Side effects and partial function application in C++

Jaakko Järvi[1] and Gary Powell[2]

[1] Turku Centre for Computer Science, University of Turku, Finland
`jaakko.jarvi@cs.utu.fi`
[2] Sierra On-Line Ltd., Seattle WA, USA
`gary.powell@sierra.com`

Abstract. Function calls that cause side effects to actual arguments are common and well accepted in OO languages. When a function is applied only partially, such side effects, or preventing them, may lead to surprises. The paper gives examples of this, and describes different approaches to deal with side effects. The main focus is on C++ and a partial function application implementation as a template library. The paper explains what are the relevant points for controlling side effects to the arguments of a partially applied function in such a library. As a practical result, the paper presents a mechanism that can be used with function calls to control side effects to each function argument individually.

1 Introduction

Lambda functions and partial function application are common features in functional programming languages. Now these features are gradually finding their way to object oriented (OO) languages as well. Eiffel has an extension called *agents* [1, 2], which allows any argument of any function call to be replaced with a question mark. Such a function call creates an *agent*, which is basically a partially applied function. In the *Sather* language [3], a corresponding feature is known as *closures*. For C++, a couple of template based extension libraries have been developed during the recent years, among them FC++ [4], FACT [5] and Lambda Library (LL) [6], which all provide forms of lambda functions for C++.

OO languages as platforms for these features are quite different from functional languages. Particularly, OO languages typically allow functions to have side effects on their arguments while functional languages do not. Furthermore, not all function arguments are treated equally in OO languages. When a member function of an object is called, that object has a special status compared to the other function arguments. Regarding these considerations, we are asking what should the form of manifestation of lambda functions and partial application be in OO languages.

We show that the question is centered around argument passing mechanisms, whether to pass arguments by value or by reference, and by what kind of reference. Passing an argument into a function by reference allows the function to change the value of the argument as a side effect of evaluating the function. Such side effects are more or less inherent in languages like C++ or Eiffel; member functions typically alter the state of the object, the side effect to the left-hand argument of an assignment operator is usually the reason why to call an assignment operator etc. Consequently, side effects on arguments make sense. But what about side effects on the arguments when the function is only applied partially? The paper shows that the issue is more complicated than it may look at the first glimpse. Partly this is common for OO languages, but partly the complications stem from C++ and its peculiarities.

Eiffel agents are implemented as a true language extension, which gives a total freedom over the syntax and semantics of the features, whereas the experimental C++ libraries stay inside the language. This obviously places some restrictions on the implementation of the features and makes the syntax a bit crippled, but whether we stay inside the language or provide a true extension, we believe that the same questions are still relevant. Nevertheless, the paper mainly discusses a C++ implementation of partial application.

Finally, as practical results we show that it is possible to implement partial function application in C++ either to allow or disallow side effects, and even do this selectively depending on the properties of the underlying function. Further, we show a mechanism that gives the programmer the control of allowing or disallowing side effects for each argument separately.

1.1 Partial application syntax in C++

The syntax of partial application in C++ can be defined in many ways. In this paper we use the syntax of the Lambda Library [6]: To apply a function partially, the function and its arguments are enclosed in a generic *bind* function. In such a function call, special *placeholder variables* `_1`, `_2`, ... can be used as actual arguments to state that an argument is left open. A partially applied function results in a new function, and `_1` refers to the first argument of this function, `_2` to the second, etc. For example:

```
bind(f, a, b, _1, _2, e)
```

takes a 5-argument function `f`, binds the first, second and fifth argument to `a`, `b` and `e`, respectively, and results in a two-argument function. Further, the expression

```
bind(f, a, b, _1, _2, e)(c, d)
```

calls this two-argument function and invokes the original function `f` as

```
f(a, b, c, d, e).
```

For operators there is no special function or keyword; a placeholder variable as an operand implicitly delays the evaluation of the expression it resides in: For example, with this syntax the expression

```
2 * (_1 + _2)
```

is interpreted as

$\lambda\ x\ y\ .\ 2(x+y)$

1.2 About implementing partial application in C++

Lambda functions and partial application are more or less the same thing when implemented as a C++ template library. The syntax is different due to the syntactic sugar in calls to operator functions, but the internal implementation is the same: a lambda function, or a partial application, creates a kind of a closure object that encloses the underlying function or operator and the arguments that were provided. The function call operator of this object can then be called with the remaining arguments. Considerably simplified, the above example of partially applying the function `f` creates an object of type:

```
struct lambda_functor {
  function f;
  arguments a, b, e;

  template<class T1, class T2>
  result_type operator()(T1 _1, T2 _2) { f(a, b, _1, _2, c); }
};
```

In C++ the question of allowing or disallowing side effects boils down to what information about the arguments should be stored in the closure of the function call: the references to, or the values of the actuals at the site of constructing the closure. At the most concrete level, the question is about the types of the arguments `a`, `b` and `c` in the above `lambda_functor` class.

2 Argument binding in C++ standard library

C++ defines two function templates for partial function application: `bind1st` and `bind2nd`. Both take a binary function, bind one of the arguments to a fixed value, and return a unary function object. The C++ standard [7] defines this function object to store the bound argument as a copy. The argument that is left open is passed as a const reference to the function call operator. Hence the intention is that there should be no side effects to the bound argument, nor to the unbound argument supplied later. For example, figure 1 shows a straightforward implementation of `bind1st` as defined by the standard.

As an example of the usage of standard binders, `sum` is a standard function template that creates a binary function object to compute the sum of its arguments (suppose `T` is some type for which the `operator+` is defined):

```
sum<T>()(a, b);           // calls a + b;
bind1st(sum<T>(), a)(b); // calls T(a) + b;
```

While comparing these calls, the only difference is that in the latter call a copy of the bound argument is taken, and the addition operator is called with the copy. Thus, it seems that there is no difference between the outcome of the two expressions above and this is probably what the programmer would expect. Note that it is possible to find cases, albeit somewhat artificial ones, where the outcome

```
template <class Oper> class binder1st
  : public unary_function<typename Oper::second_argument_type,
                          typename Oper::result_type> {
protected:
  Oper op;
  typename Oper::first_argument_type value;

public:
  binder1st(const Oper& x,
            const typename Oper::first_argument_type& y)
      : op(x), value(y) {}
  typename Oper::result_type
  operator()(const typename Oper::second_argument_type& x) const {
    return op(value, x);
  }
};

template <class Oper, class T>
inline binder1st<Oper>
bind1st(const Oper& oper, const T& x) {
  return
    binder1st<Oper>(oper, typename Oper::first_argument_type(x));
}
```

Fig. 1. Standard `binder1st` implementation

is not what was expected. For example, `operator+` might do something peculiar to its arguments, like change a mutable member of `a`. Mostly the standard binders are limited enough to keep the programmer out of trouble.

However, partial application is an appealing technique, particularly convenient with STL algorithms. And when we try to extend the mechanism to cover a larger set of functions, we are faced with the questions posed in the introduction. For example, inspired by the previous example, the next thing a programmer might want to write, could be (note that this example doesn't work with the current standard binders):

```
sum_assign<int>(a, b);             // calls a += b
bind1st(sum_assign<int>(), a)(b); // calls int(a) += b
```

Now we can see that it makes a difference how binders bind the arguments.

2.1 Binding and member functions

The final definition of binders is an open issue in the standardization process. A proposed change to the standard library, raised by Stroustrup [8], suggests that the `binder1st` and `binder2nd` templates should define another function call operator that takes a non-const reference argument:

```
typename Oper::result_type
operator()(typename Oper::second_argument_type& x) const {
  return op(value, x);
}
```

This would mean that side effects via the unbound argument would be allowed. The rationale behind the suggestion is to make something like this to work:

```
class turtle {
  public:
    void move (step s);
};

void move_all(list<turtle>& ls, const step& s) {
  for_each(ls.begin(), ls.end(),
           bind2nd(mem_fun_ref(&turtle::move), s));
}
```

The intention is to call `t.move(s)` for each element `t` in the list `ls` (`mem_fun_ref` encloses a pointer to a member function into a *bindable* function object). Without the suggested change, however, the code is erroneous. The call to `bind2nd` creates a function object with a function call operator prototype defined as:

```
void operator()(const turtle& t);
```

But then `move` is a non-const function and it cannot be called with a reference to a const turtle object. The effect of the proposed change would be to add the function call operator `operator()(turtle& t)` into the binder object, which would make the code work.

The previous example showed a function to move several turtles in a collection one step forward. How about a function to move one turtle several steps:

```
void move_many_steps(turtle& t, list<step>& ls) {
  for_each(ls.begin(), ls.end(),
           bind1st(mem_fun_ref(&turtle::move), t));
}
```

Against what one might expect, this piece of code has no effect on the turtle `t` at all. Binding `t` means taking a copy of it, hence the target of the `move` calls is a copy of `t`, which is constructed when the binder function object is created, and gets destructed after the `for_each` invocation.

There's a myriad of additional details. For example, if we use `mem_fun` and a pointer argument instead of `mem_fun_ref` and a reference, the member function of the original turtle object is invoked:

```
void move_many_steps(turtle* t, list<step>& ls) {
  for_each(ls.begin(), ls.end(),
           bind1st(mem_fun_ref(&turtle::move), t));
}
```

On the other hand, the compilation fails altogether if the list of steps is taken as a const reference (which would be a natural parameter type for this function):

```
void move_many_steps(turtle& t, const list<step>& ls);
```

Furthermore, if the argument to the `turtle::move` function was `const step&` instead of `step`, compilation would fail as well. See [9] for a discussion about the shortcomings of standard binders.

The standard tools for partial function application disallow most of the cases where the interpretation of the partially applied function may not be clear. Still, an unwary programmer may be taken by surprise, as the preceding discussion demonstrates.

2.2 Partial application taken further

We're not stuck with the limited partial function application support of the C++ Standard Library. A template library that allows partial application of function pointers, function objects and pointers to member functions up to a predefined arity limit, say for 10-ary functions, was described in [10]. Furthermore, apart from a few exceptions, any overloadable C++ operator can be overloaded to accept partial application. Taking argument binding still further, even control structures and exception handling constructs can be 'applied partially'. [6] Tools like this enable partial application in expressions where side effects occur naturally in ordinary function application. Consequently, the possibility of side effects must be taken into consideration. Instead of ignoring the issue, we must determine how to cope with it.

3 Different approaches to side effects

There are three alternatives to deal with side effects to bound arguments in partial function applications:

1. Ignore side effects. Take a copy of each bound argument and store the copy in the function object. If there are side effects, the code compiles but the side effects affect the copies. This is the approach used in the C++ Standard Library.
2. Deny side effects. Flag any expressions that might have side effects to bound arguments as errors. The Standard Library follows this approach to some extent as well.
3. Allow side effects. Store a reference to each bound argument in the function object.

A combination of these three alternatives where the semantics is dependent on the properties of the partially applied function, is also possible. Additionally, each approach can be complemented with a mechanism that gives the user the control to bypass the default behavior.

We take three example function calls which, in full application, cause side effects to their arguments, and discuss applying them partially in the light of the above three alternatives. First an operator call where the side effect to the variable `i` is the only reason to make the call:

```
int i; int j; ... i += j;
```

Leaving the left-hand operand unbound creates a unary function that increments its argument by the value of the bound argument. If the right-hand operand is left unbound, we end up with a unary function incrementing the bound variable by the argument of this unary function. For example:

```
vector<int> v; int j;
...
for_each(v.begin(), v.end(), _1 += j);
for_each(v.begin(), v.end(), j += _1);
```

The first case is straightforward, and the natural interpretation is that each element of vector `v` is incremented by the value of `j`. The second case is trickier. The programmers intent is most likely, that the sum of the elements of `v` is computed in `j`. This is also the effect in the 'allow side effects' approach. In the 'ignore side effects' alternative, however, a copy of `j` gets incremented leaving `j` intact, which is undoubtedly confusing. A safe, but more restrictive solution would be to flag the expression as an error and ban it altogether.

The second example moves turtles again. In the example of section 2, we used standard binders, here we use the binders from LL:

```
vector<turtle> tv; turtle t; vector<step> sv; step s;
...
for_each(tv.begin(), tv.end(), bind(&turtle::move, _1, s));
for_each(sv.begin(), sv.end(), bind(&turtle::move, t, _1));
```

Bear in mind that `move` is a non-const member function of `turtle` and most likely modifies its state. Again, the first `for_each` invocation is quite clear, calling `x.move(s)` for each turtle `x` in the vector `tv`. The intention of the second one is to call `t.move(y)` for each step `y` in the vector `sv`. 'Allow side effects' case performs just this, whereas against the programmers intent, the 'ignore side effects' case keeps moving a copy of the original turtle `t`. In this example as well, a safe approach would be to make the second call fail at compile time.

The third example considers calls to freestanding functions, or function pointers.

```
void add_to(int& i, const int& j) { i += j };
```

Note that the second argument to the `add_to` function is a reference. In the function body, `i` and `j` can thus be aliased.

```
vector<int> v;
...
for_each(v.begin(), v.end(), bind(add_to, _1, 5));
for_each(v.begin(), v.end(), bind(add_to, _1, v[0]));
```

The first `for_each` call is again a clear case. The bind call creates a unary function object that increments its argument by 5. In the second case it makes a difference whether we store the bound argument `v[0]` as a copy or as a reference. If a copy is taken, each element in `v` is incremented by the value of the first element. If the binding is by reference, the effect is different. The first iteration increments the first element by itself, which doubles the value of the bound argument. Consequently, each successive element is incremented by twice the original value of the first element, which is probably not what the programmer wanted. Note that the same problem is apparent in our first example as well:

```
for_each(v.begin(), v.end(), _1 += v[0]);
```

The above examples demonstrate that neither the 'ignore side effects' nor 'allow side effects' approach leads to the most natural outcome in all cases. In fact, both approaches allow expressions that are somewhat counterintuitive and thus may lead to errors that are hard to find.

In the preceding examples, the partially applied functions are created as temporary objects. The types of partially applied functions tend to be rather complex, and as C++ has no `typeof` operator or alike, it is difficult to directly declare a variable that would hold such a type. It is however possible, and with a set of helper templates it can be made relatively convenient, as demonstrated by the FC++ library [4]. This means that the function object created as a result of a partial application can be stored into a variable, and evaluated later, in an other expression. This brings up another point into the discussion. If a bound argument is stored as a reference, the argument may not exist any more at the evaluation site, leading to a dangling reference.

Eiffel approach The parameter passing mechanism in Eiffel is always *call-by-value*. But variables in Eiffel hold references to objects making the parameter passing mechanism in effect *call-by-reference* (this is not true with *expanded types*, such as INTEGER, REAL etc.). Eiffel agents, that is partially applied functions, obey the normal parameter passing rules, and thus Eiffel takes the 'allow side effects' approach. Further, variables that refer to agent objects are allowed. Hence, the agent construction and agent evaluation sites can be very different. However, dangling references cannot occur due to garbage collection.

4 Implementing the different approaches in C++

Partial application in C++ can be implemented using *expression templates* [11]. A partially applied function is an *expression object* that stores the bound arguments and the underlying function. Further, in its template arguments, the expression object encodes information about the positions, types and number of bound and unbound arguments. The LL calls these expression objects *lambda functors*.

The operator syntax for partial application is achieved by overloading operators for placeholder types that represent the open argument slots, and for lambda functor types. Partial application of function pointers, function objects and pointers to member functions is achieved by overloading the `bind` functions. As an example, figure 2 shows one overloaded `bind` function template and one of the specializations of the `lambda_functor` template. The return types of `bind` functions are instances of lambda functors. Note, that this is only an outline of the real library code, many details have been omitted.

```
template<class Function, class Arg1, class Arg2>
lambda_functor<
  tuple<
    type_mapping<Function>::type,
    type_mapping<Arg1>::type,
    type_mapping<Arg2>::type
  >,
  compute_arity<tuple<Function, Arg1, Arg2> >::value
>
bind(Function f, Arg1 a1, Arg2 a2) {
  return
    lambda_functor<
      tuple<
        type_mapping<Function>::type,
        type_mapping<Arg1>::type,
        type_mapping<Arg2>::type
      >,
      compute_arity<tuple<Function, Arg1, Arg2> >::value
    > (make_tuple(f, a1, a2));
};

template<class Args>
class lambda_functor<Args, 2> {

  Args args;
public:
  template<class A, class B>
  typename return_type_traits<Args, A, B>::type
  operator(A a, B b) {
    return substitute_arguments_and_evaluate(args, a, b);
  }
  ...
}
```

Fig. 2. Three argument bind function template and the binary lambda functor template.

The task of the `bind` function is simply to group the arguments into a *tuple*, and construct the lambda functor. Tuple is a template class that can hold an arbitrary number of elements of arbitrary types. Tuple types, and their implementation as a template library is discussed in [12].

The `lambda_functor` template has two arguments. The first is the argument tuple type, the elements of which are the types of the arguments to the `bind` function. The second is the arity of the functor, which is a property computed with a traits class from the first template argument, basically by counting the unbound arguments. The `lambda_functor` template has a specialization for each supported arity, providing a function call operator with that arity. This function call operator substitutes the actual arguments for the placeholders and evaluates the underlying function with this combined argument list. How this works exactly, is explained in [6], as well as the mechanisms for deducing the return type of the operator. The important points to consider here are:

1. The argument types of the `bind` function.
2. The types of the arguments in the argument tuple.
3. The argument types of the lambda functor's `operator()`.

These are the points that control how side effects are handled.

4.1 Bind function argument types

Partial function application implemented as a C++ template library can only support functions up to some predefined arity limit. The `bind` functions must be defined for each supported arity. These functions are obviously templated, and their calls rely on the compiler deducing the template argument types. The basic choices for defining the argument types are either as const references:

```
template<class F, class A1, class A2>
ret_type bind(const F& f, const A1& a1, const A2& a2);
```

or as non-const references:

```
template<class F, class A1, class A2>
ret_type bind(F& f, A1& a1, A2& a2);
```

The third option would be to not use references at all, but rather take the arguments as copies. That is an obvious way to prevent any side effects, but it would also prevent passing non-copyable arguments and possibly introduce unnecessary copying of objects. Hence, we focus on the above two alternatives.

Overloading based on the type of the function (the first argument) is possible, and can in some cases be used to guide which mechanism to use for certain arguments (see *Overloading bind functions* at the end of this section), but in general we cannot make any kind of distinction between the arguments. It is not known beforehand what are the prototypes of the functions that are applied partially, and thus either one of the options must be chosen for all arguments.

As an example, consider the function:

```
void foo(int& i, const double& n);
```

The following code shows a valid call to this function:

```
int a;
...
foo(a, 3.14);
```

We then examine a partial application of `foo`, now binding all arguments:

```
bind(foo, a, 3.14);
```

Suppose first, that we use the first `bind` function definition, the one with the const parameters. Consider the second argument `a`, which corresponds to the first argument of `foo`. The type of this argument in `foo` is `int&`, but in `bind` the deduced type becomes `const int&`. This means, that `a` is now regarded as const in the body of the `bind` function, as well as in the lambda functor that is created. This means that we cannot call `foo` with `a` from within the lambda functor unless we make a non-const copy of it. Or cast away constness, which could break const correctness as there is no guarantee that the actual argument wasn't const to begin with.

Next, consider what would happen if we used the second version of `bind`. Now the type of the second argument would correctly be deduced to `int&`. What about the third argument then? The type of 3.14 is `double` (not `const double`), which means that the deduced argument type becomes `double&`. But 3.14 is a temporary object, and as according to the C++ standard, a reference cannot be bound to a non-const temporary, this is a compile time error.[1]

So basically, it is not possible to create a completely transparent interface for bind functions. Either we have to somehow trick the compiler to accept non-const references through a const interface, or turn temporaries into constant types. Both can be accomplished, but it requires arguments to be wrapped with helper functions at the call site (see section 5).

The latter is almost possible even without modifications to the calls. Temporaries are created as a result of function and operator calls. Hence, by rigorously defining all functions returning temporary objects to return const types, all temporaries would be const. For example:

```
class A;
const A createA() { return A(); }
...
template<class T> void g(T& t);
g(A());
g(createA());
```

[1] Note that a non-const member function of a temporary class object can be called [7, Section 3.10.], which is very similar to binding a reference to a non-const temporary. The purpose of this exception is presumably to allow a chain of calls to member functions (e.g. `a.plus(b).multiply(c);`), but we find the rule still rather inconsistent.

The type of the expression `A()` is `A` and thus the prototype of `g` in the first call becomes `void g(A&)`. As `A()` creates a temporary, the call fails. In the second case, the prototype becomes `void g(const A&)`, and the call is valid.

There is still one more glitch here. We deliberately used a temporary of a class type in the example above. The reason for this is, that non-class type temporaries cannot be const qualified [7, Section 3.10.]. The rationale behind this rule is probably that there is no visible difference between a const temporary, and a non-const temporary for non-class types. But there is a difference in the deduction of template arguments, as was shown above.

Overloading bind functions We only discussed the most general form of the `bind` function above and stated that we have to choose either form of parameter passing for all arguments. However, we can have a bit more control by overloading `bind` for different function forms. For instance, if the target function of the partially applied function is a pointer to a non-const member function, an overloaded function of `bind` can take the *object argument* as a non-const reference, whereas for a const member function pointer, the object argument can be a reference to a const type. By object argument we refer to the argument, which is the target of the member function call. For example, `t` in the expression `bind(&turtle::move, t, _1)`.

Overloading operators for partial application is more flexible. Since each operator has a fixed number of arguments, and established default semantics, the operators can be overloaded to follow these default rules. For example, `operator+=` should be able to modify the first argument, while not the second one. Hence, the operator can be defined to take the first argument as a non-const reference, and the second as a const reference.

4.2 Types of argument tuple elements

Once the arguments have survived the first barrier, the `bind` function call, we need to consider the next point. How do we store them in the argument tuple in the lambda functor. Storing the actual arguments as references means that side effects can occur, while storing the arguments as non-reference types means copying the actual arguments, and hence side effects can occur, but to the copies of the actual arguments. Tuple types are not a limitation here, they can hold references to objects, just as well as the actual objects.

Rather than declaring the types directly as references or as plain types, they are wrapped inside a traits template. The `type_mapping` template serves this purpose in our example lambda functor implementation. This gives us control whether the arguments are stored as copies or as references, and we can even make the decision dependable on the type of the argument. For instance, as arrays cannot be copied, the `type_mapping` template can always map array types to references. Furthermore, as explained in section 4.1. we need to use wrappers to pass non-const references through the `bind` interface. The `type_mapping` traits can be used to retrieve the underlying reference from the wrapped argument (see section 5). For a discussion about type traits in general, see [13].

4.3 Argument types of function call operator

The next thing to consider is the function call operator of the expression object. This is the function typically called from an STL algorithm, and the actuals to the function are the arguments that were left open in the partial application. These argument types control whether side effects are allowed via the unbound arguments. Unlike in the standard binders where these argument types are fixed at time of constructing the expression object, the function call operator is a template and the argument types are deduced when the function call operator is invoked. Hence, we again have two main alternatives for defining the argument types: as references or as references to const.

The section 4.1 discussed the advantages and disadvantages of both alternatives, and most of the same concerns apply here as well. However, allowing side effects for the unbound arguments is maybe slightly less problematic. After all, providing the remaining actual arguments for a partially applied function is just an ordinary function call. Hence, it seems to be more natural to define the function call operators of lambda functors to take their arguments as non-const references, particularly if side effects are allowed for the bound arguments. A problem arises with this approach, if the actual arguments are non-const temporaries (see section 4.1). This can happen if dereferencing an iterator inside the STL algorithm results in a temporary. The problem can be solved, but it requires the whole partial application to be wrapped inside a function that makes the arguments const, and the number of specializations increases exponentially with respect to the number of arguments. STL algorithms, however, supports only nullary, unary and binary function objects, so this is tolerable.

The function call operator in the example lambda functor delegates the task of actually substituting the arguments and evaluating the function forward by calling the function `substitute_arguments_and_evaluate`. This function hides a complex chain of templated function calls where the arguments are passed forward to several functions. We again refer to [6] for the details, but what can be noted is, that all these functions are templates where the argument types are deduced, and they can safely take their arguments as non-const references. Once the arguments are past the first barrier, either the `bind` function or the lambda functor's function call operator, they are not temporaries anymore.[2]

5 Giving control to the client

It is apparent that partial function application implemented as a template library cannot be made entirely transparent. By transparent we mean that the parameter passing mechanism reflects precisely the prototype of the underlying partially applied function. Furthermore, even if this was possible, it is not obvious whether this should be the case; partial application is different enough from a full application to bring up surprises, as discussed in section 3.

[2] Nested partial applications, that is function composition, create temporaries, but these can be handled internally.

Whatever default semantics is chosen, it is possible to provide the programmer with tools to override it. Let us return to one of our previous examples:

```
void add_to(int& i, const int& j) { i += j };
```

Suppose we have `bind` functions that prevent side effects by taking arguments as const references. Depending on the implementation, binding the first argument of `add_to` either fails, or the the potential side effect affects a copy of the actual argument. The programmer may enable the side effect by wrapping the variable with a helper function:

```
vector<int> v; int x;
...
for_each(v.begin(), v.end(), bind(add_to, x, _1));      //fails
for_each(v.begin(), v.end(), bind(add_to, ref(x), _1)); //ok
```

Further, we showed the example where the intent was to increment all elements in a vector with the value of the first element:

```
for_each(v.begin(), v.end(), bind(add_to, _1, v[0]));
```

If `bind` functions store the arguments as copies, this is exactly what the code does. We also showed that if arguments are stored as references, the outcome is something less intuitive. However, if the side effect is what the programmer wants, even in the case where arguments are stored as copies, this can be achieved by explicitly wrapping the argument with `ref`:

```
for_each(v.begin(), v.end(), bind(add_to, _1, ref(v[0])));
```

Analogously, we can provide means to state that the argument should be stored as a copy, instead of a reference. Consider the 'turtle moving' example in section 3 and suppose that the object argument is stored as a reference:

```
turtle t; vector<step> sv;
...
for_each(sv.begin(), sv.end(), bind(&turtle::move, t, _1));
for_each(sv.begin(), sv.end(), bind(&turtle::move, plain(t), _1));
```

The first `for_each` invocation calls `t.move(s)` for each element of `sv`, while the second operates on a copy of `t` and has no effect on `t`.

5.1 Implementing argument wrappers

The argument wrappers can be implemented by creating a disguise for the true type of the argument. The wrapper object holds a reference member to the actual argument, and has an appropriately defined conversion operator for getting back to the original type. Such an object can pass a non-const reference through a const qualified parameter, or a reference through a call-by-value barrier. The following code shows the definitions of the wrapper class and the `ref` function template:

```
template<class T>
class reference_wrapper {
  T& x;
public:
  explicit reference_wrapper(T& t) : x(t) {}
  operator T&() const { return x; }
};

template<class T>
inline const reference_wrapper<T> ref(T& t) {
  return reference_wrapper<T>(t);
}
```

Wrapping a variable with `ref` creates a `reference_wrapper` object containing a reference to the variable. This object can be passed to the `bind` function where the wrapping is undone with traits templates. The `type_mapping` template (see section 4.2) has specializations for this purpose:

```
template<class T> type_mapping<reference_wrapper<T> > {
  typedef T& type;
};
```

This specialization converts the argument type back to the original reference type, and the reference gets stored in the lambda functor's argument tuple. For example:

```
vector<int> v; int x = 3;
for_each(v.begin(), v.end(),
         bind(add_to, ref(x), _1)); //ok
```

First the call to `ref(x)` returns a `reference_wrapper<int>` object which is passed to the `bind` function (as `const reference_wrapper<int>&`). The traits template (`type_mapping`) maps the reference wrapper back to `int&` which is the type of the bound value to be stored. To initialize this value, the conversion operator to `int&` of the `reference_wrapper<int>` class returns the reference to the original variable `x`. Hence, all traces of tweaking the reference into the expression object are gone by the time the `for_each` algorithm calls the partially applied function, and `add_to` gets called with a reference to the variable `x`.

Additionally, we've defined a `cref` function for wrapping references to constants:

```
template<class T>
inline const reference_wrapper<const T> cref(const T& t) {
  return reference_wrapper<const T>(t);
}
```

We do not show the implementation of the `plain` wrapper function mentioned in the turtle example in section 3. It works much the same way except that instead of returning a reference to the variable, the wrapper makes a copy of it when the conversion operator is called. Note that we do not need the `plain` wrapper to circumvent an unsuitable parameter passing mechanism, but only to instruct the tuple that a copy of the bound argument should be stored where a reference would be stored by default.

6 Conclusions

Side effects to the arguments of a function are common in a typical object oriented program. Particularly, the state of the object argument in a method invocation often changes. This is a feature taken for granted and is well accepted and natural. Adding partial function invocation to an object oriented language blurs the picture, and it is not instantly clear whether side effects are that natural anymore.

This paper identified three alternatives to deal with side effects to the bound arguments in partially applied functions: to allow, to silently ignore or to deny expressions with side effects entirely. We discussed the problems with C++ in detail showing both examples where side effects may take the programmer by surprise and examples where they are intuitive and natural. None of the approaches is a perfect solution and it is also possible to treat different types of functions differently, e.g. side effects can be allowed for the object argument in a method invocation, while not for the remaining arguments.

Regarding C++, there are further details that prevent a clean solution without modifications to the core language. We described what these details are and where the problems in C++ implementation of partial application stem from. Particularly, not being able to const qualify temporaries that are not of class types is a nuisance. At least for C++, we have to settle for what is a less than optimal solution, recognizing that beginning programmers may still have some trouble writing expressions involving complex partial function applications. Additionally, we showed how to implement a mechanism that allows the programmer to selectively state whether side effects to a certain argument are wanted or not.

7 Acknowledgments

We are grateful to Harri Hakonen for his expertise and help with Eiffel.

Gary's: Jaakko has wrestled with this problem well before I became interested in lambda expressions and hence this paper is mostly his work. I am privileged to work with him and appreciate the opportunities to learn more about the dark corners of C++.

References

1. *Agents, iterators and introspection*, 2000. `http://www.eiffel.com` (information, papers).

2. P. Dubois, M. Howard, B. Meyer, B. Rosenberg, M. Schweitzer, and E. Stapf. From calls to agents. *JOOP*, October 1999.
3. Sather web-site. `http://www.icsi.berkeley.edu/~sather/`, 2001.
4. B. McNamara and Y. Smaragdakis. Functional Programming in C++. In *Proceedings of The 2000 International Conference on Functional Programming (ICFP)*. ACM, 2000. `http://www.cc.gatech.edu/~yannis/fc++`.
5. J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 2000. `http://oonumerics.org/tmpw00/`.
6. J. Järvi and G. Powell. The Lambda Library : Lambda abstraction in C++. Technical Report 378, Turku Centre for Computer Science, November 2000.
7. International Standard, Programming Languages – C++,
ISO/IEC:1488, 1998.
8. ISO/IEC JTC1/SC22/WG21 : international standardization working group for the programming language C++. The C++ Standard Library Issues List, revision 17. `http://anubis.dkuug.dk/JTC1/SC22/WG21/`, 2001.
9. V. Simonis. Adapters and binders - overcoming problems in the design and implementation of the C++-STL. *ACM SIGPLAN Notices*, January 2000.
10. J. Järvi. C++ function object binders made easy. In *Proceedings of the Generative and Component-Based Software Engineering'99*, volume 1799 of *Lecture Notes in Computer Science*, 2000.
11. T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.
12. J. Järvi. Tuple types and multiple return values. *C/C++ Users Journal*, 2001. To appear (August).
13. J. Maddock and S. Cleary. C++ Type traits. *Dr. Dobb's Journal*, October 2000.