Science of Computer Programming 75 (2010) 638–667

Contents lists available at ScienceDirect

Science of Computer Programming

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

Design and evaluation of C++ open multi-methods☆

Peter Pirkelbauer*, Yuriy Solodkyy*, Bjarne Stroustrup

Texas A&M University, College Station, TAMU 3112, TX 77843, USA

ARTICLE INFO

Article history:
Received 2 February 2008
Received in revised form 13 June 2009
Accepted 25 June 2009
Available online 7 July 2009

Keywords:
Multi-methods
Open-methods
Multiple dispatch
Binary method problem
Augmenting method problem
Object-oriented programming
Generic programming
C++

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, performance, programming guidelines, and experiences of working with 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. 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 work-around 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, is comparable in memory usage, and runs significantly faster. In particular, the runtime cost of calling an open multi-method 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.

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++ [52], Eiffel [38], Java [6], Simula [11], and Smalltalk [30]) 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 [15]. A separate 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 two problems by separating operations from classes and enabling the choice of dynamic vs. static dispatch on a per-parameter basis.

Work-arounds for both of these problems exist for single-dispatch languages. In particular, the visitor pattern (double dispatch) [28] circumvents these problems without compromising type safety. Using the visitor pattern, the class designer

☆ This paper is an expanded version of the paper that was presented at GPCE 2007.

* Corresponding author. Tel.: +1 979 845 2938.
E-mail addresses: peter.pirkelbauer@tamu.edu (P. Pirkelbauer), yuriys@cs.tamu.edu (Y. Solodkyy), bs@cs.tamu.edu (B. Stroustrup).
URLs: http://www.parasol.tamu.edu/~peterp/ (P. Pirkelbauer), http://www.parasol.tamu.edu/~yuriys/ (Y. Solodkyy), http://www.parasol.tamu.edu/~bs/ (B. Stroustrup).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.06.002

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 [20]. In [55], Visser presents a possible solution to the extensibility problem in the context of visitor combinators, which make use of runtime type information (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* [57], *arbitrary multi-methods* [41], or "*open-methods*". Languages supporting multiple dispatch include CLOS [50], MultiJava [20,40], Dylan [47], and Cecil [17]). 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 arguments 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 [16]. It differs radically from C++'s symmetric function overload resolution rules and does not catch ambiguities.

We derive our design goals for the open-method extension from the C++ design principles outlined in [51, Section 4]. For open-methods, this means the following: open-methods should address several specific problems, be more convenient to use than all work-arounds (e.g. the visitor pattern), and 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. [37]), and parallel to the virtual and overload resolution semantics.

The contributions of this paper include:

- A design of open-methods that is consistent 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.

Section 2 presents application domains for both open-methods and multi-methods as well as discuss the style of programming and the new techniques that open-methods enable. Section 3 describes our function call and ambiguity resolution mechanisms. Section 4 explains our design decisions, and compares their trade-offs in the context of (dynamically linked) libraries to trade-offs made by other researchers. Section 5 discusses the relationship of open-methods to other language features. Section 6 shows the necessary modifications to the C++ compiler and linker model as well as extensions of the IA-64 object model [22] based on our implementation. Section 7 gives an overview of research in the area of multi-methods for C++. Section 8 compares the performance of our approach to other methods that add support for multi-methods to C++. Section 9 compares open-methods to the visitor pattern in two real-world applications. Section 10 summarizes our contributions and sketches remaining open problems.

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 like all style questions it is not a question that can in general be settled without examples and data. This is why in the context of this paper we start with presenting examples that we consider characteristic for larger classes of problems and that would benefit significantly. We then explain fundamentals that drive the design, provide the details of our implementation and compare its performance to alternative solutions. In what follows, we mark examples with 1 when they primarily demonstrate multiple dispatch and with 2 when they demonstrate open-class extensions.

2.1. *Shape intersection*[1]

An intersect operation is a classical example of multi-method usage [51, Section 13.8]. 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.

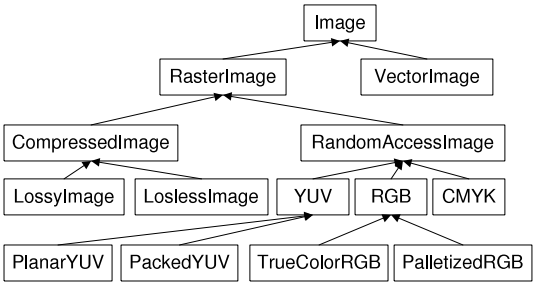
Using the multi-method syntax from [51, Section 13.8], with **virtual** indicating runtime dispatch, we can write:

```
bool intersect (virtual const Shape&,     virtual const Shape& ); // open—method
bool intersect (virtual const Rectangle&, virtual const 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[1 2]

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 cannot 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. An 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 [33,60]. In Section 9, we present an implementation of this example; here we simply demonstrate with a call:

```
bool convert(virtual const image& src, virtual image& dst);

RGB24 bmp("image.bmp");
JPEG jpeg;
convert(bmp,jpeg);
```

2.3. Type conversions in scripting languages[1 2]

Similar to Section 2.2, this example demonstrates the benefits of open-methods in the context of type conversions. Languages used for scripting are often dynamically typed and a value may often be converted to other types depending on use. For example, variable x initialized as string can be used in contexts where integers or even dates are expected, while variable y initialized as integer can perfectly be used inside the catenation of strings. In such cases, an interpreter will try to convert actual values to the type required in the context according to some conversion rules. A typical implementation of such conversion will use either nested switch statements or a table of pointers to appropriate conversion routines. None of these approaches is extensible or easy to maintain. However, multi-methods provide a natural mechanism for such implementations:

```
class ScriptableObject              { ~virtual ScriptableObject(); };
class Number  : ScriptableObject {};
class Integer   : Number            {};
class String    : ScriptableObject {};

void convert (virtual const ScriptableObject& src, virtual ScriptableObject& tgt);
void convert (virtual const Number&           src, virtual String&           tgt);
void convert (virtual const String&           src, virtual Number&           tgt);
// ...   etc.
```

2.4. Compiler pass over an AST[2]

High-level source-to-source transformation infrastructures [46,53] typically use abstract syntax trees (ASTs) to represent programs. Using OOP, the commonalities of the AST classes can be factored in an OO-hierarchy. Then, programmers can write runtime polymorphic code for a family of classes by using pointers/references to a common base class.

```
struct Expr                          { virtual ~Expr(); };
struct UnaryExpr        : Expr       { Expr& operand; };
struct NotExpr          : UnaryExpr {};
struct ComplementExpr : UnaryExpr   {};

const Expr& propagate_constants(virtual const Expr& e);
```

For example, an expression class (i.e. Expr) would be the common base for unary (e.g. not or complement) and binary expressions (e.g. add or multiplication). Analysis or transformation passes that take the semantics of the expression into account (e.g. for propagating constants) need to uncover the real type. Typical implementations rely on the visitor pattern or type-tags to uncover the concrete type. Open-methods are a non-intrusive alternative for writing these compiler passes. In Section 9, we discuss our experience in implementing such a pass with open-methods and visitors.

2.5. Binary method problem[1]

Often times we have a two-argument method, whose meaning is trivial to define when both arguments are of the same type, but not so obvious in cases when arguments are of different types, though related through inheritance. Such methods are characteristic to many logical and arithmetic operations, and have been vigorously studied by Bruce et al. [15]. They define a *binary method* of some object of type τ as a method that has an argument of the same type τ. Binary methods pose a typing problem, and among different solutions to the problem, the authors propose to use multi-methods.

In the multi-methods setting, binary methods simply become multi-methods with two arguments of the same type. Consider for example an equality comparison of two objects:

```
class Point { double x, y; };

bool equal(const Point& a, const Point& b) {
  return a.x == b.x && a.y == b.y;
}

class ColorPoint : Point { Color c; };
```

When a class ColorPoint derives from Point and adds a color property, the question arises on how equal should be defined: should it just compare the coordinates or should it also compare the color properties of both arguments? The second option is only viable if both arguments are of type ColorPoint. If the argument types differ, we can choose either to return false or to compare the coordinates only. Depending on the problem domain both choices can be acceptable. Here we simply note that multi-methods are an ideal solution for the implementation of the latter policy where the comparison of Point with ColorPoint only compares coordinates:

```
bool equal(virtual const Point&      a, virtual const Point&      b);
bool equal(virtual const ColorPoint& a, virtual const ColorPoint& b);
```

2.6. Selection of optimal algorithm based on dynamic properties of objects[1]

Often, we can use dynamic types to choose a better algorithm for an operation than would be possible using only static information. Using open-methods we can use the dynamic type information to select more efficient algorithms at runtime without added complexity or particular foresight. Consider a matrix library providing algorithms optimized for matrices with specific dynamic properties. Storing these dynamic properties as object attributes is not easily extensible and is error prone in practice. Letting the compiler track them using open-methods for dispatch (runtime algorithm selection) is simpler. For instance, the result of $A * A^T$ is a symmetric matrix—if such a matrix appears somewhere in computations, we may consider a broader set of algorithms when the result is used in other computations.

```
class Matrix                    { virtual ~Matrix(); };
class SymMatrix  : Matrix       {}; // symmetric matrix
class DiagMatrix : SymMatrix    {}; // diagonal matrix

Matrix&    operator+(virtual const Matrix&    a, virtual const Matrix&    b);
SymMatrix& operator+(virtual const SymMatrix& a, virtual const SymMatrix& b);
DiagMatrix& operator+(virtual const DiagMatrix& a, virtual const DiagMatrix& b);
```

Depending on the runtime type of the arguments, the most specific addition algorithm is selected at runtime and the most specific result type returned. The static result type would still be Matrix& when the static type of an argument is a Matrix& since we cannot draw a more precise conclusion about the dynamic type of the result (see Section 3.4 and Section 4.2 for details). However, since the operator is selected according to the dynamic type, the optimal algorithm will be used for the result when it is part of a larger expression.

Other interesting properties to exploit include whether the matrix is upper/lower triangular, diagonal, unitary, non-singular, or symmetric/Hermitian positive definite. Physical representations of those matrices may also take advantage of the knowledge about the structure of a particular matrix and use less space for storing the matrix.

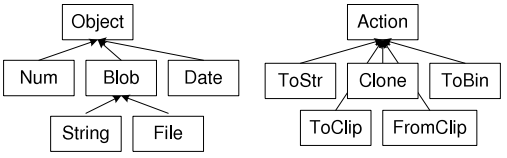
The polymorphic nature of the multiple dispatch requires the result to be returned by either a reference or a pointer to avoid slicing. Since the reference must refer to a dynamically allocated object, this creates a lifetime problem for that object. Common approaches to such problems include relying on a garbage collector and using a proxy to manage the lifetime. An efficient proxy is easy to write:

```
// A memory—managing proxy class (note lowercase name).
class matrix
{
    std:: unique_ptr<Matrix> the_matrix; // pointer to the actual polymorphic Matrix

    matrix(Matrix& actual) : the_matrix(&actual) {}
};

matrix operator+(const matrix& a, const matrix& b)
{
    // Forward operator+ to the actual open—method and
    // attach the result to the proxy.
    return matrix(∗a.the_matrix + ∗b.the_matrix);
}
```

The unique_ptr is a simple and efficient (not-reference counting) "smart" pointer that is part of the C++0x standard [9] and has been widely available and used for years [8].

2.7. Action systems[1]

Dynamically typed languages, such as Smalltalk [30], Ruby [54] or Python [45], can dispatch polymorphic calls on classes not bound by inheritance. As long as a method with a given name exists in the class, it will be called; otherwise an exceptional action is taken. Often, similar behavior is desirable in statically typed languages with possible restriction to objects derived from a certain base class. To achieve this, we may represent methods (here called actions) as objects and then apply a given action to a given set of parameters.



The above figure shows a class hierarchy with objects and actions. Note that the same action applied to a different object may have a completely different meaning.

```
Object& execute(virtual const Action&     act, virtual Object&   obj);
String& execute(virtual const ToString&   act, virtual Number& obj);
  File& execute(virtual const SaveToFile& act, virtual Blob&     obj);
// ... etc.
```

Action objects resemble function objects in C++ in a way. The main difference between actions and C++ function objects is that actions participate in call dispatching on equal bases with other arguments, while function objects invariably define the scope for call dispatching. Simply put, this means that in case of action objects, other arguments of a call can affect the choice of a call's target at runtime (symmetric behavior), while with function objects they cannot (asymmetric behavior).

2.8. Extending classes with operations[2]

Once defined, the object-oriented way to extend a class's functionality is to derive a new class and introduce the new behavior there. However, this technique only succeeds if the programmer has control over the source code that instantiates objects (for example, if the code has been designed to use a factory). Consider a system framework that responds to various events. The events may require logging in different logs and formats. While it is feasible to provide a common interface for different kinds of logs, it is rather difficult to foresee all possible formats in which logging can be done. Open-methods eliminate the need to modify class declarations directly, and improve the support for separation of concerns.

```
struct Log                          {};                          // Interface to different logs
struct FileLog           : Log      {};                          // Logs to File
struct EventLog          : Log      {};                          // Logs to OS event log
struct DebugLog          : Log      {};                          // Logs to debug output

struct Event                        { virtual ~Event(); };       // Interface for various types of events
struct Access            : Event    {};
struct FileAccess        : Access   {};
struct DirectoryAccess   : Access   {};
struct DatabaseAccess    : Access   {};
```

```
// Specializations of how to log various types of events in text format
void log_as_text   (virtual Access&          evt, Log& log, int priority );
void log_as_text   (virtual DirectoryAccess& evt, Log& log, int priority );

// Specializations of how to log various types of events in XML format
void log_as_xml    (virtual Access&          evt, Log& log, int priority );
void log_as_xml    (virtual DirectoryAccess& evt, Log& log, int priority );

// Specializations of how to log various types of events in binary format
void log_as_binary (virtual Access&          evt, Log& log, int priority );
void log_as_binary (virtual DirectoryAccess& evt, Log& log, int priority );

// etc. for any other formats that may be required in the future.
```

2.9. Open-methods programming

The benefits noted in the examples stem from fundamental advantages offered by open-methods. They allow us to approximate widely accepted programming principles better than more conventional language constructs and allow us to see conventional solutions, such as visitors, as work-around techniques. For examples like the ones presented above, open-methods simply express the design more directly. From a programmer's point of view, open-methods

- *are non-intrusive*: We can provide runtime dispatch (a virtual function) for objects of a class without modifying the definition of that class or its derived classes. Using open-methods implies less nonessential coupling than conventional alternatives.
- *provide order independence for arguments*: The rules for an argument are independent of whether it is the first, second, third, or whatever argument. The first argument of a member function (the this pointer) is special only in that it has notational support. The dynamic/static resolution choice has become independent of the choice of argument order.
- *improve ambiguity detection*: Ambiguous calls are detected in a way that cannot be done with only a per-argument check (as for conventional multiple dynamic dispatch) or only a per-translation-unit check (as for conventional static checking). Using open-methods there is simply more information for the compiler and linker to use.
- *provide multiple dynamic dispatch*: We can directly select a function based on multiple dynamic types; no work-arounds are required.
- *improve performance*: faster than work-arounds when more than one dynamic type is involved with no memory overhead compared to popular work-around techniques (see Section 8).

From a language design point of view, open-methods make the rules for overriding and overloading more orthogonal. This simplifies language learning, programming, reasoning about programs, and maintenance.
Potential objections to the use of open-methods include that:

- the set of operations on objects of a class are not defined within the class. However, that is true as soon as you allow any free-standing function and is essential for conventional mathematical notation and programming styles based on that. Information hiding is not affected.
- the first argument is not fundamentally different so open-methods do not obey the "send a message to an object" ("object-oriented") model of programming. We consider that model unrealistically restrictive for many application domains, such as classical math [51].
- the set of overriders for a virtual function is not found within a specific set of classes (the set of classes derived from the class that introduced the virtual function). On the other hand, we never have to define a new derived class just to be able to override.
- open-methods are open; that is, they do not provide a closed set of overloading candidates for a given function name. However, we consider that a good feature in that it allows for non-intrusive extension. For C++, the decision not to syntactically distinguish overriders or overloaded functions was taken in 1983 and cannot be changed now [51, Section 11.2.4].

Obviously, we consider open-methods a significant net gain compared to alternatives, but the final proof (as far as proofs are possible when it comes to the value of programming language features) will have to wait for the application of open-methods in several large real-world programs.

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):

Definition 1. An *open-method* is a free-standing function with one or more parameters declared virtual.

Definition 2. An open-method f_2 *overrides* an open-method f_1 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. In such a case, we say that f_2 is an *overrider* of f_1.

Definition 3. An open-method that does not override another open-method is called a *base-method*.

Definition 4. A base-method together with all the open-methods that override it forms an *open-method family*.

While this is not strictly necessary, for practical reasons we require that a base-method should be declared before any of its overriders. This parallels other C++ rules and greatly simplifies compilation. This restriction does not prevent us from declaring different overriders in different translation units. For every overrider and base-method pair, the compiler checks if the exception specifications and covariant return type (if present) comply with the semantics defined for virtual functions.

Definition 5. 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.

The following example demonstrates a simple class hierarchy and an open-method defined on it:

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

void print(virtual A&, virtual A&); // (1) base—method
void print(virtual B&, virtual A&); // (2) overrider
void print(virtual B&, virtual B&); // (3) overrider
```

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 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) base—method
void print(virtual Y&, virtual Y&); // (5) overrider
```

Here (4) defines a new open-method print on the class hierarchy rooted in X. Y inherits from both A and X, and according to our definition (5) overrides both (4) and (1).

We note that whether it would be better to require an overrider to be explicitly specified as such is an orthogonal decision beyond the scope of this paper. Here we simply follow the C++ tradition set up by virtual functions to do this implicitly.

3.1. Type checking and call resolution of open-methods

Type checking and resolving calls to open-methods involves three stages: compile time, link time, and runtime.

- *Overload resolution at compile time*: the goal of overload resolution is to find a unique open-method in the overload set visible at the call site, through which the call can be (but not necessarily will be) dispatched. The open-method determines the necessary casts of the arguments, and the return type expected at the call site.
- *Ambiguity resolution at link time*: the pre-linker aggregates all overriders of a given open-method family, checks them for return type consistency, performs ambiguity resolution, and builds the dispatch tables.
- *Dynamic dispatch at runtime*: the dispatch mechanism looks up the entry in the dispatch table that contains the most specific overrider for the dynamic types of the arguments and invokes that overrider.

This three-stage approach parallels the resolution to the equivalent modular-checking problem for template calls using concepts in C++0x [31]. 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 [7].

3.2. Overload resolution

The purpose of overload resolution in the context of open multi-methods is to identify an open-method that the compiler will use for type checking and inferring the result type expected from the call. In general, the C++ overload resolution rules [34] remain unchanged: the viable set includes both open-methods and regular functions and the compiler treats them equally. Once a unique best match is found, the call can be type checked against it. For the dispatch, any of its base-methods can be chosen. Which one is selected is irrelevant as any further overrider would likewise override all base-methods.

Consider the following example:

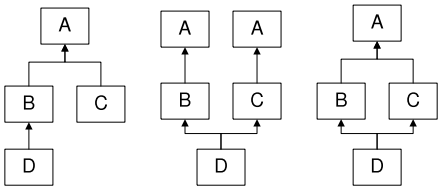
```
struct X { virtual ~X(); };
struct Y { virtual ~Y(); };
struct Z { virtual ~Z(); };

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

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

void foo(virtual XY&, virtual Y&);   // (4) overrides 1 and 2
void foo(virtual Y&,   virtual YZ&); // (5) overrides 2 and 3
```

A call foo(xy,yz) is ambiguous according to the standard overload resolution rules as overriders 4 and 5 are equally good matches. To resolve this ambiguity, a user may explicitly cast some or all of the arguments to make the call unambiguous accordingly to the overload resolution rules: e.g. calling foo(xy,**static_cast**<Y&>(yz)) will uniquely select 4 as a base-method for the call. Alternatively, a user may introduce a new overrider **void** foo(**virtual** XY&, **virtual** YZ&), which will become a unique best match for the call.

3.3. Ambiguity resolution

Once we are in the ambiguity resolution phase done by the pre-linker, we assume that the overload resolution phase has selected a unique best match for type checking of each open-method call site (otherwise it would have reported a compile-time error). At this phase we have information about all available overriders of a particular open-method family, and we only report ambiguities that prevent us from building a complete dispatch table.

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 regardless of the static types of the arguments at the call site.
- 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 Section 3.5. Before looking at that algorithm, we present some key motivating examples.

3.3.1. Single inheritance

In object models supporting single inheritance (Section 3.3), ambiguities can only occur with open-methods taking at least two virtual parameters. Such ambiguities can only be introduced by new overriders, not by extending the class hierarchy. They can be resolved by introducing a new overrider. Open-methods with one dynamic argument are identical to virtual functions and are always ambiguity free. 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.

3.3.2. Repeated inheritance

Consider the repeated-inheritance case (Section 3.3) together with this set of open-methods visible at a call site to foo(d1,d2), where d1 and d2 are of type D&:

```
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 though overriders for all possible combinations of B and C (the base classes of D) are declared, the call with two arguments of type D gets rejected at compile time. The problem in this case is that there are multiple sub-objects of type A inside D.

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 eliminates the need to choose a sub-object. 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 Section 6)) 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 Section 3.3 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 means "D along the B branch". This resolution exactly matches our ideals.

Analogously to single inheritance, extending a class hierarchy using repeated inheritance cannot introduce ambiguities. Ambiguous sub-objects are determined at compile time and reported as errors.

3.3.3. Virtual inheritance

Consider the virtual-inheritance class hierarchy from Section 3.3 together with the set of open-methods from Section 3.3.2: 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 in such a way that we cannot catch it at compile time, because D's definition could be in a different translation unit:

```
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 Section 3.5) 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 entire set of classes and overriders is known.

3.4. 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 work-around techniques (Section 9.2).

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–base-method relationship between two open-methods. Liskov's substitution principle [36] guarantees that any call type-checked based on a base-method can use overrider's covariant result without compromising type safety.

This can be used to eliminate what would otherwise have been ambiguities. Consider the class hierarchies A ← B ← C and R1 ← R2 ← 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, a closer look reveals that one of the choices is simply not type safe: a call to foo(b,b), type-checked against R3* foo(B&,A&) at compile time, would expect a pointer to an object of type R3 (or any of its sub-classes) returned, which R2 is not. This is why R2* foo(A&,B&) cannot be used for dispatching such a call. On the other hand, the same call type-checked against R2* foo(A&,B&) elsewhere is expecting a pointer to R2 (or any of its sub-classes) returned from the call, and hence would readily accept R3. This is why selecting R3* foo(B&,A&) is the only viable choice here, which consequently resolves the ambiguity.

From a pure implementational point of view, 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 dispatch 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 the 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 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.

3.5. Algorithm for dispatch table generation

Let us assume we have a multi-method $rf(h_1, h_2, \ldots, h_k)$ with k virtual arguments. Class h_i is a base of the hierarchy of the ith argument. $H_i = \{c : c <: h_i\}$ is a set of all classes from the hierarchy rooted at h_i. $X_f = H_1 \times H_2 \times \cdots \times H_k$ is the set of all possible argument type-tuples of f. Set $Y_f = \{\langle y_1, y_2, \ldots, y_k\rangle\} \subseteq X_f$ is the set of argument type-tuples, on which the user defined overriders f_j for f. The set $O_f = \{f_0, \ldots, f_{m-1}\}$ is the set of those overriders ($f_0 \equiv f$). $R = \{r_i | r_i f_i(y_1, y_2, \ldots, y_k)\}$ is the set of return types of all the overriders. A mapping $F_f : Y_f \leftrightarrow O_f$ is a bijection between type-tuples on which overriders are defined and the overriders themselves. A function $R_f : Y_f \leftrightarrow R$ maps an argument tuple of an overrider to the return type of that overrider.

Because different derivation paths may get different entries in the dispatch table, we assume that x_i in the type-tuple $x = \langle x_1, \ldots, x_k \rangle$ identifies not only the concrete type, but also a particular derivation path for it (see [56] for formal definitions). Under this assumption, we define $\beta(x_i)$ 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 Section 3.3, $\beta(\mathsf{D/B}) = \mathsf{B}$, $\beta(\mathsf{D/C}) = \mathsf{C}$, $\beta(\mathsf{C}) = \mathsf{A}$, while for the virtual- inheritance hierarchy $\beta(\mathsf{D/A}) = \mathsf{A}$, $\beta(\mathsf{B}) = \mathsf{A}$, $\beta(\mathsf{C}) = \mathsf{A}$.

For the sake of convenience, we define:

$$\beta_i(x) \equiv \langle x_1, \ldots, \beta(x_i), \ldots, x_k \rangle, \quad \text{when } \beta(x_i) \text{ exists.}$$

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

$$\beta(x) \equiv \{\beta_i(x) \mid \beta_i(x) \text{ exists}, i = 1, k\}.$$

$P(X_f, <_P) : \langle x_1, \ldots, x_k \rangle <_P \langle y_1, \ldots, y_k \rangle \Leftrightarrow \forall i : x_i <: y_i \wedge \exists j : y_j \not<: x_j$ defines a partial ordering that models the ordering of viable functions for overload resolution as defined in [34]. $most\_specific\_arg(S) = \{s \in S \subseteq Y_f : \nexists t \in S : t <_P s\}$ is the set of the most specific (refined) argument tuples of S with respect to the partial ordering P. $most\_specific\_res(S) = \{s \in S \subseteq Y_f : \nexists t \in S : R_f(t) <: R_f(s)\}$ is the set of the most specific (refined) argument tuples of S with respect to sub-classing relation $<:$ on result types.

Dispatch table DT_f is a mapping $DT_f : X_f \rightarrow Y_f$ that maps all possible argument tuples to the argument tuples of overriders used for handling such a call.

For any combination of argument types $x \in X_f$, we recursively define entries of the dispatch table DT_f as follows:

$$DT_f[x] = \begin{cases} x, x \in Y_f \\ DT_f[s], s \in S = most\_specific\_res(most\_specific\_arg(\{DT_f(y) | y \in \beta(x)\})) \wedge |S| = 1 \\ \text{Ambiguity}, \quad \text{otherwise.} \end{cases}$$

The above recursion exhibits optimal substructure and has overlapping sub-problems, which lets us use dynamic programming [23] to create an efficient algorithm for generation of the dispatch table, as shown in Algorithm 1.

To demonstrate with an example, consider a simple class hierarchy with two classes A and B, where B derives from A, and two open-methods defined on the argument tuples $\langle \mathsf{A}, \mathsf{A} \rangle$ and $\langle \mathsf{A}, \mathsf{B} \rangle$. In this scenario $X_f = \{\langle \mathsf{A}, \mathsf{A} \rangle, \langle \mathsf{B}, \mathsf{A} \rangle, \langle \mathsf{A}, \mathsf{B} \rangle, \langle \mathsf{B}, \mathsf{B} \rangle\}$ with the following relations that hold on these argument tuples: $\langle \mathsf{A}, \mathsf{B} \rangle <_P \langle \mathsf{A}, \mathsf{A} \rangle$; $\langle \mathsf{B}, \mathsf{A} \rangle <_P \langle \mathsf{A}, \mathsf{A} \rangle$; $\langle \mathsf{B}, \mathsf{B} \rangle <_P \langle \mathsf{B}, \mathsf{A} \rangle$; $\langle \mathsf{B}, \mathsf{B} \rangle <_P \langle \mathsf{A}, \mathsf{B} \rangle$; $\langle \mathsf{B}, \mathsf{B} \rangle <_P \langle \mathsf{A}, \mathsf{A} \rangle$;

The reverse topological order of elements in X_f would thus match the order in which we listed tuples in X_f. Sets of immediate ancestors with respect to $<_P$ would be:

$$\beta(x) = \begin{cases} \emptyset, x = \langle \mathsf{A}, \mathsf{A} \rangle \\ \{\langle \mathsf{A}, \mathsf{A} \rangle\}, x = \langle \mathsf{A}, \mathsf{B} \rangle \\ \{\langle \mathsf{A}, \mathsf{A} \rangle\}, x = \langle \mathsf{B}, \mathsf{A} \rangle \\ \{\langle \mathsf{A}, \mathsf{B} \rangle, \langle \mathsf{B}, \mathsf{A} \rangle\}, x = \langle \mathsf{B}, \mathsf{B} \rangle. \end{cases}$$

Note that the empty set of immediate ancestors is only possible on the tuple that starts the open-method hierarchy, where we by definition would always have an overrider—the base-method. A set of argument tuples of overriders $Y_f = \{\langle \mathsf{A}, \mathsf{A} \rangle, \langle \mathsf{A}, \mathsf{B} \rangle\}$ and thus we can directly set $DT_f[\langle \mathsf{A}, \mathsf{A} \rangle] = \langle \mathsf{A}, \mathsf{A} \rangle$ and $DT_f[\langle \mathsf{A}, \mathsf{B} \rangle] = \langle \mathsf{A}, \mathsf{B} \rangle$.

Now to fill in $DT_f[\langle \mathsf{B}, \mathsf{A} \rangle]$, where $\langle \mathsf{B}, \mathsf{A} \rangle$ is the first element in reverse topological order of X_f that is not in Y_f, we take the set of its immediate ancestors $\beta(\langle \mathsf{B}, \mathsf{A} \rangle) = \{\langle \mathsf{A}, \mathsf{A} \rangle\}$, and since there is only one, there cannot be a better match and thus $DT_f[\langle \mathsf{B}, \mathsf{A} \rangle] \leftarrow DT_f[\langle \mathsf{A}, \mathsf{A} \rangle] = \langle \mathsf{A}, \mathsf{A} \rangle$.

Similarly, to fill in the remaining $DT_f[\langle \mathsf{B}, \mathsf{B} \rangle]$ that comes last in the reverse topological order, we look at its immediate ancestors $\beta(\langle \mathsf{B}, \mathsf{B} \rangle) = \{\langle \mathsf{A}, \mathsf{B} \rangle, \langle \mathsf{B}, \mathsf{A} \rangle\}$ and compare the overriders used for them: $DT_f[\langle \mathsf{A}, \mathsf{B} \rangle] = \langle \mathsf{A}, \mathsf{B} \rangle <_P \langle \mathsf{A}, \mathsf{A} \rangle = DT_f[\langle \mathsf{B}, \mathsf{A} \rangle]$. Thus we propagate the most specific overrider: $DT_f[\langle \mathsf{B}, \mathsf{B} \rangle] \leftarrow DT_f[\langle \mathsf{A}, \mathsf{B} \rangle] = \langle \mathsf{A}, \mathsf{B} \rangle$.

To analyze its performance, we first note that comparison of two type-tuples from X_f can be done in time $O(k)$. If $n = max(|H_i|, i = 1, k)$ and $v = max(v_i, i = 1, k)$ (where v_i is a maximum number of times h_i is used as non-virtual base class in any class of hierarchy H_i) then $|X_f| <= (n * v)^k$ and the amount of edges for topological sort is less then $k * (n * v)^k$. Therefore the complexity of topologically sorting X_f is $O(k * n^k)$. The inner for-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.

3.6. Alternative dispatch semantics

While our goal is to unify virtual function dispatch and overload resolution into an open-methods semantics, this is not always possible. Consider for example the repeated-inheritance class hierarchy from Section 3.3 with a virtual function added:

Algorithm 1 Dispatch Table Generation

$T \leftarrow$ topological_sort(X_f) // Topologically sort according to $<_P$
$S \leftarrow$ reverse(T) // Reverse the order to have the least specific first
for all $x \in S$ **do**
  if $x \in Y_f$ **then**
    $DT_f[x] \leftarrow x$ // Overriders themselves are the best matches for their arguments
  else
    $ancestors = \beta(x)$ // Get type-tuples of immediate ancestors
    $most\_specific = \{DT_f[\text{extract\_any}(ancestors)]\}$
    while $\neg$ empty($ancestors$) **do**
      $a \leftarrow$ extract_any($ancestors$)
      $dominated \leftarrow false$
      for all $e \in most\_specific$ **do**
        if $DT_f[a] <_P DT_f[e]$ **then**
          // This ancestor's overrider is more specific
          $most\_specific \leftarrow most\_specific - \{e\}$
        else if $DT_f[e] <_P DT_f[a]$ **then**
          // Overrider in the most_specific set is more specific
          $dominated \leftarrow true$
          break
        else if $R_f(DT_f[a]) <: R_f(DT_f[e])$ **then**
          // Incomparable by arguments, but more specific return type
          $most\_specific \leftarrow most\_specific - \{e\}$
        else if $R_f(DT_f[e]) <: R_f(DT_f[a])$ **then**
          // Incomparable by arguments, but ancestor's return type is less specific
          $dominated \leftarrow true$
          break
        end if
      end for
      if $\neg$ $dominated$ **then**
        // When none of the overriders was more specific
        $most\_specific \leftarrow most\_specific \cup \{DT_f[a]\}$
      end if
    end while
    if $|most\_specific| = 1$ **then**
      // There was a unique most specific overrider, use it
      $DT_f[x] \leftarrow y$, where $most\_specific = \{y\}$
    else
      Error: Unable to find unique best overrider among *most_specific* for handling x
    end if
  end if
end for

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

void bar(A&);              // overloaded function
void bar(C&);              // overloaded function

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:
db.foo();                  // calls A::foo
dc.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
```

Virtual dispatch semantics and overload resolution semantics go different ways in this case. Since the two language features are not entirely orthogonal, we had to decide which semantics to follow.

From a technical point of view, both semantics can be implemented for open multi-methods. The reason we decided not to model the semantics after overload resolution in this case is that the resulting cross-casting behavior could have been surprising to the user due to the implicit switching of different sub-objects. On the other hand, the difference between the ordinary virtual function (foo) call and the ordinary overloaded resolution for (bar) in this case 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. This is why 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, the resolution of **static_cast** and **dynamic_cast** can differ even given identical arguments: **dynamic_cast** can use more information than **static_cast**.

Due to our decision to model the semantics after virtual dispatch, we require covariance of the return type on overriders, while had we modeled after overload resolution, we could have only required convertibility of return types.

4. Discussion of design decisions

Type-safety and ambiguities have always been a major concern to systems with multiple open dispatch. One of the first widely known languages to support open-methods was CLOS [50]. CLOS linearizes the class hierarchy and uses asymmetric dispatch semantics to avoid ambiguity. Snyder [49] and Chambers [16,17] observe that silent ambiguity resolution makes errors in programs hard to spot. Therefore, Cecil uses symmetric dispatch semantics and dispenses with object hierarchy linearization in order to expose these errors at compile time. Recent studies [4,27,39,41] explore the trade-offs between multi-methods and modular type-checking in languages with neither a total order of classes nor asymmetric dispatch semantics. In particular, Millstein and Chambers discuss a number of models that embrace or restrict the expressive power of the language to different degrees. The described models range from globally type-checked programs to modularly type-checked units. We will briefly discuss our evaluation of these approaches in the context of C++ later in this section.

This work aims for maximal flexibility and relies on a global type-checking [1] approach for open-methods. We motivate this approach with the goal not only to support object-oriented programming but also to enhance the support for functional and generic programming styles in C++.

The cost of the global type-checking approach is that some ambiguities can be detected late—in particular at the load time of dynamically linked libraries (DLLs). DLLs 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 outline a design addressing the major issues in such a scenario.

Our guiding principle is to support the use cases described in Section 2 with language features that are guaranteed to be type-safe in every scenario. The idea is to report errors as long as we can assume that ambiguities can be resolved by programmers. Only when it is too late for that do we have to use type-safe resolution mechanisms. This section discusses the design decisions we have made based on three language aspects: ambiguity resolution, covariant return types, and pure (abstract) open-methods.

4.1. Late ambiguities

Late ambiguities are ambiguities that are detected at a stage in the build process when programmer intervention is no longer feasible. They can occur, for example, when classes use virtual inheritance while some definitions necessary to declare a resolving overrider cannot be accessed. Consider the example given in Section 3.3.3. Examples for late ambiguities include:

- the class D was defined as a local class, since the class name would be local to the function scope.
- the class D was defined in an implementation file of a library, but the class definition was not exported in a header file.
- a library defined classes A, B, and C as well as implemented, but did not export, an open-method foo. The definition of D results in a late ambiguity.

In all cases, a programmer could not declare a resolving overrider.

A second source of late ambiguities is when independently developed libraries define conflicting overriders, but the definition of one of the involved classes is not available. Consider the single-inheritance hierarchy of Section 3.3 with an open-method foo(A,A). A library defines, but does not export B and an overrider for foo(B, A), while another library defines C and an overrider for foo(A,C). A call foo(b,c) is ambiguous but cannot be resolved, because the definition of B is not available. Ambiguities that emerge from the use of dynamically linked libraries are always late.

Resolution mechanism for late ambiguities: If there is no unique best match for a possible type-tuple, we choose an overrider from all best matches. Interestingly, any overrider will result in a correct program provided the rest of the program is correct and in principle we could even pick a random overrider from the set of best matches. Nevertheless, the choice is deterministic, but remains unspecified.

Not to specify which overrider we choose among type-safe candidates keeps the resolution mechanism symmetric as no candidate is preferred. The use of a deterministic choice is not strictly necessary, but it allows for reproducibility—the same method will always be selected from a set of candidates.

Consider the following example of image format conversion. For a discussion of the problem and an implementation see Section 2.2 and Section 9.1, respectively. The following code shows a common header file and two independently developed libraries that support additional image formats.

```
// Common header: ImageLibrary.h
struct Image {
  virtual ~Image();
  // ...
};
struct TiffImage : Image { /* ... */ };

void convert(virtual const Image&     from, virtual Image&     to) {... }
void convert(virtual const TiffImage& from, virtual Image&     to) { ... }
void convert(virtual const Image&     from, virtual TiffImage& to) { ... }

// DLL—Jpeg supporting JPEG images
#include "ImageLibrary.h"
struct JpegImage : Image { /* ... */};
void convert(virtual const Image&     from, virtual JpegImage& to) { ... }
void convert(virtual const JpegImage& from, virtual Image&     to) { ... }
void convert(virtual const TiffImage& from, virtual JpegImage& to) { ... }
void convert(virtual const JpegImage& from, virtual TiffImage& to) { ... }

// DLL—Png supporting PNG images
#include "ImageLibrary.h"
struct PngImage : Image { /* ... */};
void convert(virtual const PngImage& from, virtual Image&    to) { ... }
void convert(virtual const Image&    from, virtual PngImage& to) { ... }
void convert(virtual const TiffImage& from, virtual PngImage& to) { ... }
void convert(virtual const PngImage& from, virtual TiffImage& to) { ... }
```

The header file of an image library framework defines two classes (Image, TiffImage), and a base-method convert together with two overriders that implement conversions from TiffImage to a general Image and vice versa. A library (DLL-Jpeg) derives a new type JpegImage from Image and introduces new overriders for convert that handle all possible combinations of known image formats. Likewise, another library (DLL-Png) derives a new class PngImage from Image and introduces similar overriders. Now a call to convert a JpegImage into a PngImage is ambiguous. Libraries DLL-Jpeg and DLL-Png could stem from different vendors that do not know about each other. In systems that use dynamically linked libraries, such problems are hard to predict and design for.

Note that, since the class definitions of the respective other library were not available when DLL-Png and DLL-Jpeg were implemented, neither developer could possibly provide resolving overriders. The question thus arises to which convert should a call convert(JpegImage, PngImage) resolve.

Any overrider (including base-method) has to assume that a dynamic type resolving to Image is an unknown derived type. Consequently, each convert must be written so that it manipulates its arguments of types Image polymorphically (for example, by using virtual functions). This implies that as long as convert's code does not make more assumptions about its arguments than the interface defined in the base-class guarantees, any overrider can be chosen.

Alternative techniques to handle or prevent (late) ambiguities include asymmetric choice, preventive elimination of overriders that could be prone to symmetry, or exceptions that signal an error:

- *System specified choice:* Other systems with open-methods use a specified policy to resolve ambiguities. These involves preferred treatment of overriders that are more specialized on a specified argument (e.g. CLOS [50]) and class hierarchy linearization (CLOS, Dylan [47]). Making the resolution explicit breaks symmetric dispatch, as programmers can write code that exploits the specification.
- *Limit extensibility:* In [39,41], Millstein and Chambers discuss limitations to the type systems that prevent late ambiguities. Their system *M* disallows virtual inheritance across modules. Moreover, open-methods have a specified argument position. Adding overriders across module boundaries is permitted only when the type in that argument is covariant and the type is defined in the same module. MultiJava [20] is based on system *M*. In practice, these limitations have

been found to be overly restrictive (Relaxed MultiJava [21,40] and C++ concepts [35]). In addition, requiring C++ code to comply with the provided inheritance restrictions is not an option.

In [4], Allen et al. develop a different set of restrictions for modular type checking of multiple dispatch for Fortress [3]. Instead of restricting multiple inheritance across modules, the notion of a meet function resolves ambiguities that originate from virtual inheritance. Moreover, their set of restrictions is sensitive to whether a function is a multi-method (defined in class) or an open-method (freestanding function). Overriding open-methods across module boundaries is not possible. Like in System *M*, overriding multi-methods is tied to a single distinguished argument position (the self argument) and the module of the type definition.

Systems that require overriders defined in another module to override a specific argument position with a covariant type defined in that module are unable to handle bidirectional image conversion well. Assuming that the first argument is special, DLL-Jpeg could not provide overriders for conversions to JpegImage.

- *User specified choice:* Parasitic methods as implemented in Java [14] (an implementation for Smalltalk also exists [26]) add an object-oriented flavor to multi-methods and make them an integral part of classes. Multi-methods can be inherited from a base class and overridden (or shadowed) in the derived class. Parasitic methods give the receiver precedence over other arguments. The encapsulation guarantees that a compiler can check for multiple argument ambiguities. Virtual-inheritance ambiguities are implicitly resolved by users, as the resolution is sensitive to the order of multi-method declarations within the class definition.

  In [27], Frost and Millstein unify encapsulated multiple dispatch with predicate dispatch. They replace the dependence on textual order with first match semantics, where later predicates implicitly exclude earlier predicates.

  The global checking presented in this paper resolves fewer virtual-inheritance ambiguities silently than an encapsulated approach would. Moreover, the use of encapsulation requires control over the construction of the receiver object. Even if that can be handled by using a factory approach, this would be unable to solve and would only recast the ambiguity illustrated by the conversion example: Which converter class takes precedence, the one defined by DLL-Jpeg or the one defined by DLL-Png?
- *Glue-methods:* Relaxed Multi-Java [40] resolves ambiguity conflicts by introducing glue-methods (to glue DLL-Jpeg and DLL-Png) that the system-integrator provides. This is a viable solution for software developers integrating several libraries, but it is not a feasible scenario for end-user applications, as dynamically linked modules can be loaded into the process without the direct request of a developer. This is the case for various component object models where applications may request an object by name from the system. The operating system will locate and load the module in which the object resides.
- *Throw an exception:* Some implementations (e.g. Cmm [48]) throw an exception at dispatch time when an ambiguity is encountered. We disagree with this approach because each candidate alone is a type-safe choice and should be able to handle the requested operation. Moreover, this approach forces programmers to consider open-method calls as a potential source for exceptions, while their choice of how to handle this exception is limited and likely will result in program termination.
- *Program termination:* Instead of waiting until runtime, the application can terminate (or fail to link) when ambiguous overriders are detected. We argue analogously to the exception case that termination is an inadequate response for a choice among type-safe operations.

4.2. Consistency of covariant return types

Before we go into a detailed discussion, we would like to point out that the main focus of this section is on the consistency of covariant return types among overriders available at runtime. The use of covariant return type for ambiguity resolution is orthogonal to the problems discussed here and is discussed in detail in Section 3.4.

Different DLLs can specify conflicting covariant return types. Consider a two-class hierarchy A ← B and another two-class hierarchy R1 ← 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&)):

$AA^1 in D_1$	**A**	**B**	$AA^1 in D_2$	**A**	**B**
A	AA^1	AB^2	**A**	AA^1	AA^1
B	AA^1	AB^2	**B**	AA^1	BB^1

When both libraries are linked together, we get the dilemma of how to resolve a call with both arguments of type B. On the 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 a scenario would be rejected 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 ← 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 for the DLL case to weaken 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 Section 3, 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*.

4.3. Pure open-methods

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. When both libraries are (dynamically) linked together the presence of an overrider for the class in D1 cannot 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.

5. Relation to orthogonal features

In this section, we discuss the relationship of open-methods to other language features.

5.1. Namespace

Virtual functions have a class scope and can only be overridden in the derived classes. Open-methods do not have such a scope by default, so the question arises: when should an open-method be considered an overrider and when just a different open-method? Let us look at the following example:

```
namespace X
{
  class A {};
  void bar(virtual A&); // base method

  class B : A {};
  void bar(virtual B&); // (1)
}

namespace Z
{
  void bar(virtual B&); // (2)
}

namespace Y
{
  class D : X::A {};
  void bar(virtual D&); // (3)
}

class C : X::A {};
void bar(virtual C&); // (4)
```

In the presented implementation, an overrider has to be declared in the same namespace as its base-method (1). Open-methods with the same name and compatible parameter types, defined in different namespaces, would not be considered

overriders. The major benefit of this approach is that it is easy to understand and implement. Unfortunately such semantics are not unifiable with overrider declarations of virtual function calls, where derived classes can be declared in a different namespace. **using** declarations present a potential work around to these limitations.

An alternative would be to let overriders be declared in any namespace (1, 2, 3, 4). It is easy to understand, but defeats the purpose of namespaces that were introduced to better structure the code and avoid name-clashes among independently developed modules.

Another alternative may consider an open-method to be an overrider, if its base-method is defined in the same scope or in the scope of their argument types and their base classes. In this scenario (1, 3, 4) would override; (2) would not. Among its advantages is that it closely resembles argument dependent lookup. It would also work for virtual functions. Its downside, however, is that it is harder to comprehend.

5.2. Access privileges

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 particular open-method as a friend.

5.3. Smart pointers

In C++, programmers use smart pointers, such as auto_ptr (in current C++) as well as shared_ptr and weak_ptr (in Boost [59] and C++0x [8]) for resource management. The use of smart pointers together with open-methods is no different from their use with (virtual) member functions. For example:

```
struct A { virtual ~A(); };
struct B : A {};
void foo(virtual A&);

void bar(shared_ptr<A> ptr)
{
  foo(*ptr);
}
```

Defining open-methods directly on smart pointers is not possible. In the following example, (1) yields an error, as ptr1 is neither a reference nor a pointer type. The declaration of (2) is an error, because shared_ptr is not a polymorphic object (it does not define any virtual function). Even when shared_ptr were polymorphic, the open-method declaration would be meaningless. A shared_ptr<B> would not be in an inheritance relationship to shared_ptr<A>, thus the compiler would not recognize foo(**virtual** shared_ptr<B>&) as an overrider.

```
void foo(virtual shared_ptr<A>  ptr1); // (1) error
void foo(virtual shared_ptr<A>& ptr2); // (2) error
```

6. Implementation

We have implemented open-methods as described in Section 3 by modifying the EDG compiler front-end [24]. This includes dispatch table generation and thunk generation for multiple inheritance and covariant return. To reduce the dispatch table size, we have also implemented the dispatch table compression techniques presented in [5]. Our current implementation does not support dynamically linked libraries and detection of late ambiguities.

6.1. Changes to the compiler and linker

Our mechanism extends ideas presented in [25,57] as to the compiler and linker model. We adopted the multi-method syntax proposed in [51], which in turn was inspired by an earlier idea by Doug Lea (see [51, Section 13.8]). 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 print(int, virtual const A&); // ok

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

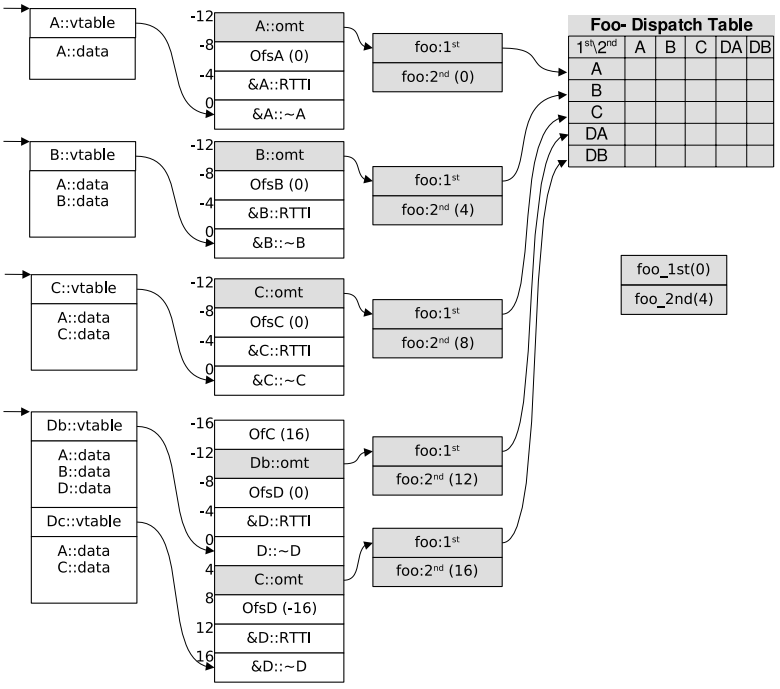


Fig. 1. Object model for repeated inheritance.

For each translation unit, the EDG compiler lowers the high level abstractions in C++ to equivalent code in C. We added an implementation that lowers open-method calls to C according to the object-model presented in Section 6.2. In addition, the compiler puts out an *open-method description* (OMD) file that stores the data needed to generate the runtime data structure discussed in Section 6.2. 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 generate class definitions for arguments to open-methods that are passed by value.

The pre-linker uses Coco/R [58] 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 initializes the data structures described in Section 6.2.

When the call of an overrider requires adjustments of the this-pointers (as is sometimes needed in multiple-inheritance hierarchies), the pre-linker creates thunks and makes the dispatch table entries refer to them instead. During dispatch table synthesis, the linker will report errors for all argument combinations that do not have a unique best overrider. The output of the pre-linking stage is a C-source file containing the missing definitions. If the linker generates a library, the pre-linker also puts out a merged OMD-file.

6.2. Changes to the object model

We augment the IA-64 C++ object model [22] 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 om-table offsets are stored as global variables.

The Figs. 1 and 2 show the layout of objects, v-tables, om-tables and dispatch-tables for repeated and virtual inheritance. Our extensions to the object-model are shown with gray 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 Section 3.3. 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 the corresponding dimension of the dispatch table. Since the size of

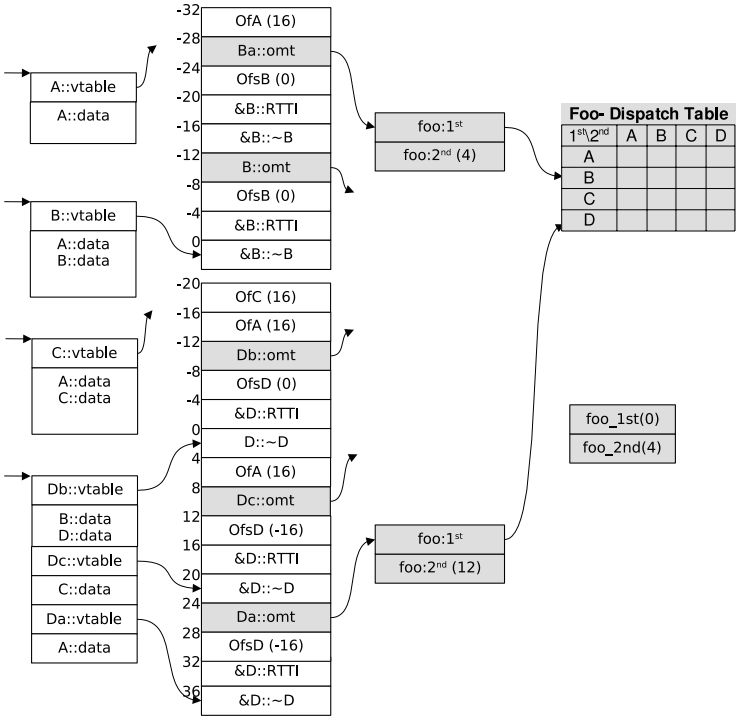


Fig. 2. Object model for virtual inheritance.

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 Figs. 1 and 2) that determines the offset within the om-tables.

Note that these figures depict 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 Section 3.3.2 in the 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 the 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.

6.3. Alternative approaches

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

6.3.1. Multi-methods

Unlike open-methods, multi-methods require the base-method 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.

With the restriction of in-class declarations imposed by multi-methods it seems logical to declare a multi-method either as a member function or as a friend non-member function. Consider:

```
class Matrix
{
 // multi−method declaration as a non−member function
 friend Matrix& operator+(virtual const Matrix& lhs, virtual const Matrix& rhs);

 // equivalent declaration as a member function
 virtual Matrix& operator∗(virtual const Matrix&);
};
```

We implemented only the non-member version of multi-methods. The member version can be implemented with exactly the same techniques. However, in many cases it is harder to write code that uses the member version because an overrider must be a member of (only) one class—and the main rationale for multi-methods is to elegantly deal with combinations of classes. Even the non-member (friend) version is hard to use.

By requiring a declaration to be present in a class, we limit the polymorphic operations to those that the class designer thought of. That requires too much foresight of the class designer or leads to unstable classes (classes that keep having multi-methods added). Such problems are well known in languages relying on member functions. Open-methods provide an abstraction mechanism that solves such problems by separating operations from classes.

6.3.2. Chinese remainders

As we saw in Section 6.2, support of open-methods required an extra indirection via an om-table to get the index of the class in the appropriate argument position. This extra indirection was needed because open-methods are not bound to the class, and as a result, we do not know how many of them a class may have; therefore we cannot reserve entries in the v-table for them. In this section, we present an "ideal" scheme for implementing open-methods, inspired by ideas presented in [29]. The proposed scheme circumvents the necessity for om-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 $I_f : T \times N \to N$ such that for any type $t \in T$ (where T is a domain of all types) and argument position $n \in N$ it returns an index of type t in the nth dimension of the f's dispatch table. If such a function is reasonably fast (preferably constant time) and its range is small (preferably from zero to the maximum number of types that can be used in any argument position) then we can efficiently implement multiple dispatch by properly arranging the rows and columns according to the indices returned by I_f. As in [29], we use the Chinese Remainder theorem [23] to generate the function I_f.

Chinese Remainder Theorem

Let $m_1, \ldots, m_k$ be integers with $gcd(m_i, m_j) = 1$ whenever $i \neq j$. Let m be the product $m = m_1 m_2 \cdots m_k$. Let $a_1, \ldots, a_k$ be integers. Consider the system of congruences:

$$\begin{cases} x \equiv a_1 (\text{mod } m_1) \\ x \equiv a_2 (\text{mod } m_2) \\ \ldots, \\ x \equiv a_k (\text{mod } m_k). \end{cases}$$

Then there exists exactly one $x \in Z_m$ satisfying this system.

Since we may have different class hierarchies in different argument positions, we have to consider each argument position separately. Assuming that there can be q different types $t_{i1}, t_{i2}, \ldots, t_{iq}$ in an argument position i, we may assign a different prime number $m_{ij} : j = 1, q$ to each of them and then according to the Chinese Remainder Theorem find a number x_i that satisfies the above equation. Storing x_i for each dimension (argument position) of the dispatch table, we will come to the dispatching algorithm shown in the listing 2. Since k is known at compile time, no actual iteration is required and the algorithm takes constant time.

Algorithm 2 Dispatching with Chinese Remainders

for all argument positions i of a multi-method f **do**
  $n_i = x_i \mod m_i$
end for
call $D[n_1, \ldots, n_k]$ with arguments provided

In this scenario, every class (or more specifically every argument position i where this class may appear as virtual argument) will have a prime number m_i assigned to it, while the dispatch table will have a number x_i computed through Chinese Remainders, associated with each of its dimensions. The result of $x_i \mod m_i$ gives us the column within the appropriate dimension of the dispatch table.

This dispatching technique has the nice property that it does not need any modifications of the v-table in order to introduce a new open-method on the class. Once allocated, prime numbers can be reused for any number of open-methods defined on the class regardless of the argument position in which a type is used. After dispatch table allocation, we simply have to compute the number x_i for each of the argument positions. Extending such a table, which may be required after the introduction of a new class in the hierarchy, is also simple: allocate new rows and columns and recompute x_i taking the prime numbers of newly added classes into account.

We demonstrate the approach with an example. Consider the following class hierarchy and an open-method foo defined on it:

```
class A           {};        // Assigned prime 2
class B : public A {};       // Assigned prime 5
class C : public A {};       // Assigned prime 3
```

```
class D : public B, public C {};  // Assigned prime 13 for D/B and 7  for D/C sub−object
class E : public D {};            // Assigned prime 17 for E/B and 19 for E/C sub−object

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&);
void foo(virtual E&, virtual E&);
```

The following dispatch table is built:

	$\mathbf{A}^2$	$\mathbf{B}^5$	$\mathbf{C}^3$	$\mathbf{D/B}^{13}$	$\mathbf{D/C}^7$	$\mathbf{E/B}^{11}$	$\mathbf{E/C}^{17}$	
$\mathbf{A}^2$	AA	AA	AA	AA	AA	AA	AA	$x_{\equiv 0(\bmod 2)}$
$\mathbf{B}^5$	AA	BB	BC	BB	BC	BB	BC	$x_{\equiv 1(\bmod 5)}$
$\mathbf{C}^3$	AA	CB	CC	CB	CC	CB	CC	$x_{\equiv 2(\bmod 3)}$
$\mathbf{D/B}^{13}$	AA	BB	BC	BB	BC	BB	BC	$x_{\equiv 3(\bmod 13)}$
$\mathbf{D/C}^7$	AA	CB	CC	CB	CC	CB	CC	$x_{\equiv 4(\bmod 7)}$
$\mathbf{E/B}^{11}$	AA	BB	BC	BB	BC	EE	EE	$x_{\equiv 5(\bmod 11)}$
$\mathbf{E/C}^{17}$	AA	CB	CC	CB	CC	EE	EE	$x_{\equiv 6(\bmod 17)}$
	$x_{\equiv 0(\bmod 2)}$	$x_{\equiv 1(\bmod 5)}$	$x_{\equiv 2(\bmod 3)}$	$x_{\equiv 3(\bmod 13)}$	$x_{\equiv 4(\bmod 7)}$	$x_{\equiv 5(\bmod 11)}$	$x_{\equiv 6(\bmod 17)}$	$x = 1062506$

Dispatching a call will then look like $DT_{foo}[X_{DT_{foo}} \bmod P(a_1), X_{DT_{foo}} \bmod P(a_2)](a_1, a_2)$. Having pointers to the actual arguments of a call, we can look up the v-tables of those arguments and the prime numbers associated with their types. Suppose that the prime number associated with the first argument is 11, while the prime number associated with the second argument is 3. To get the row number inside the dispatch table associated with the first argument, we compute the remainder of dividing 10 62 506 by 11, which is 5. Row number 5 corresponds to the B sub-object of an object with dynamic type E. Similarly, we get the column associated with the type of the second argument through finding the remainder of dividing 10 62 506 by 3, which is 2. Column number 2 corresponds to type C, which means that the dynamic type of the second argument is C. The number 10 62 506 is associated with the dispatch table, through which the call is being dispatched. To find the overrider that will be handling the call, we simply look up an address of the function that is stored at the intersection of the fifth row and the second column, which is foo(B&,C&).

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 each class and perform computations on numbers that are bound by the product of these primes. The product of only the first nine primes fits into a 32-bit integer and the first 15 primes into a 64-bit integer. Table compression techniques [5], or the use of minimal perfect hash functions [23] instead, can help overcome the problem.

We would like to mention that in response to an earlier version of this paper Gabor Greif sent us his unpublished notes on a similar use of Chinese Remainders for implementing multiple dispatch [32] in Dylan.

7. Related work

Programming languages can support multi-methods either through built-in facilities, preprocessor, or 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.

7.1. Cmm

Cmm [48] is a preprocessor-based 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 uses RTTI to recover the dynamic type of objects to identify the best overrider. The other achieves constant time dispatch by relying on a virtual function overridden in each class. Dispatch ambiguities are signaled 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. Call-site virtual dispatch delays the binding to regular overloaded functions, if one of their actual arguments is preceded by the **virtual** keyword.

```
void foo(A&);
void foo(B&); // B derives from A

// call site virtual dispatch
foo(virtual x); // which foo gets invoked depends on the dynamic type of x
```

Cmm does not provide special support for repeated inheritance, and therefore its dispatch technique does not entirely conform to virtual function semantics.

7.2. DoubleCpp

DoubleCpp [10] 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, unlike other visitor-based approaches, reports potential ambiguities.

7.3. Accessory function

The accessory functions papers [25,57] 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. Neither paper provides a detailed discussion of the intricacies when multiple inheritance is involved. The authors do not refer to a model implementation to which we could compare our approach.

7.4. Loki

Loki [2], 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.

7.5. OOLANG

In [43], Panizzi and Pastorelli describe their open-method implementation for OOLANG, a language developed for the Apemille SPMD supercomputer that has a C++ like object model. The paper gives special attention to the handling of covariant return types. OOLANG's system differs from our implementation in the handling of repeated inheritance. Classes that repeatedly inherit from a base-class must define an overrider for each open-method that uses the base-class as type for a virtual parameter. Furthermore, OOLANG does not use covariant return type information for ambiguity resolution.

7.6. Other approaches

Besides the approaches mentioned in Section 4, languages can provide multi-method abstractions through a library (e.g. Python [45]). Chambers and Chen [19] present an alternative implementation technique based on a lookup DAG. Their work generalizes multiple dispatch to be a subset of predicate-based dispatch.

7.7. Multiple dispatch in practice

In order to estimate how often multiple dispatch is used in practice, Muschevici et al. [42] studied programs that utilize dynamic dispatch. The article introduces a language independent model for describing multiple dispatch, and defines six metrics on generic functions (i.e. in C++ an open-method family or a virtual function and its overriders) that measure aspects such as the number of arguments used for dynamic dispatch, the number of overriders in a multi-method family, etc. Using these metrics, the article analyzes nine applications – mostly compilers – written in six different languages: CLOS, Dylan, Cecil, MultiJava, Diesel [18], and Nice [12]. Their results show that 13%–32% of generic functions utilize the dynamic type of a single argument, while 2.7%–6.5% of them utilize the dynamic type of multiple arguments. The remaining 65%–93% of generic functions have a single concrete method, and therefore are not considered to use the dynamic types of their arguments. In addition, the study reports that 2%–20% of generic functions had two and 3%–6% had three concrete function implementations. The numbers decrease rapidly for functions with more concrete overriders. Since the multiple dispatch semantics presented in this paper is in line with the model defined by Muschevici et al., we expect similar results for C++ with open-methods.

8. Results

In order to discuss time and space performance, we compare code generated by our C++ Open Method Compiler, described in Section 6, to a number of prototype implementations, the visitor pattern, Cmm, DoubleCpp, and the Loki library. The prototypes of our design alternatives were implemented in C to approximate the lowering of C++ code to C. They were initially developed to assess performance trade-offs of different approaches and work-around techniques and include open-methods (can be declared freely), multi-methods (have to be declared in class, thus the om-tables can be embedded into the v-table, saving two indirections per argument Section 6.3.1.), and a Chinese Remainder (Section 6.3) based implementation. These implementations lay out the dispatch table for the concrete example described below as C data structures, and then dispatch calls through it.

We wrote 20 classes (representing shapes, etc.) that 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. 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 that 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.

8.1. Implementations

We tested the approaches on a Pentium D, 2.8 GHz running CentOS Linux and a Core2Duo running Mac OSX. The code for the performance tests was compiled with g++ 4.1 (Linux) and gcc 4.0.1 (OSX) with optimization level set to -O3. The C++ Open Method Compiler 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 8 types instead of 20 and the size of the executable is omitted.

For Loki, we only tested the static dispatcher because the others require manual handling of all possible cases. Using other dispatchers would have been closer to a scenario of a 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.

8.2. 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
Multi-methods prototype	42 972	78	60
Open-methods prototype	40 636	82	63
C++ Open-method Compiler	42 504	82	64
Double Cpp	34 812	120	82
C++ Visitor	38 236	132	82
Chinese Remainders prototype	n/a	175	103
Cmm (constant time)	155 344	415	239
Cmm	155 056	1 320	772
Loki Library	75 520	3 670	2 238

Executable size: To obtain a comparable size of the executable, we used the regular EDG frontend to generate C code for the alternative approaches. Then we compiled all intermediate C files with gcc, where optimizations were set to minimize the code size. Moreover, we stripped off the symbols from the executables. The size of the dispatch tables is mentioned as one of the major drawbacks of providing multi-methods as a programming language feature [57]. However, our results reveal that the best achievable code size is roughly the same for visitors, prototyped multi-/open-method, and C++ Open-method Compiler implementations. 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 require as many v-table entries as there are in a dispatch table, 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. 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: The results for prototyped multi-methods, prototyped open-methods, and C++ Open-method Compiler 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 the prototyped multi-methods and open-methods (the comparison with the C++ Open-method Compiler is stated in parentheses) is within the expected range. Four 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 come from experiments designed to highlight the cost of multiple dispatch: the functions invoked hardly do anything. Depending on the application the improved performance may or may 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 reduces the program size compared to brute-force and work-around techniques. Under the assumption that the use of open-methods in C++ would be similar to Muschevici et al.'s results (Section 7.7), we conclude that the size of the dispatch table will remain small for most practical cases.

9. Experiences

In order to compare open-methods with double dispatch and the visitor pattern, we have implemented some of the examples from Section 2.

9.1. Image format conversion

The first example is image conversion. To meet the performance requirements typical for image processing applications, information about the exact source and destination formats is indispensable for an efficient conversion. With this information, we can call a routine geared for that specific pair of formats. Any attempt to work through a common base interface will significantly hinder performance, and should be avoided. This is why we use a fairly shallow class hierarchy to represent different image formats. Another interesting aspect of this example is that when the pair of formats is known statically, it is feasible to write a generic conversion algorithm that relies on some format traits. This is an approach taken by Adobe's GIL library [13]. Therefore the main goal in this example is to uncover the dynamic types of both arguments and pass on these uncovered arguments together with their static types to a set of overloaded template functions.

```
template <class SrcImage, class DstImage>
bool generic_convert(const SrcImage& src, DstImage& dst);

typedef unsigned char color_component;

struct image
{
  // member−functions to access row buffer, width, height etc.
};

struct RGB : image // abstract base of all RGB images
{
  struct color { color_component R, G, B, A; };
  virtual color get_color(int i, int j) const          = 0;
  virtual void  set_color(int i, int j, const color& c) = 0;
};

struct RGB32 : RGB { /*implements get_color, set_color*/};
// ... Similar definitions for RGB24, RGB16, RGB15, RGB08

struct YUV : image // abstract base of all YUV images
{
```

```
  struct color { color_component Y, U, V, A; };
  virtual color get_color(int i, int j) const          = 0;
  virtual void  set_color(int i, int j, const color& c) = 0;
};

struct UYVY : YUV { /*implements get_color, set_color*/};
// ... Similar definitions for YUY2,Y41P,CLJR,YVU9,YV12,IYUV,I420,Y800 etc.

struct CMYK : image
{
  struct color { color_component C, M, Y, K; };
  virtual color get_color(int i, int j) const          = 0;
  virtual void  set_color(int i, int j, const color& c) = 0;
};
```

Open multi-methods to handle the cases can be listed separately from class definitions:

```
// Base open—method. Fails as we do not know anything about the formats
bool convert(virtual const image& src, virtual image& dst) { return false; }

// Slow polymorphic conversions
bool convert(virtual const RGB& src, virtual RGB& dst);
bool convert(virtual const RGB& src, virtual YUV& dst);
bool convert(virtual const YUV& src, virtual RGB& dst);
bool convert(virtual const YUV& src, virtual YUV& dst);

// Fast generic conversions, generated for each combination of types
bool convert(virtual const RGB32& src, virtual RGB32& dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual RGB24& dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual YUY2&  dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual YVU9&  dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual I420&  dst) { return generic_convert(src, dst); }
```

In case of double dispatch, the code becomes cluttered with definitions to support the mechanism:

```
// Forward declare all classes that would participate in double dispatch
struct RGB32;
struct RGB24;
// ... others

struct image
{
  // member—functions to access row buffer, width, height etc.

  // Double dispatch support code
  virtual bool convert_to(image& dst) const = 0;
  virtual bool convert_from(const RGB32& src) { return false; }
  virtual bool convert_from(const RGB24& src) { return false; }
  virtual bool convert_from(const RGB16& src) { return false; }
  // ... etc. for all other leaf image classes
};

struct RGB32 : RGB
{
  virtual bool convert_to(image& dst) const { return dst.convert_from(*this); }
  virtual bool convert_from(const RGB32& src) { return generic_convert(src, *this); }
  virtual bool convert_from(const RGB24& src) { return generic_convert(src, *this); }
  virtual bool convert_from(const RGB16& src) { return generic_convert(src, *this); }
  // ... etc. for all other leaf image classes
};
```

The major disadvantage of the double dispatch approach is that we have to foresee the whole hierarchy at the moment we are defining its root. This is necessary for declaring the interface for uncovering types. Once it is defined, we cannot extend it for newly created classes—they will all be treated as their closest ancestor in the hierarchy. Another problem with double dispatch is that its supportive structures clutter the code. This may be acceptable when double dispatch is needed for only one algorithm, but when several algorithms require it (e.g. we would also like to have a polymorphic **bool** compare(**virtual const** image& a, **virtual const** image& b)) then the code may quickly get out of hand. While this aspect

of the double dispatch can be solved with the visitor pattern at the cost of two extra virtual calls, the open-method solution will remain cleaner as open-methods do not even need to be defined together with the class. We discuss the visitor pattern in greater detail in our second example.

The number of lines in the implementation with open-methods was smaller, but all in all, the number of lines in both implementations is growing as the square of the number of classes in the hierarchy. We note that for open multi-method implementations this is a rather exceptional case, because the class hierarchy was shallow, while we were interested in uncovering all possible type combinations. For the double dispatch, this is rather typical case because the supportive definitions will have to be there anyway.

In the image conversion example the main purpose of the open-methods is to discover the dynamic types of both arguments and then forward the call to an overloaded function. This raises a question of whether the introduction of parameterized overriders would not make such definitions easier. In such a case, users can introduce only a base open method and a parameterized version of all the overriders. The compiler will then use the parameterized version to generate all the entries in the dispatch table:

```
// Base open—method. Fails as we do not know anything about the formats
bool convert(virtual const image& src, virtual image& dst) { return false; }

// Slow polymorphic conversions
bool convert(virtual const RGB& src, virtual RGB& dst);
bool convert(virtual const RGB& src, virtual YUV& dst);
bool convert(virtual const YUV& src, virtual RGB& dst);
bool convert(virtual const YUV& src, virtual YUV& dst);

// Fast generic conversions, generated for each combination of types
template <class Source, class Destination>
bool convert(virtual const Source& src,virtual Destination& dst)
{
  return generic_convert(src, dst);
}
```

This feature, however, can be a subject of a separate work, so we do not investigate it here.

9.2. AST traversal

The second example discusses the use of open-methods to traverse ASTs. The key focus thereby is on extending classes with open dispatch rather than multiple dispatch. Open-methods essentially become virtual functions that can be added to a class after it has been defined. The examples in this section reflect our experience of writing an analysis pass for the Pivot source-to-source transformation infrastructure [53]. The Pivot uses the visitor pattern to type-safely uncover the dynamic type of AST nodes. The Pivot consists of approximately 150 classes, but in the ensuing discussion, we limit the AST hierarchy to only two of them, where one, Expr, is a base class for all kinds of expressions, and the other, Unary, is an implementation of unary expressions.

```
struct Expr
{
  ...
  virtual accept(Visitor& v) const { v.visit(*this); }
};

struct Unary : Expr
{
  ...
  accept(Visitor& v) const { v.visit(*this); }
};

struct Visitor
{
  void visit(const Expr&  ) = 0;
  void visit(const Unary&) = 0;
};
```

Forwarding calls to base implementations: Currently, the Pivot has some 150 node types. The Pivot provides a number of intermediate abstract base classes that factor commonalities (e.g. Expr, Type, Declaration, etc.) of the 150 node types. If the logic of the visitor can be implemented in terms of a single base class, the bodies of the more specific types will need to explicitly invoke the base implementation (compare to the implementation for Unary). Open-methods have this forwarding behavior by default.

```
struct SimpleVisitor : Visitor
{
  virtual void visit (const Expr&  e) { /* ... */ };
  virtual void visit (const Unary& u) { visit(static_cast<Expr&>(u)); } // calls visit (Expr&)
};

// All objects derived from Expr will be handled by simpleOpenMethod
void simpleOpenMethod(virtual const Expr&) {/* ... */}

void foo(Expr& e, Unary& u)
{
  SimpleVisitor vis;

  e.accept(vis); // invokes SimpleVisitor :: visit (const Expr&)
  u.accept(vis); // invokes SimpleVisitor :: visit (const Unary&) first

  simpleOpenMethod(e); // invokes simpleOpenMethod(const Expr&)
  simpleOpenMethod(u); // invokes simpleOpenMethod(const Expr&)
}
```

Passing of arguments and results: The signatures of the visit and accept functions are determined when the visitor and the AST node classes are defined. Passing additional arguments to (or returning a value from) the visit functions requires intermediate storage as part of the visitor class. In the following example, inh and syn correspond to the input and result values of a function.

```
struct AnalysisPassVisitor : Visitor
{
  const InheritedAttr& inh; // data member for input parameter
  SynthesizedAttr*     syn; // data member for return value

  Visitor(const InheritedAttr & inherited) : inh(inherited), syn() {}
  ...
};
```

To avoid code duplication, it is useful to factor constructing the visitor and reading out the result into separate functions.

```
SynthesizedAttr* visit_foo (const Expr& e, const InheritedAttr & inh)
{
  AnalysisVisitor v(inh); // construct the visitor and pass the context
  e.accept(v);
  return v.syn;           // read and return the result
}
```

With open-methods, additional arguments can easily be specified as part of their signatures.

```
SynthesizedAttr* analysisPass(virtual const Expr& e, const InheritedAttr& inh);
```

Covariant return type: Since the result requires intermediate storage, covariant return types cannot easily be implemented with the visitor pattern. Consider the following implementation of a visitor that creates and returns a copy of an AST node.

```
struct CloneExpr : Visitor
{
  Expr* result; // data member for return value

  // make a copy of an Expr object
  virtual void visit (const Expr&  e) { result = new Expr(e); }
  // make a copy of an Expr object
  virtual void visit (const Unary& u) { result = new Unary(u); }
};

Expr* clone(const Expr& e) // analog of a base—method
{
  CloneExpr v;
  e.accept(v);
  return v.result;
}
```

Cloning a unary expression loses some type information, because the cloned objects would get returned as Expr. An implementation that is able to return covariant types requires a different visitor implementation (or instantiation) with

similar boilerplate code for each covariant return type. These repetitive definitions can be eliminated by using templates. The following example shows a cloning visitor that returns a Unary object.

```
struct CloneUnary : Visitor
{
  Unary* result; // data member for covariant return value
  virtual void visit (const Expr&  e) { assert(false); } // Can never be called!
  virtual void visit (const Unary& u) { result = new Unary(u); }
};

Unary* clone(const Unary& u) // analog of an overrider with covariant return type
{
  CloneUnary v;
  u.accept(v);
  return v.result;
}
```

The definition of open-methods with covariant return types is straightforward:

```
Expr& clone(virtual const Expr&  a); // base—method
Unary& clone(virtual const Unary& u); // overrider with covariant return type
```

Similar to open-methods, a hand-crafted technique for modeling covariant return type with visitors also creates additional "dispatch tables" for each overrider with covariant return type. Those are created in a form of v-tables for additional visitors. Interestingly enough, the overall size of such dispatch tables would be larger than those generated for open-methods, because visitors require v-table entries for visit methods that can never occur at runtime (see the assert in CloneUnary::visit(Expr&) above). This is not the case with open-methods as overriders with covariant return type will operate on smaller class hierarchies for their arguments.

Passing open-methods as callbacks: Open-methods nevertheless sometimes have disadvantages in comparison with the visitor pattern. Consider a traversal mechanism that traverses an AST in certain order. The mechanism can accept either a visitor or an open-method for node visitation:

```
void evaluation_order_traversal  (const Expr& e, Visitor& v);
void evaluation_order_traversal  (const Expr& e, void(*fn)(const Expr&));
```

In the case of a visitor, we can pass or accumulate some data during visitation. However, in the case of an open-method, we would need to accumulate data elsewhere.

```
struct CodeGenerationVisitor : Visitor
{
  std::vector<Instruction> instructions; // instruction stream inside visitor
  void visit(const Expr& e) { /* generate code for e*/ }
  // ...
};

CodeGenerationVisitor v;
evaluation_order_traversal(root_node(),v);
```

Although our current implementation does not support taking the address of an open-method, we can simulate that behavior by wrapping the open-method call inside a thunk.

```
std::vector<Instruction> instructions; // global instruction stream

void generate_code(virtual const Expr&  e) { instructions.push_back(...); }
void generate_code(virtual const Unary& e) { instructions.push_back(...); }
//...
void thunk_generate_code(const Expr& e) { generate_code(expr); }

evaluation_order_traversal(root(), &thunk_generate_code);
```

10. Conclusions and future work

We have 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 types 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 work-arounds. Indeed, it is only 16% slower than 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:

10.1. Virtual function templates

Virtual function templates are a powerful abstraction mechanism not part of C++ (see [51, §15.9.3]). Generating v-tables for virtual function templates requires a whole-program view and C++ traditionally relies almost exclusively on separate compilation of translation units. In [44], we demonstrate that the pre-linker described in this paper is able to synthesize dispatch tables for an approximation of templated open-methods. The concrete semantics of templated virtual functions (and open-methods) remains an open topic.

10.2. Function pointers to open-methods

Pointers to member functions in C++ preserve polymorphic behavior when they point to a virtual member function. To be in line with this semantics, pointers to open-methods should preserve dynamic dispatch too. This could be implemented by generating a thunk every time an address of an open-method is taken, and using the address of this thunk instead. Inside the function, the compiler simply generates a call to the appropriate open-method. Note that similar to single dispatch in C++, it will not be possible to take the address of a particular open-method overrider—the returned function will always dispatch dynamically.

10.3. 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.

We propose to be able to fix the dynamic type of an argument at the point of the open-method call to a particular unambiguous base class. This can either be done via `fix_type<Base>(arg)` or by introducing a special syntax, such as `arg as Base`. The concrete form is still under discussion. Under this approach users will be able to say `foo(d1 as B, d2)`, which means that the runtime type of `d1` is considered to be `B` rather than its actual runtime type. This effectively fixes the corresponding row or column in the dispatch table during the call. We note that the static type of `d1` has to be unambiguously derived from `B` in order for such type fix to be applicable.

This approach differs from the one used in MultiJava [21], where users have a choice between `resend` and `super` calls. `resend` invokes a less specific implementation that the current overrider refines, providing it can be uniquely determined. In cases when the overrider that invokes `resend` refines multiple overriders, a compile-time error is reported. A call to `super` dispatches to an implementation for a strict superclass of the receiver object.

Acknowledgements

We would like to thank to Nan Zhang for contributions to this research in its early stages and to Quadrox NV (Belgium) for providing the source code for experimenting with image conversion ideas. We are also grateful to Luke Wagner, Damian Dechev, Jaakko Järvi, Gabriel Dos Reis, and numerous anonymous referees for their helpful suggestions for improvement.

References

[1] R. Agrawal, L.G. Demichiel, B.G. Lindsay, Static type checking of multi-methods, in: OOPSLA'91: Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications, ACM, New York, NY, USA, 1991.
[2] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele Jr., S. Tobin-Hochstadt, The Fortress Language Specification, Tech. rep., version 1.0 (March 2008).
[4] E. Allen, J. Hallett, V. Luchangco, S. Ryu, J. Guy, L. Steele, Modular multiple dispatch with multiple inheritance, in: SAC'07: Proceedings of the 2007 ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2007.
[5] E. Amiel, O. Gruber, E. Simon, Optimizing multi-method dispatch using compressed dispatch tables, in: OOPSLA'94: Proceedings of the Ninth Annual Conf. on Object-oriented Programming Systems, Language, and Applications, ACM Press, New York, NY, USA, 1994.
[6] K. Arnold, J. Gosling, D. Holmes, The Java Programming Language, 4th ed., Prentice Hall, PTR, 2005.
[7] 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.
[8] P. Becker, The C++ Standard Library Extensions: A Tutorial and Reference, 1st ed., Addison-Wesley Professional, Boston, MA, USA, 2006.
[9] P. Becker, Working Draft, Standard for Programming Language C++, Tech. Rep. N2857, JTC1/SC22/WG21 C++ Standards Committee (March 2009).
[10] L. Bettini, S. Capecchi, B. Venneri, Double Dispatch in C++, Software - Practice and Experience 36 (6) (2006) 581–613.
[11] G.M. Birtwistle, O. Dahl, B. Myhrhaug, K. Nygaard, Simula BEGIN, Auerbach Press, Philadelphia, 1973.

[12] D. Bonniot, B. Keller, F. Barber, The Nice user's manual (2008). http://nice.sourceforge.net/manual.html.
[13] L. Bourdev, J. Järvi, Efficient run-time dispatching in generic programming with minimal code bloat, in: Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon, 2006.
[14] J. Boyland, G. Castagna, Parasitic methods: An implementation of multi-methods for Java, in: OOPSLA'97: Proceedings of the 12th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 1997.
[15] K. Bruce, L. Cardelli, G. Castagna, G.T. Leavens, B. Pierce, On binary methods, Theor. Pract. Object Syst. 1 (3) (1995) 221–242.
[16] C. Chambers, Object-oriented multi-methods in Cecil, in: ECOOP'92: Proceedings of the European Conf. on Object-oriented Programming, Springer-Verlag, London, UK, 1992.
[17] C. Chambers, The Cecil language: Specification and rationale. 3.2, Tech. rep. (2004).
[18] C. Chambers, The Diesel Language, specification and rationale (2006). http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-diesel-lang/diesel-spec.pdf.
[19] C. Chambers, W. Chen, Efficient multiple and predicated dispatching, in: OOPSLA'99: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 1999.
[20] C. Clifton, G.T. Leavens, C. Chambers, T. Millstein, MultiJava: Modular open classes and symmetric multiple dispatch for Java, in: OOPSLA'00: Proceedings of the 15th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2000.
[21] C. Clifton, T. Millstein, G.T. Leavens, C. Chambers, MultiJava: Design rationale, compiler implementation, and applications, ACM Trans. Program. Lang. Syst. 28 (3) (2006) 517–575.
[22] codesourcery.com, The Itanium C++ ABI, Tech. rep. (2001).
[23] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, USA, 2001.
[24] Edison Design Group, C++ Front End, http://www.edg.com/ (July 2008).
[25] C.B. Flynn, D. Wonnacott, Reconciling encapsulation and dynamic dispatch via accessory functions, Tech. Rep. 387 (1999).
[26] B. Foote, R. E. Johnson, J. Noble, Efficient multimethods in a single dispatch language, in: Proceedings of the European Conference on Object-Oriented Programming, Glasgow, Scotland, July.
[27] C. Frost, T. Millstein, Modularly typesafe interface dispatch in JPred, in: 2006 International Workshop on Foundations and Development of Object-oriented Languages, FOOL/WOOD'07, Charleston, SC, USA, 2006.
[28] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
[29] M. Gibbs, B. Stroustrup, Fast dynamic casting, Softw. Pract. Exper. 36 (2) (2006) 139–156.
[30] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
[31] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, Concepts: linguistic support for generic programming in C++, in: OOPSLA'06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006.
[32] G. Greif, Chinese Dispatch – Unpublished notes for a talk, Dylan Hackers Conference, Berlin (July 2002). http://www.opendylan.org/cgi-bin/viewcvs.cgi/trunk/www/papers/ChineseDispatch.lout?rev=8014&view=markup.
[33] International Standardization Organization, ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines, pub-ISO, pub-ISO:adr, 1994.
[34] ISO/IEC 14882 International Standard, Programming languages: C++, American National Standards Institute, 1998.
[35] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm specialization in generic programming: Challenges of constrained generics in C++, in: PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 2006.
[36] B. Liskov, Keynote address - data abstraction and hierarchy, in: OOPSLA'87: Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum), ACM Press, New York, NY, USA, 1987.
[37] L. Martin, Joint Strike Fighter, Air Vehicle, C++ Coding Standard, Lockheed Martin, 2005.
[38] B. Meyer, Eiffel: The Language, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
[39] T. Millstein, C. Chambers, Modular statically typed multimethods, Information and Computation 175 (1) (2002) 76–118.
[40] T. Millstein, M. Reay, C. Chambers, Relaxed MultiJava: Balancing extensibility and modular typechecking, in: OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conf. on Object-oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2003.
[41] T.D. Millstein, C. Chambers, Modular Statically Typed Multimethods, in: ECOOP'99: Proceedings of the 13th European Conf. on Object-Oriented Programming, in: LNCS, vol. 1628, Springer-Verlag, London, UK, 1999.
[42] R. Muschevici, A. Potanin, E. Tempero, J. Noble, Multiple dispatch in practice, in: OOPSLA'08: Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, 2008.
[43] E. Panizzi, B. Pastorelli, Multimethods and separate static typechecking in a language with C++-like object model, The Computing Research Repository (CoRR) cs.PL/0005033.
[44] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, Dynamic algorithm selection for runtime concepts, Science in Computer Programming, in press, (http://dx.doi.org/10.1016/j.scico.2009.04.002).
[45] G.v. Rossum, The Python Language Reference Manual, Network Theory Ltd., 2003, Paperback.
[46] M. Schordan, D. Quinlan, A source-to-source architecture for user-defined optimizations, in: JMLC'03: Joint Modular Languages Conference, in: LNCS, vol. 2789, Springer-Verlag, 2003.
[47] A. Shalit, The Dylan Reference Manual, 2nd edition, Apple Press, 1996.
[48] J. Smith, Draft proposal for adding Multimethods to C++., Tech. Rep. N1463 (2003).
[49] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, in: OOPSLA'86: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, ACM, New York, NY, USA, 1986.
[50] G.L. Steele Jr, Common LISP: the Language, 2nd ed., Digital Press, Newton, MA, USA, 1990.
[51] B. Stroustrup, The Design and Evolution of C++, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.
[52] B. Stroustrup, The C++ Programming Language, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[53] B. Stroustrup, G. Dos Reis, Supporting SELL for high-performance computing, in: 18th International Workshop on Languages and Compilers for Parallel Computing, in: LNCS, vol. 4339, Springer-Verlag, 2005.
[54] D. Thomas, A. Hunt, Programming Ruby: The Pragmatic Programmer's Guide, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[55] J. Visser, Visitor combination and traversal control, in: OOPSLA'01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2001.
[56] D. Wasserrab, T. Nipkow, G. Snelting, F. Tip, An operational semantics and type safety proof for multiple inheritance in C++, in: OOPSLA'06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006.
[57] D. Wonnacott, Using accessory functions to generalize dynamic dispatch in single-dispatch object-oriented languages, in: COOTS'01: 6th USENIX Conference on Object-Oriented Technologies and Systems, San Antonio, TX, USENIX, 2001.
[58] A. Wöß, M. Löberbauer, H. Mössenböck, LL(1) conflict resolution in a recursive descent compiler generator, in: JMLC'03: Joint Modular Languages Conference, in: LNCS, vol. 2789, Springer-Verlag, 2003.
[59] www.boost.org, The Boost C++ Libraries, retrieved on July 4th, 2008.
[60] www.fourcc.org, Video codec and pixel format definitions, retrieved on February 20th, 2007.