Using Accessory Functions to Generalize Dynamic Dispatch in Single-Dispatch Object-Oriented Languages

David Wonnacott
Department of Computer Science
Haverford College
Haverford, PA 19041
davew@cs.haverford.edu
http://www.cs.haverford.edu/people/davew

Abstract

Object oriented languages generally include some form of *dynamic dispatch*; that is, in the absence of precise compile-time type information, they perform a run-time selection of the appropriate function body (or *method*) from a set of candidates. Existing single-dispatch languages restrict dynamic dispatch to the object receiving the message.

Such languages exhibit a conflict between the goals of providing an extensible a set of types and providing an extensible the set of operations that can be performed on these types. We show that this conflict is a consequence of the restriction of dynamic dispatch to the receiver object. We also demonstrate that this conflict can be resolved by introducing a generalized form of single dispatch (thus avoiding the complexity of multiple dispatch). On this evidence, we argue that dispatch technique should be decoupled from membership in a class and access to its representation.

1 Introduction

Inheritance helps a programmer create a set of classes for distinct yet similar types of objects. When inheritance is used for subtyping [13], the superclass lists the messages that must be handled by its subclasses. Code that uses superclass messages will work for subclass objects, as long as these objects respond properly to all messages listed in the superclass. Such code will also work without modification (or even recompilation) with objects of subclasses that are added to the system later.

This ability to reuse code as new kinds of objects are added to the system is touted as one of the major advantages of the object-oriented approach. However, it comes at the expense of our ability to reuse code as new operations are added to a system. Consider the general problem of designing software in which various interpretations (i.e. methods or functions) are defined for various kinds of objects (i.e. classes), as discussed by Harrison and Ossher [10], Krishnamurthi, Felleisen, and Friedman [11], and Appel [2, Section 4.2]. For example, Appel focuses on the design of an abstract syntax tree (A.S.T.) for a compiler: The different kinds of objects correspond to various program structures including expressions, statements, and declarations, while the different interpretations include static checks (such as type checking), various optimizations, or code generation for various architectures.

Such a system can be designed in the traditional object-oriented style, which lets us add new kinds of objects. However, new interpretations must be added to the superclass, requiring modification of existing source code and recompilation. The need to add operations to the class also violates a basic principle of data encapsulation, that each class should be defined with a minimal set of operations and edited only for a redesign, not a reuse.

We could, of course, abandon the object-oriented style, and adopt a style in which each function contains code for every type of object it could operate on. This lets us add new interpretations, but the introduction of a new kind of object forces us to edit each of the existing functions. We have once again prevented encapsulation and reuse without recompilation, and lost other benefits of object-oriented style as well. Appel argues that the latter style is more appropriate for his A.S.T. example, while the former is more appropriate for classic object-oriented systems (such as graphical user interfaces).

We demonstrate that this choice of styles represents a limitation of traditional object-oriented languages, not a fundamental design choice. Specifically, it is a consequence of the restriction of dynamic dispatch to the methods listed in a class. If we relax this restriction, we can create a system in which existing code can be reused (without access to source code) as both new kinds of objects and new interpretations are added. We show that it is possible to allow dynamic dispatch for functions not listed in a class (which we call *accessory functions* of the class), and demonstrate that accessory functions can be implemented efficiently.

This paper is organized as follows: We begin, in Section 2, with a brief review of the use of dynamic dispatch and its impact (both positive and negative) on code reuse. This section also covers the implementation of dynamic dispatch in C++. In Section 3, we spell out the goals that we intend to achieve by generalizing dynamic dispatch, describe the semantics (and C++ syntax) for accessory functions, and show that accessory functions can be used to enhance reuse in our example program. We then discuss, in Section 4, the relationship of accessory functions to other language properties such as support for data encapsulation. In Section 5, we briefly discuss the implementation of accessory functions in C++. Finally, we discuss related work in Section 6, and give our conclusions in Section 7.

2 Dynamic Dispatch and Reuse

One argument made by advocates of object-oriented languages is that object-oriented programming can facilitate code reuse. A well-designed class or hierarchy of classes can be reused without knowledge of, or access to, its implementation (just as a well-designed function or procedure can be reused in other languages). In this paper, we will be concerned with two types of reuse of classes. In the first, which we call reuse by inheritance, a programmer represents a new kind of data by making an extension of some existing data type. In the second, which we call reuse in a function, a programmer uses a class in the implementation of a new function (perhaps for a local variable or parameter).

We will focus on reuse that can be accomplished without modification of the source code that is to be reused. This is obviously important if software is distributed without source code, or if programmers are not able to modify the source code. It also prevents unnecessary code management complexities when source code is available. If several groups of programmers each reuse the same class, their modifications to that class must be merged, and the merged code must be tested, if these extensions are ever to be used together.

Note that reuse by inheritance (as defined above) can occur even in languages that do not support inheritance directly. Consider for example the task of representing various kinds of expression nodes in a compiler's abstract syntax tree (A.S.T.). (This example was adapted from [2]; we have focused on a simple method that can be understood with minimal knowledge of compiler construction.) In languages like C, the programmer can use a single `struct` to represent all kinds of expression nodes, distinguishing among the different kinds of nodes by including a `kind` field in the `struct` and using a `switch` statement to select code that is appropriate for each kind of expression node. Reuse by inheritance occurs if the programmer adds a new kind of expression node (perhaps because a new kind of expression has been added to the language). However, this reuse requires access to (and modification of) the existing code – the programmer must add a new `case` to the `switch` statements in existing functions, even if existing `case`s do not need to be modified.

In an object-oriented language like C++, the programmer can define the original kinds of expression nodes with a collection of classes, each of which inherits from an "abstract superclass" `Exp` (shown in C++ in Figure 1). The abstract superclass gives methods that are shared by all kinds of expressions, such as a `print_rep` method to produce a string that gives a printable representation for any kind of expression node. These methods must be defined for every subclass of `Exp`, and we can therefore request the printable representation of any object denoted by a reference of type `Exp &` (which must be of a class derived from `Exp`). Methods that are specific to one kind of node are defined only in the appropriate subclass (and thus cannot be requested through references of type `Exp &` in statically checked languages like C++).

We rely on the fact that we can request the `print_rep` for any object referred to by an `Exp &` in

```
class Exp {  // abstract superclass
public:
  virtual string print_rep() = 0;
private:
  ...
};

class Num : public Exp {
public:
  int value();
  virtual string print_rep();
private:
  ...
};

string Num::print_rep()
{
  // convert integer
  // to ASCII string
  return itoa(value());
}

class Plus : public Exp {
public:
  Exp &lhs();
  Exp &rhs();
  virtual string print_rep();
private:
  ...
};

string Plus::print_rep()
{
  return
    ``('' + lhs().print_rep() +
    ``+'' + rhs().print_rep() + ``)'';
}
```

Figure 1: Dynamic Dispatch Example

the print_rep method for class Plus. This method uses the printable representation for the left and right operands of the sum. Dynamic dispatch ensures that the print_rep method for the correct class is used, even though the compiler cannot determine statically which method will be chosen. This approach lets the programmer add new kinds of nodes by deriving a new class (with an appropriate print_rep) for each new kind of node. This does not require any modification of existing source code, and dynamic dispatch will ensure that the new class's print_rep is used for the new nodes, even in existing code (such as the print_rep method for class Plus).

Dynamic dispatch shifts the responsibility of selecting the appropriate print_rep code from the programmer to the programming language. In single-dispatch languages like C++ and Java, dynamic dispatch can be implemented by associating, with each object, a table of pointers to the code for each of the object's methods. In our example, each object of a class that inherits from Exp has a pointer to its print_rep method at a fixed offset in its dispatch table – to perform a call to print_rep when the type of object is not known, the compiler can generate an indirect function call using the table.

Note that it is possible for the programmer to implement dynamic dispatch in non-object-oriented languages that allow pointers to functions, such as C. However, this requires that the programmer undertake the tedious and potentially error-prone task of initializing using the tables of function pointers.

Unfortunately, the use of inheritance and dynamic dispatch inhibits reuse in a function. This is a consequence of the fact that only methods listed in a class can be dynamically dispatched based on that class. Consider what would happen if we wish to add a new pass to our compiler (such as an operation to interpret an expression), rather than a new type of expression node. If we had used a single class with a kind field, we could simply create a new function that takes an expression node, checks the kind field, and interprets the node in the appropriate way. However, if we wish to add this operation to the collection of classes in Figure 1, me must edit the class definitions to add an interpret method.

Editing the existing class definitions would be appropriate if we were redesigning, rather than reusing, these classes. However, we do not believe that every new use that requires dynamic dispatch should be considered a redesign: If this were the case, the author of a class would have the responsibility of enumerating all cases in which dynamic dispatch is needed for objects of that type.

There are a number of other ways to add an `interpret` method, each of which we consider unsatisfactory. First, we could introduce an `interpret` function that is not part of the `Exp` classes (in Java, it must be a method of some other class, such as a class `Interpreter`). This function could use `typeid` (in C++) or `instanceof` (in Java) in a series of `if` statements to select the appropriate code for the kind of node being interpreted. In a language without an equivalent of `typeid`, the programmer can add a `kind` field (or operation). In either case, this approach simply sets up a future problem with reuse by inheritance – any programmer adding a new kind of expression node must edit the code for `interpret` to work with the new type.

We could also use inheritance to produce new subclasses that add an `interpret` operation (deriving a class `Exp_with_interpret` from class `Exp`). However, this introduces spurious uses of multiple inheritance, which we consider highly undesirable (though we have no problem with legitimate uses of multiple inheritance): If we are to apply `interpret` to each new kind of node through an `Exp` reference, then the new node classes must share a common superclass with an `interpret` function, which introduces a second superclass for the new node classes.

Thus, if dynamic dispatch is provided only to functions listed in the class, we are forced to choose between allowing reuse in functions (if we use an explicit `switch`) or reuse by inheritance (if we use dynamically dispatched methods). To allow both kinds of reuse, we must allow dispatch for functions outside of the class. This is allowed in some languages that provide *multiple dispatch* [3, 9]. However, multiple dispatch has a higher run-time cost than single dispatch: techniques based on complete dispatch tables may require large tables, and other methods do not provide constant-time dispatch [1]. Accessory functions provide both kinds of reuse without the added complexity and cost of multiple dispatch.

Multiple dispatch can also be introduced as a programming technique rather than a language feature (for example, by using the visitor design pattern). This also introduces unnecessary overhead, and is less flexible than a compiler-generated dispatch, as we will see in Section 6.

```
// "pure virtual" accessory function
// for superclass
int interpret(virtual Exp &) = 0;

int interpret(virtual Num &n)
{
   return n.value();
}

int interpret(virtual Plus &p)
{
   return interpret(p.lhs())
        + interpret(p.rhs());
}
```

Figure 2: Accessory Function Example

3 Accessory Functions

Although no current single-dispatch language does so, it is in principle possible to allow dynamic dispatch on a parameter other than the receiver of the message. We call a function that does so an *accessory function* of the class involved in dispatch. Figure 2 shows how accessory functions can be used to add a dynamically dispatched `interpret` function to our A.S.T. example of Figure 1 (using a notation based on C++). The rest of this section gives our design goals, and gives possible syntax and semantics for integrating accessory functions into a C++ -like language.

Our goal is to provide the following properties of programs written with accessory functions:

- Accessory functions can be added to a group of classes without editing (or even reading) the source for the classes. To avoid violated the principle of data encapsulation, accessory functions do *not* have access to the private data of any classes they are not listed in. For example, the `interpret` functions of Figure 2 do not have access to the private data of any class, including the classes for the abstract syntax tree.
- New classes can be added to a program that uses accessory functions without any need to edit existing functions or classes. In other

words, we wish to allow both reuse in a function (the previous item) and reuse by inheritance.

- Accessory functions can be dispatched as efficiently as other single dispatch functions, such as virtual functions in C++.
- Except for the change in which argument is used for dynamic dispatch, function dispatch should follow the rules that exist in the language.
- The system must be able to produce errors about dispatch before the program is executed: A user must not see "method not found" errors while running a program (this was a design goal of C++).

3.1 Syntax

We need syntactic mechanisms to identify the parameter to be used in dynamic dispatch and to specify that a superclass function should be selected during a call in a subclass function. In this article, we give a syntax that is an extension of C++, and focus on definitions that are appropriate for C++, though accessory functions could be added to other statically typed single-dispatch object-oriented languages.

We identify an accessory function by using the keyword `virtual` in the declaration of a parameter. We consider `virtual` to be an attribute of a parameter rather than an attribute of the function itself. When `virtual` is used in the traditional way, we say that the member function has a virtual receiver (rather than a virtual parameter). Accessory functions for C++ may be created outside of any class, as in Figure 2, or they may be created as members (or friends) of one (or more) classes.

When an accessory function for a subclass needs to make use of the superclass function, it gives explicit type information for the virtual parameter. We use syntax that is similar to type casting for this purpose (we chose this notation because it produces the result that type casting of a reference produces for a statically dispatched function). To avoid introducing a new keyword, we reuse the word "virtual" for this purpose, i.e. the `interpret` function for `Num` could call the superclass function (were it not pure virtual) with the syntax `interpret( (virtual Exp) n )`. This is only legal if the new type is a public superclass of the argument type; its effect is analogous to using `interpret( (Exp &) n )` for a statically dispatched function.

3.2 Restrictions

We place several restrictions on the definition of accessory functions. Most are needed to prevent ambiguities that prevent us from selecting between dynamic and static dispatch at compile time.

- For any function, at most one parameter (including the receiver object) may be virtual. This is necessary to ensure that we do not need multiple dispatch. Here and in the remainder of this paper, we count the receiver object of a method as a parameter.
- No single scope can contain two functions that differ only in the dispatch mechanism of a parameter: We cannot have `f(Exp &)` and `f(virtual Exp &)`. This restriction is necessary because it would not be possible to distinguish calls to the two functions. C++ has an analogous rule for virtual functions.
- In any one scope, no two functions with the same name and arity (number of arguments) are dynamically dispatched on different parameters. This will play an important role in our function selection semantics below.
- All functions with parameters of class C must be defined before the execution of code that creates an object of class C. In traditional C++ environments, all functions are defined before program execution begins, and this restriction always holds. In environments that allow dynamic loading of classes (such as Java) this places restrictions on the relative timing of object creation and the loading of functions.
- To ease implementation in C++, we only allow accessory functions for classes that already have at least one virtual function: For example, we cannot have `f(virtual int &)`, as `int` has no dispatch table.

3.3 Function Selection Semantics

Function dispatch based on the types of multiple arguments, whether static or dynamic, raises two challenges: We must specify which function body is considered the correct choice for any given call, and we must provide a way for the program to branch to this code efficiently. In this section, we consider the question of how to adapt the existing dispatch rules of C++ for accessory functions, leaving the the question of how to branch to this function for Section 5.

The traditional choice of static vs. dynamic dispatch, and the new decision of which argument is to be used for dynamic dispatch, must be made at compile-time. These decisions are thus based on the types of the references used in the call (rather than the types of the objects they refer to), and the set of functions that are in scope at the point of the call. Once the compiler has selected dynamic dispatch on a particular object, the true "run-time" type of the object will be used in the actual call.

We ensure that we can statically determine which argument is to be used in dynamic dispatch by requiring that, in any one scope, no two functions with the same name and arity (number of arguments) are dynamically dispatched on different parameters. Essentially, we consider dispatch mechanism to be an attribute of the *message* (function name) rather than *method* (function body). Conflicts that might arise when two independent projects happen to use the same function names must be resolved via namespaces.

This restriction allows us to use the traditional C++ approach to dispatch: We select, from the set of functions that are in scope, the one with parameters types that best match the compile-time type information about the arguments used in the call. If there is no unique best match, we generate an error message. We then generate either a dynamic dispatch (based on the appropriate parameter type, if one parameter is virtual) or static dispatch (if no parameter is virtual).

Thus, if a group of functions of a given name and arity are dispatched on argument a, we produce a branch to the function that would have been called if all functions with this name and arity had been written as (possibly virtual) member functions of the classes of their a^{th} arguments. In other words, we generate a branch to the function that would have been called if we had violated the encapsulation of the classes.

As we will see in Section 5, our implementation allows us to produce warnings for certain surprising behavior that is a consequence of this combination of static and dynamic information.

3.4 Type Casting

The compiler will not produce a virtual argument by applying an implicit type cast to a value (though it may still convert a subclass type reference (or pointer) to a superclass reference (or pointer)). This is an extension of the existing C++ rule that the compiler will not apply an implicit cast to produce the receiver object. `Virtual` is generally used in the declaration of a pointer or reference type parameter: When applied to the declaration of a value parameter, it affects type casting, but not dispatch (since complete type information must be present at compile time).

The lack of casting for virtual arguments means that adding `virtual` to a parameter of an existing function may interfere with the compilation of code that had used this function: It may be necessary to add an explicit cast where an implicit one had been used previously to produce the (non-virtual) argument. We believe it would be possible to implement a system that allows implicit casting for accessory functions, but that such a system could produce highly confusing results, as casting is based on the functions that are in scope, but dispatch is based on all compatible functions in the final program.

3.5 Default Arguments

The above discussion ignores the issue of default arguments in C++. We believe these can be handled by treating a declaration with a default argument as if it were a group of declarations of overloaded functions, all but one of which simply supply extra arguments and call the original function.

4 Encapsulation

Existing single-dispatch object-oriented languages link together the following three properties: (1) The class(es) in which a function is defined, (2) The class(es) representation(s) that a function can access, and (3) The class that is used in dynamic dispatch of the function. In many languages, these three properties unified in the concept of "the" class of a method. C++ allows slightly more flexibility by allowing a function to access the representations of several classes if it is listed as a friend (or member) in each of them, though it can only be dynamically dispatched based on the (single) class of which it is a member. Even some multiple-dispatch languages unify the concept of access and dispatch: For example, in Cecil "a multi-method is granted privileged access to all objects of which the multi-method is a part, i.e., of the objects that are the method's argument constraints" [5, Section 1.5].

The unification of properties (1) and (2) essentially defines data encapsulation, which plays an essential role in reuse of classes. Since direct access to a class's representation is allowed only from those functions included in the class itself, we can rest assured that uses of the class by other functions (including all "reuse") will not corrupt any properties guaranteed by the class as it was originally written. Implicit in the idea of data encapsulation is the principle that programmers will not rampantly add operations to a completed class. If new operations are added to a class, and thus granted access to its representation, we can no longer guarantee that the representation cannot be corrupted. To retain this important property, accessory functions do *not* have access to classes involved in their dispatch (unless they are listed as friends of that class, for some reason).

We have proposed that the property of dynamic dispatch be separated from the property of inclusion in (and access to) a class. While we originally argued that this be done to support reuse, we find it appealing for several other reasons. First, it provides greater orthogonality of language features. Properties (1) and (2) above must remain unified, but dynamic dispatch is now fully independent. Second, we believe that accessory functions strengthen language support for data encapsulation. One tenet of data encapsulation is that each class should be defined with a set of operations that is both *adequate* and *minimal*:

> There can also be too many operations in a type... In this case, the abstraction may be less comprehensible, and implementation and maintenance are more difficult. The desirability of extra operations must be balanced against the cost of implementing these operations. If the type is adequate, its operations can be augmented by procedures that are outside the type's implementation. [14, Section 4.9.3]

Stroustrup also discusses this principle [17, Section 11.5.2]. Thus, it can be argued that both the `interpret` and `print_rep` functions belong outside the A.S.T. classes in our motivating example, as both can be written efficiently in terms of existing operations. However, without accessory functions, these operations must be placed inside the class.

Note that our need for dynamic dispatch for our A.S.T. example is not simply an artifact of the fact that we have not provided a more abstract way of traversing an abstract syntax tree. If we provide either an iterator or a traversal function to apply arbitrary code to each element of the tree, we still find the need to associate certain code with certain kinds of A.S.T. nodes. Dynamic dispatch provides a simple and efficient mechanism for doing so. Only the restriction of dispatch to members of the class keeps us from using it in these cases.

Since accessory functions do not have access to the private data of the data structure to which they are applied, they cannot save state information in this structure. We must, therefore, accumulate any needed information in some other way. In our "interpret" example, information is kept as temporary values in the C++ run-time stack; this works because we only need to produce a single final value (the result of the expression). If we need more complex information, such as a value associated with each node in the tree, we can build up an auxiliary data structure (for example, a second tree that contains values that correspond to the nodes in the A.S.T.). If we wish to traverse the data structure and modify it, we must modify the class (by using traditional virtual functions instead of accessory functions, or making the accessory functions into friends of the class). This is consistent with the principle that only operations listed in the class can access the class's private data.

5 Implementation for C++

Given the definitions and rules of Section 3, the implementation of accessory functions does not pose many interesting technical challenges. We simply move the existing algorithms for building dispatch tables from compile-time to link-time, retaining the general principles used for virtual functions in C++: each object contains a pointer to a table of all functions that may be dynamically dispatched based on its type, and the compiler statically produce code that will locate a given function with a (constant time) table lookup (for single inheritance, we simply use a fixed offset into the table).

The restrictions in Section 3.2 can be checked trivially as function declarations are processed at compile-time. To compile a (possibly dynamically dispatched) function call, we start by applying the C++ rules for overloaded function selection [17, Section 7.4] to the set of functions that are in scope and have the correct name and number of parameters, with the restriction that type casting of values cannot be used to create a match with a virtual parameter. If there is not a unique match, a compile-time error is produced. If there is a unique best match, we generate either a regular function call (if the best match has no virtual parameter) or a dynamic function dispatch.

If the best match had a virtual parameter, the dynamic dispatch is very much like a C++ virtual function call. We know that the virtual argument will contain a pointer to a table of dynamically dispatched functions, and generate a load of a function pointer from this table, and a branch to this address. The presence of accessory functions means that we no longer know the size or layout of this table when compiling a single file, but we handle this by treating the offset into the table as an undefined reference that will be filled in later by the linker.

Note that we need a separate entry in the dispatch table for each virtual parameter position, function name, and arity. A group of three-parameter functions may be dispatched differently from some two-parameter functions of the same name; different three-parameter functions may have different virtual parameters (as long as they are not in the same scope).

We rely on the compiler to give the linker a complete description of the DAG describing the class inheritance structure, and a list of all functions (complete with parameter types and information about which parameters are virtual). We topologically sort the inheritance DAG and we apply the algorithm used to build C++ virtual function tables, using a new offset for each set of statically distinct functions. Since the offsets are determined at this stage, we can resolve the undefined references produced at compile time.

This implementation places an increased load on the linker, and thus may increase link times. However, this is a fundamental consequence of the fact that declarations of accessory functions for a given class may be spread across several files (unlike the class' virtual functions), not a weakness of our implementation. The distribution of accessory functions over different files prevents detection by the compiler of certain errors that could be detected by traditional C++ compilers, such as the instantiation of a class with a pure virtual accessory function (one file may instantiate an object of a class for which pure virtual accessory functions are created in another file).

It may be possible to reduce the link-time overhead somewhat by replacing our implementation with a version of Millstein and Chambers' techniques for modular multimethod dispatch [15], restricted to the case of single dispatch. We have not investigated this possibility, since some degree of link-time overhead is unavoidable, and our main goal is to present a simple implementation that demonstrates that we can retain the constant-time nature of the dynamic dispatch used in C++.

6 Related Work

Snyder [16] and Liskov [13] have also studied conflicts between data encapsulation and other aspects of object-oriented programming. They discuss problems that arise when subclass operations are given access to superclass data or private operations, and Snyder [16] observes that a class's superclasses cannot be considered an implementation detail of the class in a system that allows multiple inheritance without replicating common superclasses.

Appel [2, Section 4.2], Harrison and Ossher [10], Krishnamurthi, Felleisen, and Friedman [11], and possibly many others have noted the conflict that arises between reuse by inheritance and reuse in a function. There are a number of approaches to resolving this problem, which we discuss in order of increasing familiarity for programmers familiar with C++.

Some techniques for multiple dispatch (also known as multi-methods) [3, 9, 5, 4] could be used to provide dispatch on a parameter other than the receiver of an object, or by including the type needed for dispatch in a new tuple type [12]. However, general multi-method dispatch either requires more than constant time per dispatch or excessively large dispatch tables [1]. However, recent techniques for multimethod dispatch [6] have very low overhead, and we believe they would be at least as efficient as our system for the case of single dispatch (which, as we have noted, is all that is needed to resolve the conflict between different kinds of reuse).

Work on multiple dispatch also differs from ours in that it not focused on the separation of dispatch and access. Cecil explicitly retains the unification of dispatch and access, though a change to this rule would probably not have any impact on performance. We believe the main barrier to the widespread use of these techniques to enable both kinds of reuse is the tendency of programmers to prefer familiar techniques and languages. The other approaches to solving this problem (including ours) focus on techniques or language extensions that can be applied to C++ or Java. General discussions of techniques for multi-method dispatch can be found in [1, Section 3.2] and [6, Section 3.7].

The visitor pattern could be applied to our abstract syntax tree example: Each tree class (such as `Plus`) would provide a "visit" operation that takes a "visitor" parameter, and sends the visitor a message that is specific to the tree subclass (e.g. `Plus::visit(visitor &v)` sends `v.visitPlus(this)`). This approach still interferes with the addition of new subclasses, since the visitor class must be extended to include a new method for each new subclass. The "Extended Visitor" protocol [11] fixes this problem, but still has higher overhead than a single dispatch accessory function, and to some degree shifts the burden of performing dispatch back from the compiler onto the programmer. It thus creates unnecessary opportunities for programmer error, and suffers from limitations due to the lack of compiler support. Krishnamurthi, Felleisen, and Friedman have developed a language named Zodiac to simplify the use of the extended visitor pattern, but it is not clear how quickly it will be adopted by programmers who are familiar with C++ or Java.

Harrison and Ossher [10] proposed the "Subject-Oriented Programming" style. This approach, like our accessory functions, can serve as the basis for extension of an existing language like C++ (it is currently available as a preprocessor for C++ in IBM's Visual Age for C++ Version 4). Instead of separating the property of dispatch from presence in a class, subject-oriented programming facilitates the decomposition of a class into different "subjects" that can be developed independently and then composed. A subject can correspond to one of our accessory functions, a group of functions, or functions together with associated data (like a class). This approach is more general than ours (though not more general than some of the multimethod systems), and correspondingly raises more new issues for programmers, such as the selection of composition system.

We have focused on providing a resolution to the conflict between reuse by inheritance and reuse in a function, while creating the minimal impact on programmers who are familiar with the traditional object-oriented style. Our extensions can be added to C++ by relaxing a single rule (that dynamic dispatch must be based on the receiver of the message). A preliminary description of accessory functions appeared at MASPLAS '99 [7]. We have also explored the possibility of allowing multiple virtual parameters [8], though this work does not make a significant contribution to the existing literature on multiple dispatch.

7 Conclusions

Current single-dispatch object-oriented languages provide dynamic dispatch only for functions listed in the class involved in dispatch, even if overloading is allowed for other parameters. This property gives the author of a class the responsibility of enumerating all cases in which dynamic dispatch is needed for objects of this type. This hinders code reuse by forcing the designer of a set of types to choose between allowing reuse by inheritance (by using dynamic dispatch) and reuse in a function (by using explicit `switch`es on the kind of object).

It is possible and (we believe) desirable to provide dynamic dispatch to users of a class hierarchy. In other words, we should eliminate the coupling between dispatch method and membership in (and access to) a class. This decoupling lets programmers achieve both reuse by inheritance and reuse in a function. Thus, our "accessory functions" improve the support for both reuse and data encapsulation, and can be implemented with the same efficient dispatch algorithms used in current C++ virtual function selection.

8 Acknowledgments

This work is supported by NSF grant CCR-9808694.

References

[1] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA'94 – Object Oriented Programming Systems, Languages and Applications*, pages 244–258, Oct. 1994. Published as SIGPLAN Notices Vol. 29, No. 10.

[2] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[3] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: merging lisp and object-oriented programming. In *OOPSLA'86 – Object Oriented Programming Systems, Languages and Applications*, pages 17–29, 1986.

[4] J. Boyland and G. Castagna. Parasitic methods: an implementation of multi-methods for java. In *OOPSLA'97 – Object Oriented Programming Systems, Languages and Applications*, pages 66–76, Oct. 1997. Published as SIGPLAN Notices Vol. 32, No. 10.

[5] C. Chambers. Object-oriented multi-methods in cecil. In *ECOOP '92 Conference Proceedings*, July 1992.

[6] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *OOPSLA'99 – Object Oriented Programming Systems, Languages and Applications*, pages 238–255, Oct. 1999. Published as SIGPLAN Notices Vol. 33, No. 10.

[7] C. B. Flynn and D. Wonnacott. Encapsulation, extension, and function dispatch in C++. In *The 1999 Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS '99)*, Apr. 1999.

[8] C. B. Flynn and D. Wonnacott. Reconciling encapsulation and dynamic dispatch via accessory functions. Technical Report DCS-TR-387, Dept. of Computer Science, Rutgers U., June 1999. Available as ftp://www.cs.rutgers.edu/pub/technical-reports/dcs-tr-387.ps.Z.

[9] J. Guy L. Steele. *Common Lisp: The Language (second edition).* Digital Press, 1990.

[10] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA'93 – Object Oriented Programming Systems, Languages and Applications*, Sept. 1993. Published as SIGPLAN Notices Vol. 28, No. 10.

[11] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote reuse. In *Proceedings of the Twelfth European Conference on Object-Oriented Programming ECOOP '98*, Apr. 1998.

[12] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA'98 – Object Oriented Programming Systems, Languages and Applications*, pages 374–387, Oct. 1998. Published as SIGPLAN Notices Vol. 33, No. 10.

[13] B. Liskov. Data abstraction and hierarchy (keynote address). In *OOPSLA'87 – Object Oriented Programming Systems, Languages and Applications (Addendum)*, pages 17–34, Oct. 1987. Published as SIGPLAN Notices Vol. 23, No. 5.

[14] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[15] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming ECOOP '99*, pages 279–303, June 1999. Published as Springer-Verlag LNCS 1628.

[16] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA'86 – Object Oriented Programming Systems, Languages and Applications*, pages 38–48, Oct. 1986.

[17] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1997.