are not a feature of C++. We consider this a shortcoming, especially since modern C++, with the Standard Template Library (STL) [23] as the backbone of its standard library, encourages a rather "functional" programming style where higher-order functions are frequent. For example, many oft-used STL algorithms implement common traversal patterns and are parametrized on functions. Examples include the **accumulate**, **remove_if**, and **transform** algorithms, whose counterparts in the context of functional languages are, respectively, the **fold**, **filter**, and **map** families of functions. If the STL encourages a functional programming style, the lack of a syntactically light-weight mechanism for defining simple local functions discourages it—and is a hindrance to the effective use of C++'s own standard libraries.

Typically, a lambda function has access to local definitions in the enclosing scope of its definition. The term *closure* refers to the value of a lambda expression, consisting of the code of the function and the environment in which it was defined. The environment consist of the local variables referred to in the lambda expression. Using the terminology of lambda calculus, we call such variables *free*. In situations naturally programmed with lambda functions, C++ programs rely on the well-known connection between closures and objects—member variables of a class store the environment, and the code of the lambda function is placed in a member function, usually the function call operator. In the context of C++, objects of such classes are called *function objects*.

Defining new classes and naming them explicitly is unreasonably verbose for the purpose of defining function objects to emulate lambda functions. Sophisticated template libraries [4,17,19] have finessed the function object approach to a small embedded language resembling that of writing