Open Multi-Methods for C++

Peter Pirkelbauer Yuriy Solodkyy Bjarne Stroustrup

Texas A&M University,
College Station, TX 77843-3112
peter.pirkelbauer@tamu.edu, {yuriys, bs}@cs.tamu.edu

Abstract

Multiple dispatch – the selection of a function to be invoked based on the dynamic type of two or more arguments – is a solution to several classical problems in object-oriented programming. Open multi-methods generalize multiple dispatch towards open-class extensions, which improve separation of concerns and provisions for retroactive design. We present the rationale, design, implementation, and performance of a language feature, called open multi-methods, for C++. Our open multi-methods support both repeated and virtual inheritance. Our call resolution rules generalize both virtual function dispatch and overload resolution semantics. After using all information from argument types, these rules can resolve further ambiguities by using covariant return types. Great care was taken to integrate open multi-methods with existing C++ language features and rules. We describe a model implementation and compare its performance and space requirements to existing open multi-method extensions and workaround techniques for C++. Compared to these techniques, our approach is simpler to use, catches more user mistakes, and resolves more ambiguities through link-time analysis, runs significantly faster, and requires less memory. In particular, the runtime cost of calling an open multimethod is constant and less than the cost of a double dispatch (two virtual function calls). Finally, we provide a sketch of a design for open multi-methods in the presence of dynamic loading and linking of libraries.

Categories and Subject Descriptors D [3]: 3

General Terms Design, Languages, Performance

Keywords multi-methods, open-methods, multiple dispatch, object-oriented programming, generic programming, C++

1. Introduction

Runtime polymorphism is a fundamental concept of object-oriented programming (OOP), typically achieved by late binding of method invocations. "Method" is a common term for a function chosen through runtime polymorphic dispatch. Most OOP languages (e.g.: C++ [34], Eiffel [26], Java [3], Simula [6], and Smalltalk [20]) use only a single parameter at runtime to determine the method to be invoked ("single dispatch"). This is a well-known problem for operations where the choice of a method depends on the types of two or more arguments ("multiple dispatch"), such as the binary method problem [8]. Another problem is that dynamically dispatched functions have to be declared within class definitions. This is intrusive and often requires more foresight than class designers possess, complicating maintenance and limiting the extensibility of libraries. Open-methods provide an abstraction mechanism that solves these problems by separating operations from classes.

Workarounds for both of these problems exist for single-dispatch languages. In particular, the visitor pattern (double dispatch) [18] circumvents these problems without compromising type safety. Using the visitor pattern, the class-designer provides an accept method in each class and defines the interface of the visitor. This interface definition, however, limits the ability to introduce new subclasses and hence curtails program extensibility [12]. In [37] Visser presents a possible solution to the extensibility problem in the context of visitor combinators, which make use of RTTI.

Providing dynamic dispatch for multiple arguments avoids the restrictions of double dispatch. When declared within classes, such functions are often referred to as "multi-methods". When declared independently of the type on which they dispatch, such functions are often referred to as open class extensions, accessory functions [39], arbitrary multi-methods [28], or "open-methods". Languages supporting multiple dispatch include CLOS [32], MultiJava [12, 27], Dylan [30], and Cecil [10]). We implemented and measured both multi-methods and open-methods. Since open-methods address a larger class of design problems than multi-methods and are not significantly more expensive in time or space, our discussion concentrates on open-methods.

Generalizing from single dispatch to open-methods raises the question how to resolve function invocations when no overrider provides an exact type match for the runtime-types of the arguments. Symmetric dispatch treats each argument alike but is subject to ambiguity conflicts. Asymmetric dispatch resolves conflicts by ordering the argument based on some criteria (e.g.: an argument list is considered left-to-right). Asymmetric dispatch semantics is simple and ambiguity free (if not necessarily unsurprising to the programmer), but it is not without criticism [9]. In addition, asymmetric dispatch differs radically from C++'s symmetric function overload resolution rules.

We derive our design goals for the open-method extension from the C++ design principles outlined in [33]. For open-methods, this would mean the following. Open-methods should address several specific problems, be more convenient to use than all workarounds (e.g. the visitor pattern) as well as outperform them in both time and space. They should neither prevent separate compilation of translation units nor increase the cost of ordinary virtual function calls. Open-methods should be orthogonal to exception handling in order to be considered suitable for hard real-time systems (e.g. [25]), and parallel to the virtual and overload resolution semantics.

The contributions of this paper include:

GPCE'07, October 1–3, 2007, Salzburg, Austria.

- A design of open-methods that is coherent with C++ call-resolution semantics.
- An efficient implementation and performance data to support its practicality.
- A first known consideration of repeated and virtual inheritance for multi-methods.
- A novel idea of harnessing covariance of the return type for ambiguity resolution.
- A discussion of handling open-methods in the presence of dynamic linking.

2. Application Domains

Whether open-methods address a sufficient range of problems to be a worthwhile language extension is a popular question. We think they do, but do not consider the problem one that can in general be settled objectively, so we just present examples that would benefit significantly. We consider them characteristic for larger classes of problems.

2.1 Shape Intersection

An intersect operation is a classical example of multi-methods usage [33]. For a hierarchy of shapes, intersect() decides if two shapes intersect. Handling all different combinations of shapes (including those added later by library users) can be quite a challenge. Worse, a programmer needs specific knowledge of a pair of shapes to use the most specific and efficient algorithm.

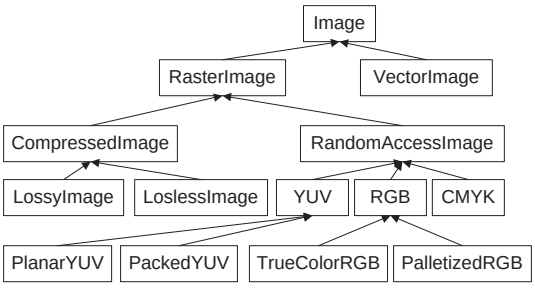
Using the multi-method syntax from [33], with **virtual** indicating runtime dispatch, we can write:

```
bool intersect(virtual Shape&, virtual Shape&); // open−method
bool intersect(virtual Rectangle&, virtual Circle&); // overrider
```

We note that for some shapes, such as rectangles and lines, the cost of double dispatch can exceed the cost of the intersect algorithm itself.

2.2 Data Format Conversion

Consider an image format library, written for domains such as image processing or web browsing. Conversion between different representations is not among the core concerns of an image class and a designer of a format typically can't know all other formats. Designing a class hierarchy that takes aspects like this into account is hard particularly when these aspects depend on polymorphic behavior. In this case, generic handling of formats by converting them to and from a common representation in general gives unacceptable performance, degradation in image quality, loss of information, etc. Optimal conversion between different formats requires knowledge of exact source and destination types, therefore it is desirable to have open-class extensions in the language, like open-methods. Here is the top of a realistic image format hierarchy:



A host of concrete image formats such as RGB24, JPEG, and planar YUY2 will be represented by further derivations. The optimal conversion algorithm must be chosen based on a source-target pair of formats [22] [41].

2.3 Compiler Pass over an AST

High-level compiler infrastructures ([35], [29]) typically use abstract syntax trees (AST) to represent programs. OOP enables modeling the language constructs in an oo-hierarchy. Then, storing pointers to a base class allows the use of several classes in the same context. For example, a statement class would be the common base for selective (e.g.: **if** or **switch**) and iterative constructs e.g.: **for** or **while**. Analysis or transformation passes that would take the semantics of the statement into account (e.g.: dataflow framework) need to uncover the real type. This is typically implemented with visitors or type-tags used to cast to a derived class. Open-methods are a non-intrusive technique to write these compiler passes. They guarantee type-safety, and allow retroactive extension of the oo-hierarchy to support new language features or dialects.

3. Definition of Open-methods

Open-methods are dynamically dispatched functions, where the callee depends on the dynamic type of one or more arguments. ISO C++ supports compile-time (static) function overloading on an arbitrary number of arguments and runtime (dynamic) dispatch on a single argument. The two mechanisms are orthogonal and complementary. We define open-methods to generalize both, so our language extension must unify their semantics. Our dynamic call resolution mechanism is modeled after the overload resolution rules of C++. The ideal is to give the same result as static resolution would have given had we known all types at compile time. To achieve this, we treat the set of overriders as a viable set of functions and choose the single most specific method for the actual combination of types.

We derive our terminology from virtual functions: a function declared virtual in a base class (super class) can be overridden in a derived class (sub class):

- an *open-method* is a non-member function with one or more parameters declared virtual
- an *overrider* is an open-method that refines another open-method according to the rules defined in §3.1
- an open-method that does not override another open-method is called a *base-method*.

For example:

```
struct A { virtual ~A(); } a;
struct B : A {} b;

void print(virtual A&, virtual A&); // (1)
void print(virtual B&, virtual A&); // (2)
void print(virtual B&, virtual B&); // (3)
```

Here, both (2) and (3) are overriders of (1), allowing us to resolve calls involving every combination of A's and B's. For example, a call print(a,b) will involve a conversion of the B to an A and invoke (1). This is exactly what both static overload resolution and double dispatch would have done.

To introduce the role of multiple inheritance, we can add to that example:

```
struct X { virtual ~X(); };
struct Y : X, A {};

void print(virtual X&, virtual X&); // (4)
void print(virtual Y&, virtual Y&); // (5)
```

Here (4) defines a new open-method print on the class hierarchy rooted in X. Y inherits from both A and X, and since both print open-methods have the same signature, – (5) is an overrider for both (4) and (1).

3.1 Overriding

DEFINITION 1. *An open-method is considered an overrider (or) for an open-method (om) in the same translation unit if it has the same name, the same number of parameters, covariant virtual parameter types, invariant non-virtual parameter types, and a possibly covariant return-type.*

A base-method must be declared before any of its overriders. This restriction parallels other C++ rules and greatly simplifies compilation. As shown in the previous example, an overrider can be associated with more than one base-method.

For every overrider and base-method pair, the compiler checks, if the exception specifications and covariant return type (if present) comply with the semantics used for virtual functions.

DEFINITION 2. *An open-method that is not an overrider and an overrider that introduces a covariant return type are considered a* base-method *for a translation unit.*

DEFINITION 3. *A* Dispatch table (DT) *maps the type-tuple of the base-method's virtual parameters to actual overriders that will be called for that type-tuple.*

Millstein and Chambers show in [28] that open-methods cannot be modularly type checked if the language (like C++) supports multiple implementation inheritance. Therefore, we split our call resolution mechanism into three distinct stages:

- Overload resolution
- Ambiguity resolution
- Runtime dispatch

The goal of overload resolution is to find at compile time a unique base-method, through which the call can be dispatched. This base-method determines a dispatch table through which the call will be made, the necessary casts of the arguments, and the expected return type. The actual overrider to handle the call is determined at runtime.

The C++ overload resolution rules [23] are unchanged: the viable set includes both open-methods and regular functions and treats open-methods like any other freestanding functions. Following the static rules exactly would imply that a base-method is used only if an open-method is the best match. We can do slightly better; that is, we can resolve some static ambiguities at link time (where more information is available): if a set of best matches consists of open-methods only and the intersection of their base-methods has a single element - overload resolution does not report an ambiguity. We demonstrate with an example:

```
struct X;
struct Y;
struct Z;

void foo(virtual X&, virtual Y&); // (1)
void foo(virtual Y&, virtual Y&); // (2)
void foo(virtual Y&, virtual Z&); // (3)

struct XY : X, Y {} xy;
struct YZ : Y, Z {} yz;

void foo(virtual XY&, virtual Y&); // (4)
void foo(virtual Y&, virtual YZ&); // (5)
```

Open-methods 1,2 and 3 are three independent base-methods defined on different class hierarchies. Because XY and YZ are parts of several hierarchies, overriders 4 and 5 refine several base-methods. In particular, 4 is an overrider for 1 and 2, while 5 is an overrider for 2 and 3.

A call `foo(xy,yz);` is now ambiguous according to the standard overload resolution rules as both 4 and 5 are equally good matches. Our relaxed rule, however, does not reject this call as ambiguous at compile time, because these overriders have a unique base-method through which the call can be dispatched – 2.

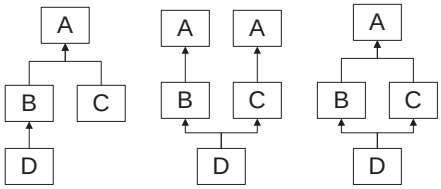
At link time, when all the overriders have been seen, we check the overriders for return type consistency, perform ambiguity resolution and build the dispatch tables.

Runtime dispatch simply looks up the entry in the dispatch table that corresponds to the dynamic types of the arguments and dispatches to that function.

This three-stage approach parallels the resolution to the equivalent modular-checking problem for template calls using concepts in C++0x [21]. Further, the use of open-methods (as opposed to ordinary virtual functions and multi-methods) can be seen as adding a runtime dimension to generic programming [4].

3.2 Ambiguity Resolution

C++ supports single-, repeated-, and virtual inheritance:



Note that to distinguish repeated and virtual inheritance, this diagram represents sub-object relationships, not just sub-class relationships. We must handle all ambiguities that can arise in all these cases. By "handle", we mean resolve or detect as errors.

Our ideal for resolving open-method calls combines the ideals for virtual functions and overloading:

- virtual functions: the same function is called independently of which class in an inheritance hierarchy is used in the call.
- overloading: a call is considered unambiguous if (and only if) every parameter is at least as good a match for the actual argument as the equivalent parameter of every other candidate function and that it has at least one parameter that is a better match than the equivalent parameter of every other candidate function.

This implies that a call of a single-argument open-method is resolved equivalently to a virtual function call. The rules described in this paper closely approximate this ideal. As mentioned, the static resolution is done exactly according to the usual C++ rules. The dynamic resolution is presented as the algorithm for generating dispatch tables in §3.4. Before looking at that algorithm, we present some key motivating examples.

3.2.1 Single Inheritance

In object models supporting single inheritance (§3.2) ambiguities can only occur with open-methods taking at least two virtual parameters. Ambiguities in this case have to be resolved by introducing a new overrider. The resolution of an open-method with one argument is identical to that of a virtual function. Thus, open-methods provide an unsurprising mechanism for expressing non-intrusive ("external") polymorphism. This eliminates the need to complicate a class hierarchy just to support the later addition of additional "methods" in the form of visitors.

3.2.2 Repeated Inheritance

Consider the repeated inheritance case (§3.2) together with this set of open-methods visible at a call site to `foo(d1,d2)`:

```
void foo(virtual A&, virtual A&);
void foo(virtual B&, virtual B&);
void foo(virtual B&, virtual C&);
void foo(virtual C&, virtual B&);
void foo(virtual C&, virtual C&);
```

Even tough the compiler can determine a unique base-method foo(A&, A&) through which the call can be dispatched, the call with two arguments of type D gets rejected at compile-time. The problem in this case, is that there are multiple pathes from D to sub-objects of type A.

To resolve that conflict, a user can either add an overrider foo(D&,D&) visible at the call site or explicitly cast arguments to either the B or C sub-object. Making an overrider for foo(D&,D&) available at the call site, renders the choice of the path to the subobject irrelevant. It would always be dispatched to the same overrider.

If the (B,C)-vs.-(C,B) conflict is resolved by casting, a question remains on how the linker should resolve a call with two arguments of type D? We know at runtime (by looking into the virtual function table's open-method table (see §4) which "branch" of a D object (either B or C) is on. Thus, we can fill our dispatch table appropriately; that is, for each combination of types, there is a unique "best match" according to the usual C++ rules:

	A	**B**	**C**	**D/B**	**D/C**
A	AA	AA	AA	AA	AA
B	AA	BB	BC	BB	BC
C	AA	CB	CC	CB	CC
D/B	AA	BB	BC	BB	BC
D/C	AA	CB	CC	CB	CC

This depicts the dispatch table for the repeated-inheritance hierarchy in §3.2 and the set of overriders above. Since the base method is foo(A&,A&) and A occurs twice in D, each dimension has two entries for D: D/B meaning "D along the B branch". This resolution exactly matches our ideals.

3.2.3 Virtual Inheritance

Consider the virtual inheritance class hierarchy from §3.2 together with the set of open-methods from §3.2.2: In contrast to repeated inheritance, a D has only one A part, shared by B, C, and D. This causes a problem for calls requiring conversions, such as foo(b,d); is that D to be considered a B or a C? There is not enough information to resolve such a call. Note that the problem can arise is such a way that we cannot catch it at compile time:

```
C& rc = d;
foo(b,rc);
B& rb = d;
foo(b,rb);
```

Using static type information to resolve either call would violate the fundamental rule for virtual function calls: use runtime type information to ensure that the same overrider is called from every point of a class hierarchy. At runtime, the dispatch mechanism will (only) know that we are calling foo with a B and a D. It is not known whether (or when) to consider that D a B or a C. Based on this reasoning (embodied in the algorithm in §3.4) we must generate this dispatch table:

	A	**B**	**C**	**D/A**
A	AA	AA	AA	AA
B	AA	BB	BC	??
C	AA	CB	CC	??
D/A	AA	??	??	??

We cannot detect the ambiguities marked with ?? at compile time, but we can catch them at link time when the full set of overriders are known.

3.3 Covariant Return Types

Covariant return types are a useful element of C++. If anything, they appear to be more useful for operations with multiple arguments than for single argument functions. Covariant return types complicate the use of workaround techniques. In case of the visitor, it would require even more foresight and lead to a proliferation of accept methods that have to be replicated in each derived class.

As an example for using covariant return type, consider a class Symmetric derived from Matrix:

```
Matrix& operator+(Matrix&, Matrix&);
Symmetric& operator+(Symmetric&, Symmetric&);
```

It follows that we must generalize the covariant return rules for open-methods. Doing so turns out to be useful because covariant return types help resolve ambiguities.

In single dispatch, covariance of a return type implies covariance of the receiver object. Consequently, covariance of return types for open-methods implies an overrider (or) - base-method (bm) relationship between two open-methods. Liskov's substitution principle [24] guarantees that any call type-checked based on bm can use or's covariant result without compromising type safety.

This can be used to eliminate what would otherwise have been ambiguities. Consider the class hierarchies $A \leftarrow B \leftarrow C$ and $R1 \leftarrow R2 \leftarrow R3$ together with this set of open-methods:

```
R1* foo(virtual A&, virtual A&);
R2* foo(virtual A&, virtual B&);
R3* foo(virtual B&, virtual A&);
```

A call foo(b,b) appears to be ambiguous and the rules outlined so far would indeed make it an error. However, choosing R2* foo(A&,B&) would throw away information compared to using R3* foo(B&,A&): An R3 can be used wherever an R2 can, but R2 cannot be used wherever an R3 can. Therefore, we prefer a function with a more derived return type and for this example get the following dispatch table:

	A	**B**	**C**
A	AA	AB	AB
B	BA	BA	BA
C	BA	BA	BA

At first glance, this may look useful, but ad hoc. However, an open-method with a return type that differs from its base method becomes a new base method and requires its own dispatch table (or equivalent implementation technique). The fundamental reason is the need to adjust the return type in calls. Obviously, the resolutions for this new base method must be consistent with the resolution for its base method (or we violate the fundamental rule for virtual functions). However, since R2* foo(A&,B&) will not be part of R3* foo(B&,A&)'s table, the only consistent resolution is the one we chose.

If the return types of two overriders are siblings, then there is an ambiguity in the type-tuple that is a meet of the parameter-type tuples. Consider for example that $R3$ derives directly from $R1$ instead of $R2$, then none of the existing overriders can be used for (B,B) tuple as its return type on one hand has to be a subtype of $R2$ and on the other a subtype of $R3$. To resolve this ambiguity, the user will have to provide explicitly an overrider for (B,B), which must have the return type derived from both $R2$ and $R3$.

Using the covariant return type for ambiguity resolution also allows the programmer to specify preference of one overrider over another when asymmetric dispatch semantics is desired.

To conclude: covariant return types do not only improve static type information, but also enhance our ambiguity resolution mechanism. We are unaware of any other multi-method proposal using a similar technique.

3.4 Algorithm for Dispatch Table Generation

Let us assume we have a multi-method $rf(h_1, h_2, ..., h_k)$ with k virtual arguments. Class h_i is a base of hierarchy of the i^{th} argument. $H_i = \{c : c <: h_i\}$ is a set of all classes from the hierarchy rooted at h_i. $X = H_1 \times H_2 \times \cdots \times H_k$ is the set of all possible argument type-tuples of f. Set $Y = \{(y_1, y_2, \cdots, y_k)\} \subseteq X$ is the set of argument type-tuples, on which the user defined overriders f_j for f. The set $O_f = \{f_0, \cdots, f_{m-1}\}$ is the set of those overriders ($f_0 \equiv f$). A mapping $F : Y \leftrightarrow O_f$ is a bijection between type-tuples on which overriders are defined and the overriders themselves.

Because different derivation paths may get different entries in the dispatch table, we assume that x_i in the type-tuple $x = (x_1, \cdots, x_k)$ identifies not only the concrete type, but also a particular derivation path for it (see [38] for formal definitions). Under this assumption, we define $B(x_i)$ to be a direct ancestor (base-class) of x_i in the derivation path represented by x_i. For example, for the repeated inheritance hierarchy from §3.2, $B(D/B) = B, B(D/C) = C, B(C) = A$, while for the virtual inheritance hierarchy $B(D/A) = A, B(B) = A, B(C) = A$.

For the sake of convenience, we define:

$$B_i(x) \equiv (x_1, \cdots, B(x_i), \cdots, x_k).$$

With it, we extend the definition of B to type-tuples as follows:

$$B(x) \equiv \{B_1(x), B_2(x), \cdots, B_k(x)\}.$$

$P(X, <) : (x_1, ..., x_k) <_P (y_1, ..., y_k) \Leftrightarrow \forall i : x_i <: y_i \wedge \exists j : y_j \not<: x_j$ defines a partial ordering that models ordering of viable functions for overload resolution as defined in [23]. $max\_set(S) = \{x \in S \subseteq X : \nexists y \in S : x < y\}$ is a set of maximal elements of S with respect to the partial ordering P.

Dispatch table DT is a mapping $DT : X \rightarrow O_f$ that maps various combinations of argument types to the overriders used to handle that combination.

For any combination of argument types $x \in X$, we recursively define entries of the dispatch table DT as following:

$$DT[x] = \begin{cases} F(x), x \in Y \\ DT[max\_set(B(x))], |max\_set(B(x))| = 1 \\ Ambiguity, otherwise \end{cases}$$

The above recursion exhibits optimal substructure and has overlapping sub-problems, which lets us use dynamic programming [14] to create an efficient algorithm for generation of dispatch table, shown in Algorithm 1.

To analyze its performance, we first note that comparison of two type-tuples from X can be done in time $O(k)$. If $n = max(|H_i|, i = 1, k)$ and $r = max(r_i, i = 1, k)$ (where r_i is a maximum number of times h_i is used as non-virtual base class in any class of hierarchy H_i) then $|X| <= (n * r)^k$ and the amount of edges for topological sort is less then $k * (n * r)^k$. Therefore the complexity of topologically sorting X is $O(k * n^k)$. The second loop has complexity $O(k^2 * n^k)$ so the overall complexity is $O(n^k)$ since k is a constant defining the amount of virtual arguments. This means that the algorithm is linear in the size of the dispatch table.

3.5 Alternative Dispatch Semantics

Our open-method semantics strictly corresponds to virtual member function semantics in ISO C++ but does not entirely reflect overload resolution semantics. The reason is that less information is available for compile-time resolution than for link-time or runtime resolution. For example, consider the repeated inheritance class hierarchy from §3.2 with a virtual function added:

Algorithm 1 Dispatch Table Generation

```
S ← topological_sort(X)
for all x ∈ S do
  if x ∈ Y then
    DT[x] ← F(x)
  else
    max_set = {B_1(x)}
    for i ← 2, k do
      dominated ← false
      for all e ∈ max_set do
        if F^-1(DT[e]) <_P F^-1(DT[B_i(x)]) then
          max_set ← max_set − {e}
        else if F^-1(DT[B_i(x)]) <_P F^-1(DT[e]) then
          dominated ← true
          break
      if not dominated then
        max_set ← max_set ∪ {F^-1(DT[B_i(x)])}
    if |max_set| = 1 then
      DT[x] ← F(max_set)
    else
      Report ambiguity for x
```

```
struct A { virtual void foo(); };
struct B : A {};
struct C : A { virtual void foo(); };
struct D : B, C {};

void bar(A&); // conventional overloading
void bar(C&);

void foobar(virtual A&); // open−method
void foobar(virtual C&); // open−method

D d;
B& db = d; // B part of D
C& dc = d; // C part of D

// (runtime) Virtual Member Function Semantics:
b.foo(); // calls A::foo
c.foo(); // calls C::foo
d.foo(); // error: ambiguous

// (compile time) Overload Resolution Semantics:
bar(db); // calls bar(A&)
bar(dc); // calls bar(C&)
bar(d); // calls bar(C&) (why not ambiguous?)

// (runtime) open−method Semantics:
foobar(db); // calls foobar(A&)
foobar(dc); // calls foobar(C&)
foobar(d); // error: ambiguous
```

The difference between the ordinary virtual function (foo) calls and the ordinary overloaded resolution for (bar) is odd and depends on pretty obscure rules that may be more historical than fundamental. Calls to the open-method foobar follow the virtual function resolution.

Further differences emerge in cases where a different resolution become possible in cases where additional information from other translation units may become available to resolve open-methods (see §5). This parallels decisions for related parts of the language. For example, the resolution of **static_cast** and **dynamic_cast** can

differ even given identical arguments: **dynamic_cast** can use more information than **static_cast**.

Another difference to overloading is that the return type of the overriders is bounded by the return type of the base-method.

4. Implementation

We implemented open-methods as described here by modifying the EDG compiler front-end [15].

4.1 Changes to Compiler & Linker

Our mechanism extends ideas presented in [16, 39] as to compiler and linker model. We adopted the multi-method syntax proposed in [33], which in turn was inspired by an earlier idea by Doug Lea. One or more parameters of a non-static freestanding function can be specified to be **virtual**. Overloading functions based only on the virtual specifier is not allowed.

A virtual argument must be a reference or pointer to a polymorphic class (that is, a class containing at least one virtual function). For example:

```
struct A { virtual ~A(); };

void print(virtual A&); // ok
void print(int, virtual A&); // ok

void dump(virtual Shape); // compiler error
void dump(virtual int); // compiler error
```

Open-methods are generic freestanding functions, which do not have the access privileges of member functions. If an open-method needs access to non-public members of a class, that class must declare it a friend. There are no abstract (pure virtual) open-methods; that is every open-method must be defined. Consider a (dynamic) library D1 that introduces a new class and a second (dynamic) library D2 that defines a new abstract open-method. Then, the presence of an overrider for the class in D1 could not be guaranteed. The alternative would be runtime "method not defined" errors (reported as exceptions), but that solution would be inconsistent with the rest of C++ and would limit the use of open-methods in embedded systems.

For each translation unit, the compiler generates an *open-method description* (OMD) file that stores the data needed to generate the runtime data structure discussed in §4.2. This includes the names of all classes, their inheritance relationships, and the kind of inheritance. Open-methods are represented by name, return type, and their parameter list. Finally, the OMD-file also contains definitions of all user-defined types that appear in signatures of open-methods (both as virtual and regular parameters). These definitions are necessary to regenerate prototype declarations for the open-methods, which pass data through by value.

The pre-linker uses Coco/R [40] to parse the OMD-files. Then, the pre-linker synthesizes the OMD-data, associates all overriders with their base-methods, generates dispatch tables, issues link-errors for ambiguities, determines the indices necessary to access the open-method and dispatch-table, as well as defines and interlinks the om-tables of each sub-object type according to §4.2.

4.2 Changes to the Object Model

We augment the IA-64 C++ object model [13] by four elements to support constant time dispatching of open-methods. First, for each base-method there will be a dispatch table containing the function addresses. Second, the v-table of each sub-object contains an additional pointer to the *open-method table (om-table)*. Finally, the indices used for the open-method-table offsets are stored as global variables.

Figure 1 shows the layout of objects, v-tables, om-tables and dispatch-tables for repeated (left) and virtual (right) inheritance. Our extensions to the object-model are shown with grey background. From left to right the elements in each diagram represent the object, v-table, om-table, and dispatch table(s) for the class hierarchy in §3.2. From top to the bottom, the objects are of type A, B, C, and D respectively.

An open-method can be declared after the declarations of the classes used in its virtual parameters. Therefore, the compiler cannot reserve v-table entries to store the data related to open-method dispatch immediately in a class's virtual function table. Hence, we always extend every v-table by one pointer referencing the om-table, which can be laid down later by the pre-linker.

The om-table reserves one position for each virtual parameter of each base-method, where objects of this type can be passed as arguments. This position stores an index into corresponding dimension of the dispatch table. Since the size of the om-tables is not known at compile time, our technique relies on a literal for each open-method and virtual parameter position (called foo_1st, foo_2nd in Figure 1) that determines the offset within the om-tables.

Note that Figure 1 depicts our actual implementation, where entries for first argument positions already resolve one dimension of the table lookup. Entries for all other argument positions store the byte offset within the table.

In the presence of multiple-inheritance, a this-pointer shift might be required to pass the object correctly. In this case, we replace the address of the overrider by an address of a thunk that takes care of correctly adjusting the this-pointer. As described in §3.2.2 in case of repeated inheritance different bases can show different dispatch behavior depending on the sub-object to which the this-pointer refers. As a result, different bases may point to different om-tables. In case of virtual inheritance, the open-method dispatch entries are only stored through the types mentioned in the base-method. Hence, in the virtual inheritance case, all open-method calls are dispatched through the virtual base type.

4.3 Alternative Approaches

We considered a few other design alternatives to explore their trade-offs in extensibility and performance.

Multi-Methods: Multi-methods differ from open-methods in that the base-method has to be declared in the class definition of its virtual parameters. This allows the offset within the v-table be known at compile time, which saves two indirections per argument of a function call (one for the om-table, and one to read the index within the om-table). For a call with k virtual arguments, open-methods need $4k + 1$, while multi-methods need only $2k + 1$ memory references to dispatch a call. The downside of multi-methods is that existing classes cannot easily be extended with dynamically dispatched functions.

Chinese Remainders: In this section, we present an "ideal" scheme for implementing open-methods, inspired by ideas presented in [19]. The proposed scheme circumvents the necessity for open-method tables by moving all the necessary information from the class to the dispatch table.

Suppose that for every multi-method f there is a function $I_f : T \times N \rightarrow N$ such that for any type $t \in T$ (where T is a domain of all types) and argument position $n \in N$ it returns index of type t in the n^{th} dimension of the f's dispatch table. If such function is reasonably fast (preferably constant time) and its range is small (preferably from 0 to maximum number of types that can be used in any argument position) then we can efficiently implement multiple dispatch by properly arranging rows and columns accordingly to the indices returned by I_f. As in [19] we use the Chinese reminder theorem [14] to generate function I_f.

Despite its elegance, this approach is rather theoretical because it is hard to use for large class hierarchies. The reason is that we need to assign different prime numbers to classes and perform

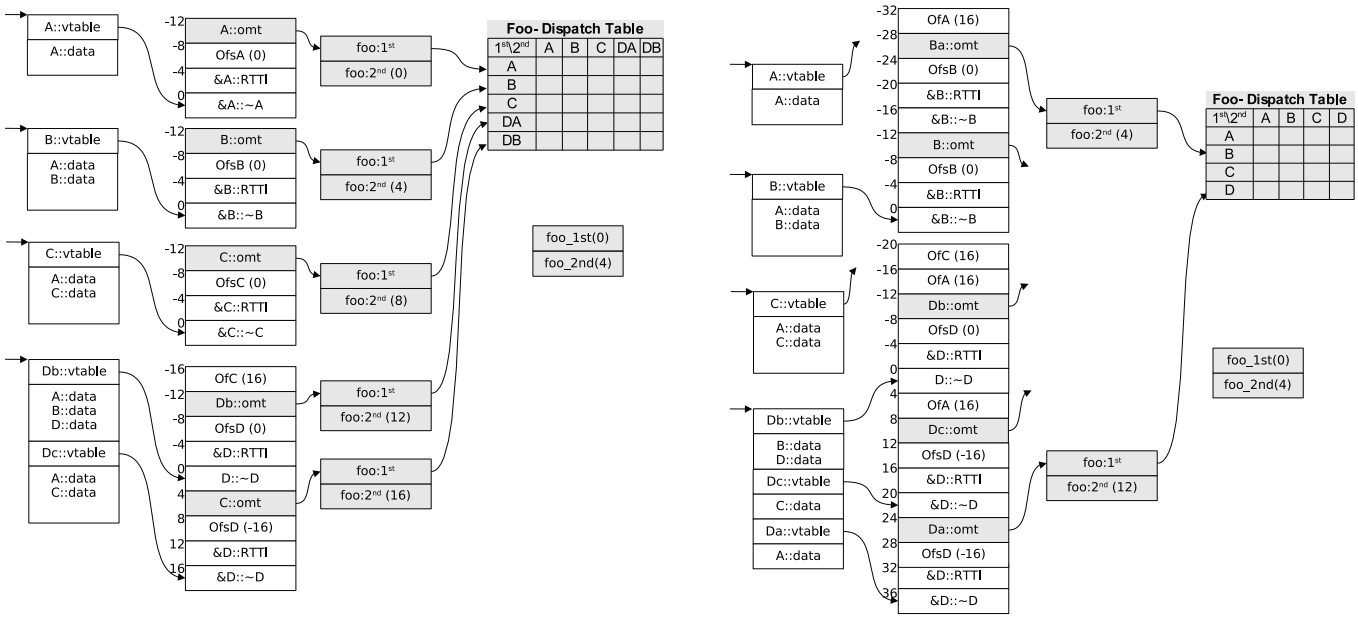


Figure 1. Object Model for repeated (left) and virtual (right) inheritance

computations on numbers that are bound by the product of these primes. Such product can fit into 32-bit integers for only the 9 first primes and into 64-bit integers for only the 15 first primes. Table compression techniques [2] or use of minimal perfect hash functions [14] instead, can help overcome the problem.

$AA^1 in D_1$	**A**	**B**
A	AA^1	AB^2
B	AA^1	AB^2

$AA^1 in D_2$	**A**	**B**
A	AA^1	AA^1
B	AA^1	BB^1

5. Dynamic Linking

Outside embedded systems, dynamically linked libraries are almost universally used with C++. Thus, a design for open-methods that does not allow for DLLs is largely theoretical. We do not currently have an implementation supporting dynamic linking, but we can outline a design addressing the major issues for open-methods in a dynamically linked library. It guarantees that the most specialized overrider available at runtime that preserves type-safety of a call will be used to dispatch a call.

Dynamic modules, compiled independently, may have different sets of overriders defined at the time of compilation. Furthermore, there could be new classes added to a hierarchy in one of the modules and objects of those classes may be passed into code of other modules. This is not a problem for regular virtual functions as their v-tables are found in the module where the class was defined. In case of open-methods, the dispatch table generated within a particular module can be simply unaware of a class, defined somewhere else. To account for this, the dynamic loader will have to update dynamically dispatch tables of each open-method as new modules are loaded and more information becomes available.

Covariant return introduces subtleties when dynamic linking is used. Consider a two-class hierarchy $A \leftarrow B$ and another two-class hierarchy $R1 \leftarrow R2$. The base-method R1 foo(**virtual** A&, **virtual** A&) is defined in a header visible by two dynamically linked modules D_1 and D_2 that do not know anything about each other. Module D_1 introduces overrider R2 foo(A&, B&) and module D_2 introduces overrider R1 foo(B&, B&). Each of the dynamically linked modules perfectly type-checks and links with foo() resolved through the dispatch table (a superscript in a cell denotes the type that is returned by an overrider e.g. AB^2 denotes R2 foo(A&, B&)):

When both modules are loaded together, we get the dilemma of how to resolve a call with both arguments of type B. On one side foo(B&,B&) from D_2 is more specialized, but on the other side foo(A&,B&) from D_1 imposes the additional requirement that the return type of whatever is called for (B,B) should be a subtype of R2, which R1 is not. Such scenario would have been rejected should it happen at compile/link time, however at load time we do not have this option anymore.

Keeping all dispatch tables of a particular open-method consistent on the overrider that will be called for a particular combination of types will force us to choose between suboptimal and type unsafe alternatives. What is worse - is that there may not be a unique type-safe alternative.

Imagine for example that a module D_3 introduces an overrider R3 foo(B&, A&) where $R1 \leftarrow R3$, so $R2$ and $R3$ are siblings. When D_1 and D_3 are loaded together, neither R2 foo(A&, B&) nor R3 foo(B&, A&) can be used to resolve a call with both arguments of type B - both alternatives are type unsafe for the other overrider.

To deal with this subtlety, we propose to weaken for the DLL case the requirement that the same overrider should be called for the same tuple of dynamic types regardless of the static types used at the call site. We require that the same overrider be used only if it is type-safe for the caller. Strictly speaking R1 foo(B&,B&) is not an overrider of R2 foo(A&, B&) as defined in §3.1, because its return type is not changing covariantly in respect to the types of arguments. Therefore, it cannot be considered for the dynamic resolution of calls made statically through the base-method R2 foo(A&, B&).

Taking the above into account, we propose that the dynamic linker fills in the dispatch table of every base-method independently. This results in:

AA^1	**A**	**B**	AB^2	**B**	BA^3	**A**	**B**
A	AA^1	AB^2	**A**	AB^2			
B	BA^3	BB^1	**B**	AB^2	**B**	BA^3	BA^3

It looks as if the dispatch table for the base-method R1 foo(A&,A&) now violates covariant consistency, but in reality it does not because all the return types in it are cast back through thunks to R1, which is the type statically expected at the call site.

As can be seen, this logic may result in different functions being called for the same type tuple depending on the base-methods seen at the call site. We note, however, that *the call is always made to the most specialized overrider that is type-safe for the caller*.

Even when covariant return types are not used, dynamic linking can introduce ambiguities in the dispatch tables. The simplest example would be to think of two different modules that both provide overrider for the same pair of dynamic types.

Let us consider a plausible scenario involving three DLLs:

```
// dll−1
struct GuiButton { virtual ~GuiButton(); };
struct GuiEvent { virtual ~GuiEvent(); };

void handleEvent(virtual GuiButton&, virtual GuiEvent&);

// dll−2
#include<dll1>
struct MyButton : GuiButton { };

void handleEvent(virtual MyButton&, virtual GuiEvent&);

// dll−3
#include<dll1>
struct SpecialEvent : GuiEvent {};

void handleEvent(virtual GuiElement&, virtual SpecialEvent&);
```

The first DLL defines a class GuiButton, a class GuiEvent, and a base-multi-method handleEvent. Internally, a second DLL derives a new type MyButton from GuiButton and introduces a new overrider for handleEvent. Likewise, the third DLL derives a new internal class SpecialEvent from GuiEvent and introduces a new overrider. The second and third DLL could stem from different vendors that do not know about each other.

Now a call of handleEvent with a MyButton and a SpecialEvent is ambiguous. The writer of the total system (the "system integrator") should in principle have considered that possibility and dealt with it. Therefore, one solution would be to terminate the program or to throw an exception [31]. However, such problems are hard to predict and design for. Relaxed Multi-Java [27] resolves these conflicts by introducing glue methods (to glue DLL2 and DLL3) that the system-integrator provides. While this might be a viable solution for software developers integrating several libraries, it is not a feasible scenario for end-user applications, as dynamically linked modules can be loaded into the process without direct request of a developer. This, for example, is the case with various component object models when application may ask the system to create an object with a particular name and operating system will locate and load the module it is resided in.

We emphasize that in current C++, it is not possible to write programs against both types without getting an ambiguity. We report such an ambiguity when it is detected at compile or link time, however, at load time we do not have such an option as it is detected on a user's machine.

One way to handle such a scenario will be to treat both classes as their base classes and dispatch appropriately. We note, however, that in the plug-in like usage scenarios modules may have only seen the interface: the base-multi-method and the roots of the hierarchies on which it is defined. Nevertheless, such use-cases would expect more refined overriders to handle calls on derived classes.

In principle, *both* handleEvent functions should correctly handle the event; that is, each handleEvent function must assume that its argument type is just base class and the actual argument could be of an unknown derived class. Consequently, each handleEvent function must be written in a way that is generic on its argument (probably using virtual functions on the individual argument). This implies that as long as an event handler's code does not make more assumptions about its arguments than the interface defined in the base-class guarantees, it can be replaced by the other event handler. Even a non-deterministic selection of the overrider would produce a correct result! However, developers and testers strongly prefer deterministic choices, so we consistently choose one of the alternatives. Furthermore, as we have already mentioned, two DLLs may provide different implementations for the same overrider, in which case we have to make a choice.

With this said we propose to resolve ambiguities at load time as following. For each base-method of a given open-method:

- If there is a unique best match among type-safe overriders that can handle a particular combination of argument types – use it.
- If there is no unique best match, – make a deterministic choice among all best matches.

We chose an unspecified determinism over non-determinism to guarantee that the same method will always be selected and to keep the dynamic dispatch symmetric.

6. Related Work

Programming languages can support multi-methods either through built-in facilities, pre-processors, or through library extensions. Naturally, tighter language integration enjoys a much broader design space for type checking, ambiguity handling, and optimizations compared to libraries. In this section, we will first review both library and non-library approaches for C++ and then give a brief overview of multi-methods in other languages.

6.1 Cmm

Cmm [31] is a preprocessor based prototype implementation for an open-method C++ extension. It takes a translation unit and generates C++ dispatch code from it. Cmm is available in two versions. One of them uses RTTI to recover the dynamic type of objects to identify the best overrider. The other approach achieves constant time dispatch by relying on a virtual function overridden in each class. Dispatch ambiguities are signalled by throwing runtime exceptions. Cmm allows dynamically linked libraries to register and unregister their open-methods at load and unload time. In addition to open-method dispatch, Cmm also provides call-site virtual dispatch. Cmm does not provide special support for multiple inheritance and therefore its dispatch technique does not entirely conform with virtual function semantics.

6.2 DoubleCpp

DoubleCpp [5] is another preprocessor based approach for multi-methods dispatching on two virtual parameters. It essentially translates these multi-methods into the visitor pattern. For doing so, DoubleCpp requires access to the files containing the class definitions in order to add the appropriate accept and visit methods. DoubleCpp, like any other visitor-based approach, does not report but quietly resolve ambiguities.

6.3 Accessory Function

The accessory functions papers [16, 39] allow open method dispatch based on a single virtual argument and discuss ideas to extend the mechanism for multiple dispatch. The compilation model they

describe uses, like our approach, a compiler and linker cooperation to perform ambiguity resolution and dispatch table generation. However, the accessory functions are integrated into the regular v-tables of their receiver types, which requires the linker to not only generate the dispatch table but also to recompute and resolve the v-table index of any other virtual member function. Both papers do not provide a detailed discussion of the intricacies when multiple inheritance is involved. The authors do not refer to a model or prototype implementation to which we could compare our approach.

6.4 Loki

Loki [1], based on Alexandrescu's template programming library with the same name, provides several different dispatchers that balance between speed, flexibility, and code verbosity. Currently, it supports multi-methods with two arguments only, except for the constant-time dispatcher that allows more arguments. The static dispatcher provides call resolution based on overload resolution rules, but requires manual linearization of the class hierarchy in order to uncover the most derived type of an object first. All other dispatchers do not consider hierarchical relations and effectively require explicit resolution of all possible cases.

6.5 Other Languages

One of the first widely known languages to support multi-methods was CLOS [32]. CLOS linearizes the class hierarchy and uses asymmetric dispatch semantics to avoid ambiguity conflicts. Cecil [9, 10] views silent ambiguity resolution as a potential source for programming errors. Therefore, it uses symmetric dispatch semantics and dispenses with object hierarchy linearization in order to expose these errors at compile-time. In [28], Millstein and Chambers discuss the trade-offs between multi-methods and modular type-checking in languages with neither a total order of classes nor asymmetric dispatch semantics. Ranging from globally type-checked programs to modularly type-checked units, the models embrace or restrict the expressive power of the language to different degrees. Based on these findings, MultiJava [12] implements a model that allows separate compilation and eliminates the need for a link-time type-checker but also curtails extensibility. Relaxed MultiJava [27] re-introduces a link-time type checker and relies on the system integrator to resolve ambiguities by providing new overriders (glue-methods). Parasitic methods model multi-methods as class members and give the receiver precedence over other arguments. Implementations for Java [7] and Smalltalk [17] exist. An example of a language adding multi-methods through a library is Python [36]. Chambers and Chen [11] present an alternative implementation technique based on a lookup DAG. Their work generalizes multiple dispatch to be a subset of predicate-based dispatch.

7. Results

In order to discuss time and space performance, we compare a number of handcrafted implementations with EDG/Omm, our implementation described in §4, Cmm, DoubleCpp, and the Loki library. The handcrafted approaches include multi-methods and open-methods, model implementations used to initially assess the performance trade-offs, a Chinese Remainder (§4.3) based implementation, and the visitor pattern.

We wrote 20 classes (representing shapes, etc.) which can intersect each other. Overall, this results in 400 combinations for binary dispatch functions. We implemented 40 specific intersect functions to which all of the 400 combinations are dispatched to. In order to get a reliable timing of the function invocation, these 40 intersect functions only increment a counter. Since not all techniques we use support multiple inheritance, these 20 classes only use single inheritance. The actual test consists of a loop that randomly chooses 2 out of 32 objects and invokes the intersect method. We implemented a table-based random number generator that is simple and does not contain any floating-point calculations or integer-divisions. We ran the loop twice with the same random numbers: The first run allows implementations, which build the dispatch data structure on the fly to warm up and load data/code into the cache. The second loop was timed. The clock-cycle based timer takes the time before and after the loop and we calculate the average number of clock-cycles per loop to compare the results.

7.1 Implementations

We tested the approaches on a Pentium D, 2.8 Ghz running CentOS Linux and a Core2Duo running Mac OSX. The code was compiled with g++ 4.0 (Linux) and gcc 4.0.1 (OSX) with optimization level set to -O3. EDG/Omm generates source code lowered to C, which was compiled with the corresponding gcc versions and linked to the pre-linker generated dispatch tables.

Using the Chinese Remainder approach, the number associated with the dispatch table grows exponentially with the number of types. Therefore the test is limited to eight types instead of 20 and the size of the executable is omitted. We could only implement a simplified version that can handle eight types instead of 20. Hence, we omit the size of the program executable.

For Loki, we only tested the static dispatcher because the other require manual handling of all possible cases. Using other dispatchers would have been closer to a scenario of manually allocated array of functions through which calls are made. However, as we indicated before, the dual nature of multi-methods require them to provide both dynamic dispatch and automatic resolution mechanism.

7.2 Results & Interpretation

Our experimental results can be summarized in terms of execution time and program size:

Approach	**Size (bytes)** Linux	**Cycles/Loop** Pentium-D	**Cycles/Loop** Core2Duo
Virtual function	n/a	75	55
C++ Multi-method	19 547	78	60
C++ Open-method	19 725	82	63
EDG/Omm	70 647	82	64
Double Cpp	20 859	120	82
C++ Visitor	35 289	132	82
Chinese Remainders	n/a	175	103
Cmm (constant time)	112 250	415	239
Cmm	111 305	1 320	772
Loki Library	34 908	3 670	2 238

Executable size: We present the executable size obtained on the Linux system. The size of dispatch tables is mentioned as one of the major drawbacks of providing multi-methods as programming language feature [39]. However, our results reveal that the visitor that is based on a brute force implementation is 80% bigger than the multi-method approach. With the visitor, each shape class has intersect methods for all 20 shapes of the hierarchy. A somewhat smarter approach would be to remove redundant intersect overriders. However, removing specific overriders is tedious and difficult to maintain, since the dispatch would be based on the static type information of the base class. Even an optimized approach would be at best able to match the multi-method implementation, simply because each type contains 20 intersect entries in the v-table. Multiplying this with the number of shapes, 20, results in 400, exactly the number of entries found in the dispatch table. The executable generated by EDG/Omm is bigger because it is statically linked to EDG's C++ runtime library. We do not discuss the program size of

the two Cmms and Loki, since they use additional header files such as <typeinfo> and <stdexcept> that distort a direct comparison.

Execution time: Multi-Methods, Open-Methods, and EDG/Omm are (as expected) roughly comparable to a single virtual function dispatch, which needs 75 (55 on the Core2Duo) cycles per loop. Hence, the better performance compared to the visitors is not surprising. However, the fact that multi-methods reduce the runtime to 62% (73%) of the reference implementation using the visitor is noteworthy. We conjecture this is an effect of the size of the class hierarchy and that the time to double dispatch depends on the number of overriders. On the Pentium D, two observations support our conjecture: firstly, the DoubleCpp-based visitor has no redundant overriders and runs slightly faster. Secondly, we simulated an analysis pass dispatching over AST-objects of 20 different types and counting the category to which they belong (type, declaration, expression, statement, other). In this case, the double dispatch has only 20 leaf-functions instead of 400 and our dispatch test runs 78 cycles instead of 132. The open-method approach requiring only five overriders, is still faster and needs 68 cycles.
The difference between multi-methods and open-methods (EDG/Omm) is within the expected range. Three more indirections require 4 (4) more clock cycles on the Pentium and 3 (4) more on the Core2Duo. Although significantly slower, Cmm (constant time) performs better than expected, since its author estimates the dispatch cost as 10 times a regular virtual function call. As expected the two non-constant time approaches perform worst.

Significance of performance: The performance numbers comes from experiments designed to highlight the cost of multiple dispatch: the functions invoked hardly do anything. Depending on the application the improved performance might or might not be significant. For the image conversion example, gains in execution speed are negligible compared to time spent in the actual conversion algorithm. In other cases, such as the evaluation of expressions using user-defined arithmetic types, traversal of abstract syntax trees, and some of the most frequent shape intersect examples, the speed differences among the double dispatch approaches appear to be notable.

Contrary to much "popular wisdom", our experiments revealed that for many applications the use of dispatch tables for open-methods and multi-methods actually reduce the program size compared to brute-force and work-around techniques.

8. Conclusions and Future Work

We presented a novel approach to dispatching open multi-methods that is in line with the multiple inheritance semantics of the current C++ object model and the C++ overload resolution rules. This implies compile-time or link-time detection of ambiguities. By considering covariant return type in the ambiguity resolution, we reduce the number of potential conflicts. We have discussed an implementation based on modifications to the EDG compiler front-end and have described a mechanism that supports the integration of several translation units. Our evaluation of different approaches to implementing open-methods in C++ shows that our approach is significantly better (in time and space) than current alternatives. Indeed, it is almost as efficient as single dispatch. Since the dispatch is constant time and does not rely on exceptions to signal ambiguities, it is applicable in embedded and hard real-time systems.

Future plans to extend our work include:

8.1 Virtual Function Templates

Virtual function templates are a powerful abstraction mechanism not part of C++. Generating v-tables for virtual function templates requires a whole-program view and C++ traditionally relies almost exclusively on separate compilation of translation units. The pre-linker technique described here should be able to synthesize v-tables for virtual function templates as it does for open-methods.

8.2 Function Pointers to Open-methods

Pointers to open-methods would generalize pointer to member-functions. This allows to separate n-ary open-methods from their arguments and write code abstracted from a concrete operation.

8.3 Calling a Base Implementation

C++ provides a syntax to call a particular base implementation of a virtual member function directly, avoiding dynamic dispatch. This is often used to call the function in the base class. To do this, C++ requires the user to use a fully qualified name of virtual member function: e.g.: `p->MyClass::foo()`; It is likely that similar functionality will be required for open-methods.

8.4 Space Optimizations

With large class hierarchies, the size of dispatch tables can become significant, especially when we consider support for covariant return types. The implementation of space optimizations techniques to compressing and reusing of dispatch tables [2] would reduce the memory footprint.

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. AW C++ in Depth Series. Addison Wesley, January 2001.

[2] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA '94*, pages 244–258, New York, NY, USA, 1994. ACM Press.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, 3rd edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] L. Bettini, S. Capecchi, and B. Venneri. Double dispatch in C++. *Software - Practice and Experience*, 36(6):581 – 613, 2006.

[6] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula BEGIN*. Auerbach Press, Philadelphia, 1973.

[7] J. Boyland and G. Castagna. Parasitic methods: an implementation of multi-methods for Java. In *OOPSLA '97*, pages 66–76, New York, NY, USA, 1997. ACM Press.

[8] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theor. Pract. Object Syst.*, 1(3):221–242, 1995.

[9] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, pages 33–56, London, UK, 1992. Springer-Verlag.

[10] C. Chambers. The Cecil language: Specification and rationale. 3.2. Technical report, Department of Computer Science and Engineering. University of Washington, 2004.

[11] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. In *OOPSLA '99*, pages 238–255, New York, NY, USA, 1999. ACM Press.

[12] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[13] Codesourcery.com. The Itanium C++ ABI. Technical report, 2001.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[15] Edison Design Group. C++ Front End, March 2006.

[16] C. B. Flynn and D. Wonnacott. Reconciling encapsulation and dynamic dispatch via accessory functions. Technical Report 387, Rutgers University Department of Computer Science, 1999.

[17] B. Foote, R. Johnson, and J. Noble. Efficient Multimethods in a Single Dispatch Language. *Proceedings of the European Conference on Object-Oriented Programming, Glasgow, Scotland, July*, 2005.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[19] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36(2):139–156, 2006.

[20] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[21] D. Gregor, J. Jarvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming. *OOPSLA'06*, 2006.

[22] International Organization for Standardization. *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. 1994.

[23] ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.

[24] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87*, pages 17–34, New York, NY, USA, 1987. ACM Press.

[25] Lockheed Martin. *Joint Strike Fighter, Air Vehicle, C++ Coding Standard*. Lockheed Martin, December 2005.

[26] B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[27] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: balancing extensibility and modular typechecking. In *OOPSLA '03*, pages 224–240, New York, NY, USA, 2003. ACM Press.

[28] T. D. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP '99*, volume 1628 of *LNCS*, pages 279–303, London, UK, 1999. Springer-Verlag.

[29] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03*, volume 2789 of LNCS, pages 214–223. Springer-Verlag, August 2003.

[30] A. Shalit. *The Dylan Reference Manual. 2nd edition*. Apple Press, 1996.

[31] J. Smith. Draft proposal for adding multimethods to C++. Technical Report N1463, JTC1/SC22/WG21 C++ Standards Committee, 2003.

[32] G. L. Steele Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[33] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[34] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[35] B. Stroustrup and G. Dos Reis. Supporting SELL for high-performance computing. In *LCPC'05*, volume 4339 of LNCS, pages 458–465. Springer-Verlag, October 2005.

[36] G. van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., September 2003.

[37] J. Visser. Visitor combination and traversal control. In *OOPSLA '01*, pages 270–282, New York, NY, USA, 2001. ACM Press.

[38] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA '06*, pages 345–362, New York, NY, USA, 2006. ACM Press.

[39] D. Wonnacott. Using accessory functions to generalize dynamic dispatch in single-dispatch object-oriented languages. In *COOTS*, pages 93–102. USENIX COOTS, 2001.

[40] A. Wöß, M. Löberbauer, and H. Mössenböck. LL(1) conflict resolution in a recursive descent compiler generator. In *JMLC'03*, volume 2789 of LNCS, pages 192–201. Springer-Verlag, 2003.

[41] www.fourcc.org. Video codec and pixel format definition, February 2007.