Science of Computer Programming 75 (2010) 762–772

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

C++ lambda expressions and closures

Jaakko Järvi *, John Freeman

Texas A&M University, College Station, TX, USA

ARTICLE INFO

Article history:
Received 8 August 2008
Received in revised form 9 January 2009
Accepted 30 April 2009
Available online 15 May 2009

Keywords:
C++
Closures
Lambda expressions

ABSTRACT

A style of programming that uses higher-order functions has become common in C++, following the introduction of the Standard Template Library (STL) into the standard library. In addition to their utility as arguments to STL algorithms, function parameters are useful as callbacks on GUI events, defining tasks to be executed in a thread, and so forth. C++'s mechanisms for defining functions or function objects are, however, rather verbose, and they often force the function's definition to be placed far from its use. As a result, C++ frustrates programmers in taking full advantage of its own standard libraries. The effective use of modern C++ libraries calls for a concise mechanism for defining small one-off functions in the language, a need that can be fulfilled with *lambda expressions*.

This paper describes a design and implementation of language support for lambda expressions in 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. The presented design is included in the draft specification of the forthcoming major revision of the ISO C++ standard, dubbed C++0x. In rewriting typical C++ programs to take advantage of lambda functions, we observed clear benefits, such as reduced code size and improved clarity.

1. Introduction

Many programming languages support defining local unnamed functions "on-the-fly", within another function or expression. These include practically all functional programming languages and also a growing number of imperative or object-oriented mainstream languages, such as C# 3.0 [25, Section 26.3], Python [26, Section 5.11], and ECMAScript [8, Section 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 graphical user interface widgets, as tasks to be executed in a concurrent thread, 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) [29] as the backbone of its standard library, encourages a programming style where higher-order functions are commonplace. 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. The lack of a syntactically lightweight mechanism for defining simple local functions is a hindrance to taking advantage of STL's abstractions, and thus to the effective use of C++'s own standard libraries.

* Corresponding address: Department of Computer Science, Texas A&M University, TAMU 3112, College Station, TX 77843, United States.
E-mail addresses: jarvi@cs.tamu.edu (J. Järvi), jfreeman@cs.tamu.edu (J. Freeman).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.04.003

Typically, a lambda expression has access to local definitions in the enclosing scope of its definition. The term *closure* refers to the value of a lambda expression, comprising the code of the function and the environment in which it was defined. The environment consists of the local variables referenced 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 a member function contains the code of a lambda function. Usually this member function is the function call operator, which can be invoked with the function call syntax, as if the object was an ordinary C++ function. Such objects are called *function objects*.

Although function objects can serve as closures, they are not particularly well-suited for emulating lambda expressions. Defining a class and constructing an object of that class is syntactically verbose. In particular, if a closure's environment is not empty, defining member variables and a constructor is necessary. Moreover, C++ restricts the use of unnamed classes defined within function bodies, so that programmers usually need to invent a name for a class that emulates a lambda expression and place the class definition in "namespace scope", textually far from the use of the class. Sophisticated template libraries [23,18,5] have finessed the function object approach to a small embedded language resembling that of writing local unnamed functions, but the solutions remain inadequate, as explained in Section 2.

This paper describes a design for lambda functions for C++ as a built-in language feature. We define the semantics of lambda functions via their translation to function objects; our implementation applies this translation at the level of abstract syntax trees. The approach resembles that of, say, C#, where lambda expressions can be regarded as anonymous class definitions [25, Section 26.3].[1] In a language without automatic object lifetime management, such as C++, this approach is challenging. In particular, if a lambda function outlives the scope of its definition, free variables in the lambda function's body dangle. Consequently, our lambda functions entrust the programmer with control over the closure's contents. To be minimally disruptive to C++'s type system, our design does not introduce a new kind of function type; a lambda expression has an unspecified type. In situations where a definite type is necessary, lambda functions integrate smoothly with the well-established library facility of *polymorphic function wrappers* [15], [1, Section 20.7.16], which can wrap arbitrary function objects in an object whose type defines an explicit call signature.

C++'s manual memory management, modular type checking (discussed below), stack-based management of activation records, and the already rich feature set all conspire against introducing a language construct for lambda functions—yet it is sorely needed in C++. We present a design that takes the above constraints into account and, we believe, fits well into C++. Our design has been accepted by the ISO C++ standards committee [17] and is part of the working draft of the next standard revision [1]. Our (partial) implementation is publicly available as a branch of GCC [9].

Beyond what is part of the draft standard, we discuss polymorphic lambda expressions, where parameter types need not be declared. Polymorphic lambda functions are desirable for their conciseness—the C++ standards committee has expressed support for adding the feature at a later phase of the standardization process. The next revision of standard C++, dubbed C++0x, includes constrained templates and supports their modular type checking [13]. To extend modular type checking to polymorphic lambda functions, their parameter types must be inferred. C++ does not support type inference in general, but it is possible to deduce the parameter types of a polymorphic lambda function from the context of its definition. Specifically, when a lambda function is bound to a type parameter of a template, the constraints of that type parameter contain a call signature for the lambda function; this information is enough for deducing the parameter types and, consequently, type checking the body of the lambda function. Finally, C++'s unconstrained templates support polymorphic function objects that can be passed into generic functions and invoked at different call sites with different argument types. This form of polymorphism can be preserved for lambda functions in the context of C++0x's constrained templates. A generic function can constrain the same function object type with multiple call signatures—type checking the lambda function body against each signature guarantees the absence of type errors in the body of the generic function.

2. Motivation and background

Function objects are an expressive mechanism for representing closures, but their syntactic overhead is excessive. The call to the standard library's **find_if()** algorithm in Fig. 1 demonstrates. Defining a new class **less_than_i**, constructing an object of that type, and invoking **find_if()** using the object is so verbose that it would be much easier to write, and likely clearer to read, a loop that implements the functionality of the call to **find_if()**. Similar arguments apply to many other functions in the standard library, such as **for_each()** and **transform()**. This is not obvious from textbook examples—simple cases, such as the example in Fig. 1, can be encoded with a set of function objects from the standard library:

```
find_if(v.begin(), v.end(), bind2nd(less<int>(), i));
```

Though less verbose, this is still rather clumsy, and leaves many programmers reaching out for a language manual. The standard function objects like **less** and simple composition functions like **bind2nd** are essentially a small embedded language for defining unnamed functions. One quickly learns that this language can express only a very limited set of functions.

[1] We do not consider the expression tree aspect of C#'s lambda expressions.

```
class less_than_i {
  int i;
public:
  less_than_i(int i) : i(i) {}
  bool operator()(int x) const {return x < i;}
};

vector<int> v;
int i;
  ...
find_if(v.begin(), v.end(), less_than_i(i));
```

Fig. 1. The "function object idiom" for emulating lambda expressions in C++.

The draft C++ standard library offers a more expressive embedded language with its family of **bind** functions [22], [1, Section 20.7.12]. Libraries, such as the Boost Lambda Library [19,18], FC++ [23], and Phoenix [5], take the "embedded language" approach still further. For example, using the Boost Lambda Library, the example in Fig. 1 can be written as

```
find_if(v.begin(), v.end(), _1 < i);
```

The variable **_1** is a predefined name for the first parameter of the unnamed function.

This improves on the original STL's function objects in many ways: the embedded language is almost the same as the rest of C++, the syntax is very concise, and polymorphic lambda functions (as demonstrated by the above example) are supported. Unfortunately, these seemingly elegant solutions suffer from serious problems. For example, erroneous uses of lambda functions often manifest in extremely long and cryptic error messages, all but the simplest expressions require excessive compilation resources (time and memory), and run-time performance can suffer depending on compilers' optimizing ability (e.g., aggressive inlining is crucial). Further, the predefined parameter names (**_1**, **_2**, ...) may feel unnatural to programmers, and because the parameter names have no scope, composing lambda functions is not directly supported. (The latter restriction is not inherent to all library solutions; the FACT! [31] library is one such exception.) The most severe limitation, however, is that the embedded language for defining lambda functions is only "almost expressive enough". As a result, there are many common situations where a subtle behavior of these libraries is very difficult for users to understand, and others where the libraries do not work at all. For example, member access syntax cannot be supported. This means that an expression like **_1.size()**, which would define a unary lambda function that invokes the **size()** member function on the lambda function's argument, is invalid—even though it follows naturally from the general syntax of the embedded language. The expression that works, **bind(&vector<int>::size, _1)**, is much more verbose, has to specify the receiver object's type, and will not work if the member function **size** is overloaded (without suffering the additional syntactic overhead of further annotating its type). In analyzing or correcting an ill-formed lambda expression, neither the C++ type checker nor a debugger is of much help. The libraries discussed above are based on *expression templates* [34], where the original abstraction, the lambda expression, is translated at compile time into a complex structure representing the parse tree of the expression, and thus the abstraction is visible neither for the type checker nor in the generated executable code.

Based on feedback from users of various lambda libraries, it is clearly difficult for programmers to grasp the subtleties of the library-based lambda functions, and a lot of development time is wasted in trying to bend the libraries to do what they cannot do.

In sum, function objects are sufficient for expressing closures in C++ but they are overly verbose to define. Increasingly elaborate library techniques aimed at a more concise notation are useful, but they are also a source of complexity, confusion, and even other forms of verbosity. The popularity of the lambda libraries, despite the struggles programmers experience when using them, shows that lambda functions are a necessary feature for C++. The designs of these libraries demonstrate that lambda functions can be implemented without major changes to the core language, by relying on existing constructs in the language and on its libraries.

3. Lambda expressions

This section describes the design and use of lambda functions informally with the help of a series of examples, and gives a rationale for the design decisions. A detailed specification can be found in the current draft of the C++ standard [1], and in the standards committee's technical reports [16,17].

A C++ lambda expression consists of three main parts: the definition of the parameters, the code of the lambda function, and a specification of how the environment is captured in the closure. In the simplest case, the body of a lambda function contains no free variables and the last part boils down to a simple syntactic indicator "**[]**" that marks the start of the lambda expression. Apart from a missing function name, the syntax of lambda expressions is then similar to that of function definitions. For example, the following lambda function computes the maximum of its arguments:

```
[](int x, int y) { return (x > y) ? x : y; }
```

Definitions of lambda functions are expressions, and thus they can syntactically appear anywhere C++ allows expressions.

3.1. Free variables in lambda expressions

Following established terminology, a variable occurrence that refers to a parameter of a lambda expression is *bound* by that lambda expression. For example, the occurrences of the variables **x** and **y** in the above lambda expression are bound. We can also consider local variables defined within a body of a lambda expression to be bound. All other variable occurrences, i.e., references to variables that are not introduced by the lambda expression, are *free*.

If the body of a lambda function contains free variables, the resulting closure must by some means arrange access to these variables. Furthermore, if the closure outlives the scope of its definition, it must be ensured that the free variables still refer to existing variables, instead of becoming dangling references. C++'s lack of automatic object lifetime management makes this challenging.

Consider a straightforward implementation of closures, where the environment is stored as a list of references to free variables, or alternatively as a single pointer to the activation record of the function where the closure was created, through which the local variables in the enclosing scope can be accessed. In C++, activation records are stored in a stack and a function's record is popped off immediately after the function exits, which makes any references to the activation record dangle. Extending the lifetime of activation records, or individual variables, for example by allocating them selectively on the heap instead of a stack, would too drastically change the basic compilation model and the expected performance characteristics of C++; for example, some form of garbage collection would be required. Instead of such measures, we allow, and require, programmers to explicitly specify what variables are stored and how they are stored in the closure—lifetimes of variables are never extended, but programmers can instruct that their values should be copied into the closure.

Consider the following example:

```
vector<double> v;
double sum = 0;
int factor = 2;
  ...
for_each(v.begin(), v.end(), [](double x) { return sum += factor * x; });
```

Here, **sum** and **factor** are free variables and must thus be stored in the closure by some means. The lambda expression does not specify how and is invalid for that reason. C++ provides two options by supporting both reference and copy semantics. Here it is obvious that the intent is to collect the result to the **sum** variable, and thus the closure should store a reference to **sum**. It would be safe to store a reference to **factor** as well, as its lifetime extends beyond that of the closure, but copying **factor** to the closure would be just as viable.

On the other hand, consider the next example where a lambda function is used as a callback bound to a GUI event:

```
void init_gui() {
  label* lbl = new label("A");
  button* btn = new button("Change label");
  btn-> set_on_push_callback([]() { return lbl-> set_text("B"); });
  ...
}
```

Again, the lambda expression above is invalid, as it does not specify how the closure should store the free variable **lbl**. Assume the closure contains a reference to the pointer **lbl**. When the "on push" event occurs, the **init_gui** method has likely been exited. With C++'s object lifetime rules, **lbl** is no longer alive, and the behavior of the lambda function is undefined. If instead the closure stores a copy of **lbl**, whose lifetime is the same as that of the closure itself, the callback can be safely called after the function **init_gui()** returns. Of course, blindly copying every free variable into the closure would be quite problematic, possibly leading to unintentional object slicing, expensive copying, and other surprises. Moreover, many types are not even copyable.

The syntactic means to control which and how variables are stored in the closure is the *capture list*, a list of variable names within the brackets that indicate the start of a lambda expression. A lambda expression thus has two parameter lists: the function parameter list and the list of free variables. To demonstrate, the two (invalid) lambda functions in the above examples omitted the capture list, and must be rewritten as

```
[&sum, factor](double x) { return sum += factor * x; }
```

```
[lbl]() { return lbl-> set_text("World"); }
```

To keep the syntactic overhead low, the full type of a free variable is not specified, just its name, optionally preceded by **&** to indicate that the closure should store a reference to the variable. In the first function, the closure holds a reference to **sum** but stores a copy of **factor**. In the second function, **lbl** is stored by copy.

Note that the library-based lambda functions offer features to selectively instruct how certain arguments should be stored into a closure, and that approach has proven effective (see **ref** and **cref** functions described in [18], also adopted to the draft standard [1, 20.7.5.5]).

Requiring explicit declaration of the variables that are to be stored in the closure has the benefit that the programmer is *forced* to express his or her intention on what storage mechanism should be used for each free variable. However, in some

cases, such as when many free variables appear in a lambda function, or when they are all captured the same way, it may be cumbersome to list all of them in this way. Since one of the main goals of lambda expressions is conciseness, we also provide two default capture forms marked by a special symbol, either a **&** or **=**, at the beginning of (and possibly in place of) the capture list. When using one of these forms, free variables may go unannounced in the capture list and will be stored using the specified default, either by-reference or by-copy, respectively. The default storage mechanisms can be overridden for particular variables, by requesting the non-default storage mechanism for them in the capture list. To demonstrate, we rewrite the last two lambda functions to take advantage of a default capture mechanism; the semantics of the lambda functions remain unchanged:

```
[&, factor](double x) { return sum += factor * x; }
[=]() { return lbl→ set_text("World"); }
```

3.2. Return type deduction

In the above examples, the return type of the lambda function is not specified; instead, it is deduced from the body of the function as the type of the return expression. This return type deduction is supported whenever the lambda function's body consists of a single return statement. Otherwise we require an explicit annotation from the programmer, because the function may have multiple return points with an ambiguous common type. One could extend and employ the typing rules of the conditional operator [1, 5.16] to attempt to infer the moral equivalent of "least common super type" of all return expressions but, to avoid subtleties, we favored the simple syntactic rule to determine when return type deduction succeeds.

For example, in the following lambda expression, the return type **double** is, and must be, specified explicitly:

```
[](double x) → double { if (x < 0) return 0; else return x; }
```

The return type is placed after the parameter list; C++0x will support a similar syntax for ordinary functions [20,24].

3.3. Semantics via translation

The semantics of lambda functions follow from a translation to function objects. For example, the lambda function **[&sum, factor](double x) { return sum += factor * x; }** gives rise to the class definition

```
class F {
  double& sum;
  int factor;
public:
  F(double& sum, int factor) : sum(sum), factor(factor) {}

  double operator()(double x) const { return sum += factor * x; }
};
```

The lambda expression at its point of definition is replaced with a constructor call that creates the closure object. Here, the free variables **sum** and **factor** are stored in the closure's environment, and are thus passed to **F**'s constructor as **F(sum, factor)**.

A special case in the translation is the treatment of "**this**". A lambda function that occurs within a member function can contain references to **this** as a free variable and include it in the capture list. All references to **this** in the body of a lambda function are translated to use the original value of **this**, as if it appeared directly outside of the lambda function. The specification in the draft standard gives full details [1, 5.1.1].

Specifying the semantics of lambda functions via a translation to function objects is economical: questions on how lambda functions interact with the very large feature set of C++ and how they can be optimized are answered without additional specification work. For example, the scoping rules of a nested lambda function are the same as those of a class nested inside another class's member function. The only exception is the treatment of "**this**", as explained above. Moreover, the translation gives a useful and familiar mental model for programmers. In fact, the direct translation is also the basis for our implementation, though other mechanisms are certainly possible; the draft standard specification is careful to give enough leeway for more optimized implementations. For example, a special provision is included for lambda functions that do not capture any variables by copy [1, Section 20.7.18], which permits the "pointer to the stack frame" implementation discussed in Section 3.1.

3.4. Lambda functions and constness

C++ uses the **const** keyword to denote conceptual constness of objects: a **const** variable cannot be assigned after it is initialized, a **const** member function cannot modify the state of its object, etc. As part of enforcing this concept in the type system, only **const** member functions can be invoked on **const** objects. Lambda functions need a policy for constness, since closures are (function) objects that encapsulate state. Such a policy effectively determines whether or not, in our semantics by translation, the function call operator in the classes for closure objects should be declared **const**. As the foundation for this

policy we look to answer what it means for a lambda function to be **const**, and consider existing conventions for programming with function objects.

Higher-order functions have been in use in C++ for several years with existing tools like function pointers and function objects. Function pointers go all the way back to C, which has no notion of constness. C++ does not have a notion of **const** functions either. Member functions, however, can be declared **const**. The effect is to make the receiver object, the object pointed to by "**this**", **const**, so that the member function cannot modify the object's member variables and thus the object's state. Note that constness of member variables is "shallow": const member functions are free to modify objects referred to by reference member variables. The meaning that we attach to constness of lambda functions follows these principles: a constant lambda function does not modify the state of a closure, where the state of a closure is considered to be the free variables copied into it. Modification of variables referenced from within a closure is allowed regardless of the constness of the closure.

The STL ignores constness of function objects: STL's functions pass all function object parameters by value, accepting either **const** or non-**const** function objects. The STL recognizes that other conventions might be used: all its stateless function objects declare their function call operator **const** to allow calls from contexts where the function object is **const**. Similarly, we wish to provide "constness" for free, allowing programmers to employ lambda functions in **const** contexts without extra annotations: we expect lambda expressions that do not modify the state stored in the closure to be the common case, and so make the closure object's function call operator **const** by default. The compiler thus rejects lambda expressions that attempt to mutate a free variable captured by copy, unless the lambda expression is specially annotated. A lambda function declared **mutable** allows modification of free variables, as shown in the example below. Without the **mutable** keyword, the example lambda function would be rejected for its attempt to modify the **counter** variable in the closure.

```
int counter = 0;
[counter]() mutable { return ++counter; } // OK
```

Finally, the presented point in the design space of constness of lambda functions, the one selected for standardization, is not the only consistent alternative. For example, we have previously advocated attaching no notion of constness to closures—the translation to attain this would combine a **const** function call operator with a **mutable** declaration for each member variable of the closure [16].

3.5. About types of lambda functions

The type of a lambda function is unspecified: the compiler is free to synthesize an arbitrary type to represent a closure. By not introducing a new function type to C++'s type system, we avoid all the complicated interactions that such a type would have with the rest of the language. The downside is that the programmer cannot write out the type of a lambda function. As a result, a non-generic function's parameters or its return value cannot be declared to have the type of a lambda function. Even though generic functions can accept lambda functions as parameters – their parameter types can be type parameters – this is still limiting. For example, template functions cannot be placed in dynamically linked libraries, and defining functions that construct and return lambda functions is impossible.

To provide a non-generic interface for lambda functions, we rely on a known library technique, namely the **function** template [15], [1, Section 20.5] that provides uniform wrapper types to C++'s built-in functions and function pointers, member functions, and function objects. For example, the wrapper type of functions taking two **int** parameters and returning **int** is **function<int(int, int)>**. Any of the above forms of functions are implicitly convertible to an instance of the **function** template, assuming the signature matches. This applies to lambda functions as well—no special arrangements are needed to make lambda functions work with **function**. Moreover, constructing and copying small function objects, which lambda functions commonly are, wrapped into **functions** have negligible overhead.

As an example of this approach, the **ctr** function below returns a lambda function that is wrapped with an instance of the **function** template:

```
std::function<int(int)> ctr() {
  int counter = 0;
  return [counter](x) mutable { return counter += x; };
}
```

Note also that C++0x will support a form of local type inference in variable declarations: the **auto** keyword can be used to indicate that a variable's type should be deduced from the type of its initializer [21]. This enables easy declaration of variables that refer to lambda functions. For example,

```
char sep = ',';
  ...
auto printer = [sep](int x) { cout << x << sep; };
for_each(a.begin(), a.end(), printer);
for_each(b.begin(), b.end(), printer);
```

Note that since **cout** is a global variable, it is not mentioned in the capture list.

```
template<typename I1, typename O1, typename F>
requires InputIterator<I1> && OutputIterator<O1, Callable1<F, I1::value_type>::result_type>
O1 transform(I1 first, I1 last, O1 result, F f) {
  for ( ; first != last; ++first, ++result)
      *result = f(*first);
  return result;
}
```

Fig. 2. A possible implementation of the generic **transform** algorithm using C++0x's constrained templates.

4. Generic lambda functions

Although the standards committee has decided to include only non-generic lambda functions in the next standard C++, we have explored the idea of generic lambda functions that do not require explicit parameter types. There is much desire to see such a feature in C++, in particular since programmers are accustomed to using generic lambda functions that several libraries [5,18,23,31] support. Regardless, the implementation effort was considered too large and the level of current experience too limited for the next standard revision's timetable. Here we explain our design for generic lambda functions, which we plan to bring forward to the standards committee for future consideration.

Note that in current C++, generic lambda functions could be easily implemented via function objects with a templated function call operator member. This approach is no longer feasible with C++0x's constrained templates [13]: except in specific situations, calls from constrained templates to unconstrained templates can exhibit undefined behavior [14, Section 6.9]. Instead, the implementation requires that we infer a signature for a lambda function from the context of its definition.

4.1. Deducing parameter types

C++ was not designed to support type inference, so inferring parameter types from the body of a lambda function is not possible. Instead, we seek to deduce the parameters from the context where the lambda function is defined. A lambda function can be either invoked directly or bound to a variable, such as a formal parameter of a function. Deducing the parameters of the lambda function in the first case is straightforward; one merely takes the types of the parameters to be those of the arguments passed to the lambda function. Of course, calling a lambda function at the point of its definition is seldom useful. Indeed, in the typical case a lambda function is passed as an argument to another function. As we explain above, the type of a lambda function is unspecified and thus a lambda function can only be bound to a function argument whose type is a type parameter: a lambda function's argument types must be deduced from the *constraints* of that type parameter.

To explain how type deduction works, we very briefly discuss C++0x's constrained templates and their modular type checking. For a thorough description of these new features, see [13]. A new language construct, **concept**, is used to express requirements on types. Relevant to lambda expressions is the concept **Callable**. The draft standard library defines a general form, applicable to all arities, of this concept using *variadic templates* [12]—we show a specific version for binary functions:

```
auto concept Callable2<typename F, typename A1, typename A2> {
  typename result_type;
  result_type operator()(F&, A1, A2);
}
```

Omitting details, the constraint **Callable2<F, T, U>**, for example, is satisfied if an object of type **F** can be called with an argument list consisting of two objects that are of, or can be converted to, types **T** and **U**, respectively. The key is to satisfy the function call operator requirement in the **Callable2** concept; in the constraint **Callable2<F, T, U>** the requirement becomes **result_type operator()(F&, T, U)**. The type **result_type** is an "associated type" of the **Callable2** concept, and here denotes the return type of the function call operator.

As an example of how concepts constrain type parameters, consider the implementation for the standard algorithm **transform** shown in Fig. 2. The **InputIterator** constraint that the type parameter **I1** must satisfy justifies that **I1::value_type** is a valid type, and that the uses of the dereferencing (∗), increment (**++**), and inequality (**!=**) operators are valid. **OutputIterator** justifies dereferencing and increment operations. The call **f(∗first)** in the body of the function is justified by the constraint **Callable1<F, I1::value_type>**. Further, **Callable1<F, I1::value_type>::result_type** as the second parameter in the **OutputIterator** constraint guarantees that **F**'s return type is the type of the objects written to the result sequence, and thus justifies the assignment in the for-loop's body. The draft standard contains specifications for the **Callable** [1, Section 20.2], and **InputIterator** and **OutputIterator** concepts [1, Section 24.1].

To illustrate the type deduction process, consider the code

```
vector<T> v; T factor = ...; // T is some type
transform(v.begin(), v.end(), v.begin(), [factor](x) { return x *= factor; });
```

When type checking this call to **transform()**, the type parameter **F** (in Fig. 2) is deduced to be the type of the lambda function, which is some unspecified class type, call it **L**. The type parameter **I1** is deduced to the type of **v.begin()**, which is **vector<T>::iterator**, and thus **I1::value_type** is deduced to **T**. Substituting these to **Callable1<F, I1::value_type>** leads to the constraint **Callable1<L, T>** and to the requirement **result_type operator()(L&, T)** (a unary function whose parameter type is **T**), which we can use to inject the following function call operator into the class representing the closure:

```
R operator()(T x) const { return x *= factor; }
```

The return type *R* is deduced to the type of the expression **x *= factor**; the type of the parameter **x** is now known, so return type deduction is analogous to the case of non-generic lambda functions. As with non-generic lambda functions, return type deduction is restricted to functions whose body consists of a single return statement. The **decltype** operator [20] that queries the type of an expression, part of C++0x, will be useful for explicitly specifying the return type of a generic lambda function.

After the parameters, and possibly the return type, are deduced, type checking the call site of the **transform** function can then continue with type checking the generated function call operator—which corresponds to type checking the lambda expression. If it succeeds, the call **f(*first)** in the **transform** function's body is guaranteed to type check as well.

4.2. About polymorphic lambda functions

C++ function objects are polymorphic if the function call operator is defined as a template. One can pass a polymorphic function object to a function that calls it at several places with different argument types. For example, the following code shows a function that applies a function object to each element of a pair (we could define a similar function for a tuple of arbitrary length), and a call to this function with a polymorphic function object.

```
template <typename A, typename B, typename F>
void for_each_in_pair(const pair<A, B>& p, F f) {
  f(p.first);
  f(p.second);
}

class output {
public:
  template <typename T>
  void operator()(T t) { cout << t; }
};

auto p = pair<int, string>(1, "one");
for_each_in_pair(p, output());
```

Generic lambda functions can retain this polymorphic behavior, though constrained templates naturally limit the valid calls to those justified by the template's constraints. For example, more than one **Callable** requirement can be placed on a single type parameter. The constrained version of **for_each_in_pair** and a call to it with a lambda expression demonstrate:

```
template <typename A, typename B, typename F>
requires Callable1<F, A> && Callable1<F, B>
void for_each_in_pair(const pair<A, B>& p, F f) {
  f(p.first);
  f(p.second);
}

for_each_in_pair(p, [](t) { return cout << t; });
```

The call instantiates the requirements **Callable1<F, int>** and **Callable1<F, string>**, based on which we can generate a closure object with two function call operators, so that calls with an **int** or **string** argument are accepted.

5. Evaluation

As implementations of C++ lambda expressions reach mainstream programmers, code bases for large-scale evaluation of the impact of this new feature will become possible. We conducted experiments on a smaller scale to obtain an early assessment of how programmer productivity is affected. We settle for estimations of improvements in productivity, based on a number of subjective qualities, such as readability, maintainability, and ease of composition. To guide these estimates, we measured the utility of lambda expressions by examining regular C++ code, analyzing the instances where lambda expressions proved useful, and recording the reduction in lines of code (LOC) they facilitated.

Regarding performance, we measured the cost of using lambda expressions compared to more primitive lower-level code as the *abstraction penalty*, defined as the ratio of the execution time of an abstracted implementation over a direct implementation [4, Section D.3]. We also looked at the impact on the size of the executable. It is generally expected that in modern optimizing C++ compilers the use of an STL algorithm and a simple function object incurs no significant performance penalty compared to a handwritten piece of code that performs the same task. As our implementation of lambda functions

is essentially in terms of function objects, we can expect the performance characteristics of lambda functions to be similar to those of function objects.

5.1. Experiment 1: OpenGL demo

For the first experiment, we chose a small piece of software for "active-edge-table" based drawing of two-dimensional polygons. The project consisted of 409 LOCs in 12 header files and 422 LOCs in 7 source files, 831 LOC in total, that we had written before lambda functions were available. We rewrote the code in the project with lambda functions in our toolbox, and analyzed the effect.

Impact on productivity. In all, there were eight opportunities to use lambda expressions. One of these was a nested lambda expression, so we used a total of nine lambda expressions. All of the uses were in calls to STL algorithms, one to **remove_if** and eight to **for_each**. In three cases the calls to the STL algorithms were already in place, and we merely replaced a handwritten function object with a lambda expression. In the remaining cases we replaced a loop with a call to an STL algorithm and a lambda expression. Small reductions in code size were observed: three replacements cut one LOC, one replacement two LOCs, and the case of **remove_if** cut four LOCs of the program.

One benefit of the proposed built-in lambda functions, over library solutions, is that they support arbitrary nesting. In this regard, of particular interest was one case in which a nested lambda function replaced both a loop structure and a function object. The resulting code was as follows:

```
for_each(aet.begin(), aet.end(),
        [](list<Edge*>& ael) { for_each(ael.begin(), ael.end(), [] (Edge*& e) { delete e; }); }
    );
```

Impact on performance. The two revisions of the source code, one with lambda expressions, one without, were compiled at three different levels of optimization using our prototype implementation in GCC [9], and then tested on a 1.33 GHz PowerPC machine with 1.25 GB memory running OS X 10.4.11. With no optimization, lambdas produced an increase in compiled executable size of over 6%, and ran almost 1% slower. As expected, once optimization was turned on, at levels -O1 and -O2, the differences in the executable size and running time were negligible.

5.2. Experiment 2: LyX

For the second, larger experiment, we looked at LyX [33], an open-source WYSIWYM document processor. The code base makes extensive use of the STL, Boost Bind [6], and Boost Lambda [19] libraries, following a number of modern best practices in software, thus making it a prime candidate for refactoring with lambda functions. We selected the top source directory to search for opportunities to use lambda functions. This directory consisted of 108 headers and 104 source files.

Impact on productivity. Although not every opportunity for using a lambda function was exercised, or even discovered, we edited 16 source files where we added 53 lambda expressions. This trimmed 159 lines of code in all. One more header file was also touched, to remove a class definition of a function object. Of the lambda expressions we added, 32 replaced uses of handwritten function objects, 8 replaced handwritten for-loops, and 13 replaced uses of library-based lambda expressions of the Boost Bind and Boost Lambda variety. Two of the replacements cut one LOC, three replacements cut two LOCs, two replacements cut three LOCs, and four replacements cut four LOCs. Fourteen classes that implemented function objects were removed, saving 144 lines in class declarations.

There were a few details worth noting. In one of the more significant instances, a handwritten for-loop was refactored into a call to the standard **transform** algorithm, saving four lines and notably enhancing readability. Also, quite a few of the existing calls were to STL algorithms like **find_if**, **remove_if**, and **for_each**. Such utilization of higher-order functions is normally uncommon because of the high barrier to entry that function objects present. Lambda expressions improved these uses, and opened doors for more. Finally, there were eight cases in which a lambda expression actually increased the number of lines of code at the call site, but was justified by allowing the removal of a function object class definition of significantly greater length at a distant source location. The most extreme case of this phenomenon was a lambda expression that added four lines, but replaced a function object whose class definition occupied twelve lines and was used only once in the program.

6. Related work

Lambda functions and closures are a well-studied topic. Various languages have taken their own unique approaches to supporting them, with different advantages and disadvantages. C++, too, has seen proposals in this design space. Nested functions – functions defined within other functions – have been proposed [3,28]. These proposals predated templates, and they did not support copying local variables into closures, being thus unusable outside their defining scope. More recently, Samko [27] proposed lambda functions with a rather similar approach to specification and implementation as that of ours, but did not discuss type deduction or integration with constrained templates. The design space of lambda functions for C++ was explored in [35], on which the presented work builds.

Of other approaches, we focus on object-oriented languages utilizing the correspondence between objects and closures. Java's [11] *anonymous inner classes*, by allowing class definitions as expressions, can be used to mimic lambda functions. Free variables are limited to refer to local variables declared **final**. Anonymous inner classes are powerful, with most of the expressiveness of a class, but as a consequence are notably more verbose than lambda expressions. In addition to anonymous inner classes, Java seems to be moving towards direct syntax for lambda expressions and closures [2]. C# [25] progressed to lambda expressions via first adding anonymous functions (delegates). Our approach is similar to that of C# in that it is based on a translation to an existing language construct (function objects for C++, delegates for C#). The approaches differ mainly in the polymorphic behavior of lambda functions. A polymorphic lambda function in C++ can be bound to a type parameter, and its parameter types are deduced from the constraints of that parameter. As discussed in Section 4.2, multiple call signatures can be deduced in this way. In C#, type inference occurs when binding a lambda expression to a delegate type, which gives the lambda function exactly one call signature. C++ lambda functions have no equivalent to C#'s mechanism to access the lambda functions' expression trees. Other notable object-oriented languages leveraging on the closures as objects approach include Eiffel with its *Agents* [7, Section 7], and of course Smalltalk with *blocks* [10].

The reported work also draws from the experiences gained with template libraries that emulate lambda expressions, and shares with these libraries the basic approach of representing closures as function objects. Several such libraries exist, including the Boost Lambda [18], the FC++ [23], and the FACT! [31,30] libraries, developed independently, as well as the more recent Phoenix library [5], that seeks to improve the applicability and extensibility over what is offered by Boost Lambda and FC++. The Boost Bind library [6] includes a subset of the above libraries, most of which is part of the C++ standard library draft.

7. Conclusions

Lambda functions are very useful, even necessary, for effective use of modern C++ libraries. That C++ does not directly support lambda functions has led to a series of library solutions that emulate them. The libraries are a notable improvement, but they are still severely inadequate.

This paper presents a design of lambda functions for C++. The proposed lambda expressions are concise and expressive. They allow full control of how free variables are stored in closures—which is necessary for the safe use of lambda functions in a language without automatic object lifetime management. We implemented the proposed design as an extension to GCC, used it to rewrite parts of a few existing C++ code bases to take advantage of lambda functions, and analyzed the results to estimate the impact of lambda functions on programming productivity. These experiments indicated that lambda functions provide clear benefits. For qualitative evaluation by others, we refer to an early experience report with one vendor's implementation [32], that predicts lambda functions to have a major impact on modern C++ programming practices.

Our implementation, as well as the draft standard specification, currently supports lambda functions with explicitly typed parameter lists; we are working on full support for polymorphic lambda functions and parameter type deduction, as described in this paper. Polymorphic lambda functions are a likely candidate for inclusion in the standard at a later time, and this remains as our future work.

Lambda expressions as presented in this paper have already gained acceptance: the specification is included in the draft of the next revision of ISO standard C++ [1], and several compiler vendors are taking a head start on implementing lambda functions according to our specification.

Acknowledgments

Many have helped in arriving at the design presented in this paper, including Jeremiah Willcock, Douglas Gregor, Lawrence Crowl, and Bjarne Stroustrup. Suggestions from several members of the C++ standards committee, including Clark Nelson, Herb Sutter, Jason Merrill, Jonathan Caves, and Daveed Vandevoorde have had an impact on the syntax and other details of lambda functions. This work was partially supported by NSF grant CCF-0541014.

References

[1] P. Becker, Working Draft, Standard for Programming Language C++, Technical Report N2798=08-0308, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, Oct. 2008. www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf.
[2] G. Bracha, N. Gafter, J. Gosling, P. von der Ahé, Closures for the Java programming language (v0.5). www.javac.info/closures-v05.html.
[3] T.M. Breuel, Lexical closures for C++, in: USENIX C++ Conference, October 1988, pp. 293–304. people.debian.org/~aaronl/Usenix88-lexic.pdf.
[4] Technical Report on C++ Performance, Technical Report N1487=03-0070, ISO/IEC JTC1, Information Technology, Subcommittee SC 22, Programming Language C++, 2003.
[5] J. de Guzman, Phoenix v1.2.1, Boost, Sept. 2002. www.boost.org/libs/spirit/phoenix/index.html.
[6] P. Dimov, The Boost Bind Library, Boost, 2001. www.boost.org/libs/bind.
[7] ECMA International, Standard ECMA-367: Eiffel Analysis, Design and Programming Language, 2nd ed., 2006.
[8] ECMA, ECMAScript Language Specification, 3rd ed., 1999, www.ecma-international.org/publications/standards/Ecma-262.htm.
[9] C++0x Language Support in GCC. www.gnu.org/software/gcc/projects/cxx0x.html#lambdas.
[10] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA, 1983.
[11] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, 3rd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.
[12] D. Gregor, J. Järvi, Variadic templates for C++, in: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, ACM Press, New York, NY, USA, 2007, pp. 1101–1108.

[13] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, in: Proceedings of the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '06, ACM Press, 2006, pp. 291–310.
[14] D. Gregor, B. Stroustrup, J. Widman, J. Siek, Proposed wording for concepts (revision 6), Technical Report N2676=08-0186, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2008. www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2676.pdf.
[15] D. Gregor, Boost.Function, Boost, 2001. www.boost.org/doc/html/function.html.
[16] J. Järvi, P. Dimov, J. Freeman, Constness of Lambda Functions (Revision 1), Technical Report N2658=08-0168, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, June 2008. www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2658.pdf.
[17] J. Järvi, J. Freeman, L. Crowl, Lambda functions and closures for C++ (Revision 4), Technical Report N2550=08-0060, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, Feb. 2008. www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf.
[18] J. Järvi, G. Powell, A. Lumsdaine, The Lambda Library: Unnamed functions in C++, Software—Practice and Experience 33 (2003) 259–291.
[19] J. Järvi, G. Powell, The Boost Lambda Library, 2002. www.boost.org/libs/lambda.
[20] J. Järvi, B. Stroustrup, G.D. Reis, Decltype (revision 5), Technical Report N1984=06-0054, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, April 2006.
[21] J. Järvi, B. Stroustrup, G.D. Reis, Deducing the type of variable from its initializer expression (revision 4), Technical Report N1984=06-0054, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, April 2006.
[22] J. Järvi, C++ function object binders made easy, in: Proceedings of the Generative and Component-Based Software Engineering'99, in: Lecture Notes in Computer Science, vol. 1799, Springer, Berlin, Germany, 2000, pp. 165–177.
[23] B. McNamara, Y. Smaragdakis, Functional programming with the FC++ library, Journal of Functional Programming 14 (4) (2004) 429–472.
[24] J. Merrill, New function declarator syntax wording, Technical Report N2541=08-0051, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2008.
[25] Microsoft Corporation. C# version 3.0 specification, May 2006. msdn.microsoft.com/vcsharp/future/default.aspx.
[26] Python Software Foundation. Python 2.4.1 documentation, March 2005. www.python.org/doc/2.4.1.
[27] V. Samko, A proposal to add lambda functions to the C++ standard, Technical Report N1958=06-0028, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2006. www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n1958.pdf.
[28] J.M. Skaller, F. Henderson, A proposal for nested functions, Technical Report N0295=93-0088, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 1993. www.open-std.org/jtc1/sc22/wg21/docs/papers/1993/N0295.pdf.
[29] A. Stepanov, M. Lee, The Standard Template Library, Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, 1994. www.hpl.hp.com/techreports.
[30] J. Striegnitz, S.A. Smith, An expression template aware lambda function, in: First Workshop on C++ Template Programming, Erfurt, Germany, Oct. 2000. http://oonumerics.org/tmpw00/.
[31] J. Striegnitz, The FACT! library home page, 2000. http://www.fz-juelich.de/jsc/FACT/start/index.html.
[32] H. Sutter, Trip Report: February/March 2008 ISO C++, 2008. herbsutter.wordpress.com/2008/03/29/trip-report-februarymarch-2008-iso-c-standards-meeting.
[33] The LyX Team, LyX — The Document Processor, version 1.5.6, July 2008. www.lyx.org.
[34] T. L. Veldhuizen, Expression templates, C++ Report 7 (5) (1995) 26–31. Reprinted in: Stanley Lippman (Ed.), C++ Gems.
[35] J. Willcock, J. Järvi, D. Gregor, B. Stroustrup, A. Lumsdaine, Lambda functions and closures for C++, Technical Report N1968=06-0038, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2006. www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf.