Double Dispatch in C++

Lorenzo Bettini*, Sara Capecchi and Betti Venneri

Dipartimento di Sistemi e Informatica, Università di Firenze, Viale Morgagni 65, 50134 Firenze, Italy
{bettini,capecchi,venneri}@dsi.unifi.it

SUMMARY

Double dispatch is the ability of selecting dynamically a method not only according to the run-time type of the receiver (single dispatch), but also to the run-time type of the argument. This mechanism unleashes the power of dynamic binding in object-oriented languages, so enhancing re-usability and separation of responsibilities. However, many mainstream languages, such as, e.g., C++ and Java, do not provide it, resorting to only single dispatch. In this paper we propose an extension of C++ (applicable also to other OO languages) that enables double dispatch as a language feature. This yields dynamic overloading and covariant specialization of methods. We define a translation from the new constructs to standard C++ and we present the preprocessor implementing this translation, called doublecpp. The translated code enjoys static type safety and implements the semantics of double dispatch by using only standard mechanisms of static overloading and dynamic binding, with minimal impact on the performance of the program.

KEY WORDS: Object-Oriented Programming, Double Dispatch, Multi Methods, Dynamic Overloading, C++, Language Extension.

INTRODUCTION

Polymorphism and dynamic binding are two fundamental mechanisms in object-oriented programming languages. The former, together with subtyping and substitutivity, permits treating different, but related, objects uniformly and the latter ensures that the right operation is performed when a message is sent to such objects, according to their actual type. Furthermore, the choice of combining dynamic binding with static (polymorphic) typing seems to be the suitable solution to achieve both flexibility and reliability in most popular languages, such as Java [1] and C++ [2]. However, when considering software engineering requirements of object technology, static typing raises some limitations in exploiting dynamic method selection. Let us consider, for example, the following general software scheme. It is a good design technique not to overwhelm a class with many operations: some operations

*Correspondence to: Lorenzo Bettini, Dipartimento di Sistemi e Informatica, Università di Firenze, Viale Morgagni 65, 50134 Firenze, Italy. bettini@dsi.unifi.it

Contract/grant sponsor: This work has been partially supported by EU within the FET - Global Computing initiative, project AGILE IST-2001-32747 and by the MIUR project EOS. The funding bodies are not responsible for any use that might be made of the results presented here.

Copyright © 2000 John Wiley & Sons, Ltd.
should be abstracted in another class, or, better, in a separate hierarchy. So we end up having a class hierarchy for a specific operation with a superclass, say, \texttt{Operation}, that is separate from the hierarchy of the elements with a superclass, say, \texttt{Elem}. This separation easily allows to change dynamically the kind of operation performed on elements by simply changing the object responsible for the operation (indeed, object composition is to be preferred to class inheritance, in situations where these changes have to take place dynamically, see [3]). For instance, we may structure the class \texttt{Operation} as follows,

\begin{verbatim}
class Operation {
    op(ElemA *e) {}
    op(ElemB *e) {}
    op(ElemC *e) {}
    ...}
\end{verbatim}

in order to perform a specific operation according to the type of element (suppose the \texttt{ElemX} all belong to the \texttt{Elem} hierarchy). However, such methods \texttt{op}, having different signatures, are considered overloaded, and thus, in languages such as C++ and Java, they are both type checked and selected statically: at compile time it is decided which version fits best the type declaration of the argument \texttt{e}. So, the following code:

\begin{verbatim}
Operation *o;
Elem *e;
...
o->op(e);
\end{verbatim}

would not dynamically select the most appropriate method according to the actual (run-time) type of \texttt{e}. In a sense, polymorphism is not exploited at the highest level (indeed [4] calls this kind of polymorphism “ad-hoc”).

This problem is due to the lack of \textit{double dispatch}, i.e., the ability of dynamically selecting a method not only according to the run-time type of the receiver (\textit{single dispatch}), but also to the run-time type of the argument (when all arguments are considered, we have \textit{multiple dispatch}). Though double dispatch is an old concept, widely studied in the literature, many mainstream languages do not provide this powerful mechanism; due to this lack, programmers are forced to resort to RTTI (run time type information) mechanisms and if statements to explore manually the run-time type of an object, and to type downcasts, in order to force the view of an object according to its run-time representation. Indeed, in many cases, this is a necessary solution to overcome lacks of the language [5]. These techniques are discouraged by object-oriented design, since they undermine re-usability and evade the constraints of static type checking. Cleaner solutions, such as the \textit{Visitor} pattern [3], still require the programmer attention and efforts. Instead, having double dispatch as a linguistic feature, allows the compiler to analyze the code, check type correctness, and notify the programmer of potential errors.

Furthermore, double dispatch enables safe \textit{covariant specialization} of methods, where subclasses are allowed to redefine a method by specializing its arguments. There is a general evidence that covariant code specialization is an indispensable practice in many situations [6, 7]. Its most expressive application appears with \textit{binary methods} [8], i.e., methods that act on objects of the same type: the receiver and the argument (see the ‘Examples’ section for an implementation of binary methods). It seems quite natural to specialize binary methods in subclasses and to require that the new code
DOUBLE DISPATCH IN C++

replaces the old definition when performing code selection at run-time according to the actual type of the argument and the receiver.

Despite its practical usefulness, covariant method specialization with late selection strategy is often disregarded as an unsafe mechanism w.r.t. the static type checker; namely, only the opposite rule, that is contravariance on the argument type, correctly fits the subtyping relation on method types and then substitutivity. G. Castagna clarifies this controversy in [9], by showing that method overriding (contravariance) and code specialization (covariance) are distinct concepts with different roles, so that both can be integrated in a type safe manner in object oriented languages (we refer the reader to [9] for a comprehensive discussion on this subject).

In this paper we propose a linguistic extension of C++ [2] with multi methods [10, 11, 12, 13] supporting the mechanisms of double dispatch and covariant specialization. A multi method can be seen as a collection of overloaded methods associated to the same message, but the selection takes place dynamically according to multiple dispatch. In our approach, we limit multi methods to double dispatch (section ‘Towards Multiple Dispatch’ motivates this restriction and hints at how to extend the approach to multiple dispatch). Programmers state their intentions through a distinct syntax of method invocation, choosing between C++ standard method selection mechanism and double dispatch. Thus, our extension is conservative, since it does not affect the kernel typing and semantics of C++ programs that do not use the new constructs. We provide the translation scheme for representing the proposed extension in standard C++, exploiting only static overloading and dynamic binding. This translation is thought to be automatically executed by a program translator (a preprocessor) that has to be run before the C++ compiler. A prototype implementation of such a preprocessor, doublecpp, (freely available at http://www.lorenzobettini.it/software/doublecpp) has been implemented and experimented in many case studies. Given a C++ program using the new linguistic constructs, doublecpp produces standard C++ code.

Four distinctive features characterize our proposal, among many related solutions to the problem at issue:

- We provide a full form of multi methods that are not encapsulated (see, e.g., [8, 14]) at the cost of sacrificing modularity. With encapsulated multi methods the first method selection is performed according to the run-time class of the receiver and then only the subselection is performed according to the run-time class of the argument. Instead, in our approach, the receiver and the argument participate equally to the method selection by double dispatch without any priority. On the other hand, encapsulated multi methods have the main advantage of modularity allowing a smoother separate compilation of classes, which full multi methods prevent.
- Our extension of C++ is conservative since it preserves static (syntactic) overloading, while adding the additional form of dynamic overloading. In particular, the new mechanism is semantically consistent with static overloading and standard dynamic binding in C++; this point becomes clear when considering the interaction between such dynamic overloading and C++ access control (see section ‘Private, Protected and Virtual Methods’).
- Our solution is characterized by the crucial issue of the type safety property: the translated code is type safe in the sense that upon dynamic overloading method invocation no run-time error will be raised due to a missing branch or to ambiguities (see section ‘Typing Issues’).
- Concerning a practical evaluation, our translation does not introduce crucial overhead during method selection: it takes place in constant time, since it basically uses dynamic binding twice,
and overloading is resolved statically by the compiler once and for all. For highlighting this point, we present some benchmarks (see section ‘Evaluation’) that compare the performance of our approach with respect to a translation scheme based on RTTI and type casts.

A typical scenario where dynamic overloading is very useful is when there are two separate class hierarchies and the classes of a hierarchy have to operate on instances of classes of the second hierarchy according to their dynamic types. Quoting from [15], “Detecting the need for multi methods is simple. You have an operation that manipulates multiple polymorphic objects through pointers or references to their base classes. You would like the behavior of that operation to vary with the dynamic type of more than one of those objects.” Indeed, in [16], Stroustrup admits that he had considered adding multi-methods to C++ but he then abandoned this idea with regret because he “couldn’t find an acceptable form under which to accept it”. The two goals he had in mind were to find:

1. “A calling mechanism that was as simple and efficient as the table lookup used for virtual functions;
2. A set of rules that allowed ambiguity resolution to be exclusively a compile-time matter”.

Our approach enjoys these two requirements, that can be summarized as efficiency and type safety. These properties are not satisfied altogether by other proposals found in the literature. Moreover, the main context where [16] analyzes the impact of multi methods is the binary method one: “Such a language facility would be a boon to writers of code that deals with binary operations on diverse objects”. In particular, [16] proposes a technique similar to the one employed here as a “manual” workaround for multi methods.

THE EXTENDED C++

For our purpose we restrict our attention to kernel features of C++:

- class definitions (sequences of field and method declarations) and inheritance to define subclasses;
- object creation by new operator, so that objects are referred to by pointers;
- class names are types and subclassing implies subtyping (here denoted by \( \leq \)).

For simplicity, in the first part of the paper, we will concentrate on public methods only, and we assume all methods as virtual. We consider the general case, when such restrictions are removed, in section ‘Private, Protected and Virtual Methods’.

Then, C++ syntax is extended by adding a new construct to define multi methods with their branches. For simplicity, since we focus on double dispatch only, we restrict multi method signatures to one parameter (this restriction is removed in the actual implementation).

The syntax of a multi method declaration is as follows:

\[
\text{multi-method} ::= \text{branches name branches endbranches} \\
\text{branches} ::= \text{branch} \\
\text{branch} ::= \text{type ( type }\ast\text{ arg ) ;} \\
\text{branch} ::= \text{type ( type }\ast\text{ arg )} \\
\text{branch} ::= \text{type ( type }\ast\text{ arg )} \{ \text{body} \} ;
\]
Intuitively, all branches of a multi method $m$ constitute different behaviors of $m$ on different arguments, where each branch’s body is a standard C++ method definition. A subclass can extend its superclasses multi methods by providing additional branches or redefining some of them.

Moreover, there are two forms of expressions for multi method invocation. If $m$ is declared as a multi method, then $\text{exp} \rightarrow m(t)$ denotes the standard method invocation while $\text{exp} \rightarrow m_\text{DB}(t)$ is a new construct using _DB as a suffix for $m$. These two kinds of expressions correspond to different semantics, i.e., static and dynamic overloading, respectively, as explained in the following section.

Since we consider dynamic overloading a dynamic enhancement of static overloading, just like dynamic binding for virtual methods is an enhancement of method invocation, and since it is possible to avoid dynamic binding by using the :: scope resolution operator, we want to allow the programmer to decide between dynamic and static overloading with the two method invocations shown above. The same design choice is present also in [17]. Moreover, this allows a redefined branch to call the version of the branch in a superclass (as shown later in the example of binary methods).

**Method selection**

Let us illustrate the basic idea by the example in Listing 1 where two multi methods, $m$ and $n$, are defined. In each class, all the branches of a multi method represent a standard definition of a C++ overloaded method; as a consequence, for instance, the return type is not used in method selection. However, differently from C++, multi methods with all their branches are implicitly inherited in derived classes (except for private ones, as explained in section ‘Private, Protected and Virtual Methods’), thus obtaining a full form of *copy semantics of inheritance* [18]. Message passing is the key point, from the semantic point of view. Besides including the two standard mechanisms of method invocation, i.e., static overloading and dynamic binding (single dispatch), multi methods allow to model the additional mechanism of double dispatch: if $m$ is declared as a multi method, then the invocation of $m$ with the suffix _DB ($m_\text{DB}$) is interpreted by using *dynamic overloading* †. Namely, when $m_\text{DB}$ is invoked, the method executed is always chosen dynamically among all the available branches of $m$. In particular, this dynamic selection will choose the *most specialized* branch according both to the dynamic type of the receiver (as in standard single dispatch) and to the dynamic type of the actual parameter (i.e., double dispatch). The property of static well-typedness guarantees that this selection can always be made without ambiguities.

For instance, if we have the following code (assuming that $T_j \leq T_i$ if $i \leq j$):

```cpp
A *a = new A;
T1 *t = new T2;
a->m(t); // static overloading
a->m_DB(t); // dynamic overloading
```

then, the first method invocation is performed according to the static overloading semantics, i.e., $A::m(T1 *)$ is selected (statically), while the second method invocation selects (dynamically) $A::m(T2 *)$.

†Instead of introducing a new keyword or a different notation for method invocation with double dispatch, we have chosen to use the method name plus the suffix _DB in order to simplify the translation.
The new construct enables covariant specialization of the parameter types in the branches of a multi method. Indeed, B redeﬁnes the branch $m(T2 \ast \star)$ but also specializes the multi method by adding a new branch, $m(T3 \ast \star)$, where $T3 \leq T2$. Again, the most specialized branch of a multi method will be selected dynamically for invocation on objects belonging to subclasses. Thus, considering the following code:

```cpp
A *a = new B;
T1 *t = new T1;
a-->m_DB(t); // dynamic overloading
t = new T2;
a-->m_DB(t); // dynamic overloading
t = new T3;
a-->m_DB(t); // dynamic overloading
```

the ﬁrst invocation will select $A::m(T1 \ast \star)$, the second one will select $B::m(T2 \ast \star)$, because $B$ has redeﬁned the branch $m(T2 \ast \star)$ and the third one will select $B::m(T3 \ast \star)$, because $B$ has deﬁned a specialized branch for $m(T3 \ast \star)$.

**Non-encapsulated vs. Encapsulated**

Let us remark that the receiver and the parameter participate together in the dynamic selection of the method as in [12, 19]. Differently, encapsulated multi methods [8, 14] are characterized by the fact that the receiver has the precedence over the parameters, during dynamic method selection. With this respect, non-encapsulated multi methods provide a full form of copy semantics of inheritance [18], which, on the contrary, encapsulated multi methods prevent.

These concepts deal with visibility and scoping of the branches of a multi method in the class hierarchy: the main drawback of encapsulated multi methods is that the (re)deﬁnition of a multi method in a subclass completely overrides the old one (see [8]). This means that the branches deﬁned in the

---

**Listing 1:** A ﬁrst example in C++ with double dispatch

```cpp
class A {
    // fields...
    // methods...
    branches m
    T (T1+t) { ... };
    T (T2+t) { ... };
    ...
    endbranches

    branches n
    S (S1+t) { ... };
    S (S2+t) { ... };
    ...
    endbranches
};

class B : public A {
    // fields...
    // methods...
    branches m
    T (T2+t) { ... };
    T (T3+t) { ... };

    branches n
    S (S3+t) { ... };
    ...
    endbranches
};
```
DOUBLE DISPATCH IN C++

superclass are not automatically inherited in a subclass that specializes (i.e., adds a branch) a multi method or redefines a branch. For instance, in Listing 1, class $B$ would not inherit the two branches of $n$ defined in $A$, with encapsulated multi methods. This would make use of multi methods quite impractical in situations where the derived classes have to modify or add a branch of a multi method of the superclass (see the implementation of the visitor classes in section ‘Extended C++ at work’).

Let us observe that C++ scoping rules are still valid, so the programmer can rule the dynamic selection of a specific branch by using the :: scope resolution operator. For instance, if in the code above we replace $a->m_DB(t)$ with $a->A::m_DB(t)$ we basically restrict the dynamic branch selection to the scope of class $A$. Thus, the specialized branch $B::m(T3 \ast)$ will not be considered during branch selection. However, since dynamic binding is still employed for selecting the implementation of a specific branch, $B::m(T2 \ast)$ will be selected (summarizing the three method invocations above will select $A::m(T1 \ast)$, $B::m(T2 \ast)$ and $B::m(T2 \ast)$ again, respectively). This holds because we implicitly consider each method as virtual. In the following we will show how the programmer can effectively specify whether branches are virtual or not.

**Two alternatives in method selection**

In our approach the decision whether to use dynamic or static overloading for a multi method is taken at method invocation time, by choosing between two syntactic forms. This means that the caller of the method must be aware of the fact that he is about to use a multi method. With this respect, our choice is in contrast with C++ (and Java): when you call a method on a pointer or reference, unless you use the explicit scope resolution operator, you do not need know whether the method is virtual or not. Thus, in C++ it is the programmer of the class that somehow decides the invocation policy, and the code that uses a method must not be changed if the method is turned into virtual, or it is made non virtual anymore.

Since dynamic overloading is a new concept in C++ we preferred to adopt a more conservative strategy requiring to use the _DB suffix upon method invocation in order to achieve the dynamic overloading semantics. In some sense, we think that this is a safer choice, not changing the behavior of existing pieces of C++ code in a program. However, if someone is not comfortable with this decision and wants the existing code calling a method $m$ to transparently switch to dynamic overloading when $m$ is turned into a multi method, the implementation doublecpp allows them to do so: when using --rename-overloaded command line option, the branches of a multi method are given a different name (e.g., $m_\ast$) and the original method name is used instead of the _DB suffixed name. For instance, the code snippet at the beginning of the section, using the multi method in Listing 1, is rewritten as follows

```c++
A *a = new A;
T1 *t = new T2;
a->m(t); // dynamic overloading
a->m_(t); // static overloading
```

This is consistent with the C++ choice, where the programmer of the class decides the default invocation strategy and code using $m$ will transparently adapt to a transformation of $m$ from a standard method into a multi method (or the other way round). In any case, we still allow the caller of the
class Visitor {
  branches visit
  void (ElemA *t);
  void (ElemB *t);
  endbranches
};

class ExtVisitor : public Visitor {
  branches visit
  void (ElemA *t);
  void (ElemC *t);
  endbranches
};

Listing 2: An implementation of the visitor in C++ with double dispatch

method to go round the default behavior. Notice that the translation we present in this paper is basically independent from the specific method renaming.

Extended C++ at work

In scenarios with separate hierarchies that have to interact, in order to avoid the awful use of RTTI and type casts, the design pattern Visitor [3] is widely used. The idea behind this pattern is that the base class Visitor defines all the methods that have to operate on the elements of the second hierarchy, with base Elem.

With our proposed language extension, the visitor classes are easily programmed by defining a multi method visit with a branch for each Elem that has to be handled, as illustrated in Listing 2. The Elem hierarchy does not have to be modified by the programmer: all the internal issues will be handled by the preprocessor of the construct for multi methods. Furthermore, a subclass of Visitor can add a branch for a new Elem subclass (as ExtVisitor for ElemC) without affecting the base class (covariant specialization); in the Visitor pattern this cannot be done smoothly, without resorting to casts.

The programmer can visit an element by simply calling visit_DB on any Elem instance:

Elem *elem = new ElemB;
Visitor *visitor = new ExtVisitor;
visitor->visit_DB(elem);

We would like to stress that our proposal is not an automatic implementation of the visitor pattern but the implementation of a more general concept, i.e., double dispatch. The visitor pattern is a programming discipline “to iterate through a collection of polymorphic objects” (typically together with the iterator pattern) [3]. This is the case, for instance, of a compiler: the visitor classes are all the classes performing several controls on the abstract syntax tree, and the nodes of the tree are the elements that are to be visited. Then, if the language does not provide double dispatch the programmer must write the accept methods in the classes of the objects that are to be visited, in order to achieve the same functionality of dynamic overloading. With our linguistic extension, the programmer can simply write the visitor methods without having to write all the accept methods. Let us observe that encapsulated multi methods would make the implementation of the visitor classes much harder: since with encapsulated multi methods a derived class that redefines or specializes a branch of a multi method hides the branches of the same multi method in the superclass (i.e., it does not inherit the branches from the superclass), the programmer of ExtVisitor would be required to redefine also the branch for ElemB. This is not necessary with our non-encapsulated multi methods.
Dynamic overloading and covariant specialization come in hand also in implementing many other design patterns that are based on the collaboration of separate class hierarchies, for instance, the pattern *Observer* and the pattern *Strategy* (we refer the reader to [3] for all the details of these patterns).

A further useful application of double dispatch, related to covariance of the argument type, is the implementation of binary methods, a well known problem in the literature, as suggested in [8]. For instance, exploiting the usual example of *Point* and *ColorPoint*, we can implement the method *equals* as illustrated in Listing 3. Equality of points can be tested smoothly as follows:

```cpp
Point *p1 = new Point;
Point *p2 = new ColorPoint;
Point *p3 = new ColorPoint;
p1->equals_DB(p2); // (1) invoke Point::equals(Point *)
p2->equals_DB(p1); // (2) invoke Point::equals(Point *)
p2->equals_DB(p3); // (3) invoke ColorPoint::equals(ColorPoint *)
```

Notice that all the branches of the same multi method are implicitly inherited by the subclass, so *ColorPoint* is able to handle also *Point* instances passed to *equals*: in this case the implementation *Point::equals(Point *)* will be called. However, the programmer of *ColorPoint* may want to consider a *ColorPoint* and a *Point* different (since the latter has no color); in that case he can simply redefine that branch in *ColorPoint* as follows

```cpp
branches equals
bool (Point *) { return false; }
...
endbranches
```

Of course, in this case, the second invocation in the previous code snippet would invoke *ColorPoint::equals(ColorPoint *)*.

Let us observe that it is crucial to have the possibility of using both dynamic overloading and static overloading: see, for example, *ColorPoint::equals(ColorPoint *)* that relies on the implementation of *Point::equals* by static overloading. This led to the main choice of incorporating both kinds of overloading in our extension, allowing the programmer to use different method invocations in different contexts.
TYPING ISSUES

In this section we discuss how the type system of the basic C++ language can be extended with the new constructs, i.e., to multi method definitions and invocations. Our typing extension is widely inspired by [12, 14].

Firstly, the syntax of types is extended with multi types that are sets of arrow types associated to branches of multi methods. A multi type \( \Sigma \) is of the form
\[
\Sigma = \{ I_1 \rightarrow T_1, \ldots, I_n \rightarrow T_n \}
\]
where each input type \( I_i \) is a pair of C++ types, \((A \times B)\); namely, when \((A \times B)\) is the input type of a multi method branch, then \(A\) is the receiver type and \(B\) is the argument type.

Well formedness of multi types. Multi types are constrained by three crucial consistency conditions that are checked statically. A multi type \( \Sigma = \{ I_1 \rightarrow T_1, \ldots, I_n \rightarrow T_n \} \) is well-formed if and only if for any \( I_i, I_j \) (\( i \neq j \)):

(i) \( I_i \neq I_j \);
(ii) if \( I_i \leq I_j \) then \( T_i \equiv T_j \);
(iii) if \( I_i \) and \( I_j \) have a common subtype, then for every maximal type \( I_k \) of the set of their common subtypes there must be one arrow type \( I_k \rightarrow T_k \) in \( \Sigma \).

Condition (i) is quite standard on overloaded definitions also in C++. In condition (ii) we require \( T_i \equiv T_j \) (instead of \( T_i \leq T_j \) as in [12, 14]) according to the philosophy of C++ where return types are not used in overloaded method selection. Condition (iii) implies that for any type \( I \), such that \( I \leq I_1 \) and \( I \leq I_j \), then there must be one input type in \( \Sigma \), say \( I_k \), such that \( I \leq I_k, I_k \leq I_1 \) and \( I_k \leq I_j \). This last condition will play a crucial role in catching at compile time any possible static and dynamic ambiguity in multi method calls. Its meaning will become clear when discussing well-typedness at the end of this section.

Subtyping. Subtyping extends to multi types in a quite natural way. Let us assume the standard subtyping relation on arrow and product types, i.e.
\[
A \leq A_1, B \leq B_1 \Rightarrow (A \times B) \leq (A_1 \times B_1)
\]
and
\[
A \leq A_1, B \leq B_1 \Rightarrow A_1 \rightarrow B \leq A \rightarrow B_1
\]
Then, \( \Sigma_1 \leq \Sigma_2 \) if and only if for every arrow type in \( \Sigma_2 \) there is at least one smaller (or equal) arrow type in \( \Sigma_1 \).

Typing multi methods and multi method invocations.

We recall that a multi method defined in a superclass can be extended in a derived class by inheriting its definition and possibly adding new branches. Moreover the behavior of some of its branches can be overridden in the subclass while preserving the signature, i.e., the argument type and the return type.
Typing rule for multi method declaration. A multi method $m$ in the class $C$ has the multi type $\Sigma$  

$$\Sigma = \{(C \times A_1) \rightarrow T_1, \ldots, (C \times A_n) \rightarrow T_n\} \cup \Sigma' \cup \Sigma'_C$$

iff

- $\{A_1 \rightarrow T_1, \ldots, A_n \rightarrow T_n\}$ is the set of the types of all the branches of $m$ defined in $C$;
- $\Sigma'$ is the (possibly empty) multi type of $m$ in the superclass of $C$;
- $\Sigma'_C$ is obtained from $\Sigma'$ by substituting the receiver type with $C$ in the input types (formally $\Sigma'_C = \{(C \times A) \rightarrow T \mid (B \times A) \rightarrow T \in \Sigma'\}$);
- $\Sigma$ is well-formed.

Typing rule for multi method invocation. The invocation of $m$ (both of $m$ and of $m_{\text{DB}}$) on a receiving object of type $A$ with an argument of type $B$ has type $T$ if and only if there is an arrow type $(A' \times B') \rightarrow T \in \Sigma$ such that $(A \times B) \leq (A' \times B')$.

Properties of typing. The above conditions of well-formedness play a crucial role in multi method invocation. Let us discuss this in further details (we refer to [12, 14] for the formal treatment of this issue). If a multi method invocation is well typed, with type $\Sigma$, then for any possible input type $I = (A \times B)$ the set $S_{(A \times B)}$ of invocable branches input types

$$S_{(A \times B)} = \{(A_i \times B_i) \mid (A \times B) \leq (A_i \times B_i)\}$$

selected from $\Sigma$, is not empty. This means that there will always exist a possible choice of a branch to call for all actual choices of static types $(A \times B)$ and dynamic subtypes of $(A \times B)$. This ensures that in a well typed program no message-not-understood error will take place.

Moreover, since $\Sigma$ is well-formed, the above set $S_{(A \times B)}$ contains a minimal input type, say $I_k = (A_k \times B_k)$, i.e., $(A_k \times B_k) \leq (A_i \times B_i)$ for all $(A_i \times B_i) \in S_{(A \times B)}$, by condition (iii) of well-formedness. The branch of type $I_k \rightarrow T_k$ is the only one that “best approximates” the input type $I = (A \times B)$. This means that there is one and only one most specific applicable branch for any possible choice of $I$, i.e., in a well typed program no message-ambiguous error will take place during the execution. Furthermore, by condition (ii) of well-formedness, $T_k = T = T_i$, for each $I_i \rightarrow T_i$ where $I_i \in S_j$; this ensures that the return type $T$ is univocally determined in the above rule for typing multi method selection.

Let us stress that the above argument motivating the absence of message-ambiguous holds not only for any static input type $I$, but also for any dynamic input subtype $I^* \leq I$. In this case the set of input types of callable branches, $S_{I^*}$, could become larger ($S_I \subseteq S_{I^*}$), and so the best approximating branch to be selected dynamically (minimal input type) could have a more specialized input type, but still the same return type by condition (ii). The existence and uniqueness of such branch are still ensured by condition (iii) of well-formedness, as explained above.

Summarizing, the static well typedness property guarantees that, for each possible choice of dynamic input type,

1. the static return type is unique and preserved during evaluation,
2. there will always be one and only one most specialized branch to be selected.
This avoids message-not-understood and message-ambiguous run-time errors during the evaluation of a program that has been statically type-checked successfully. As a consequence, the new rules preserve type safety of the basic C++ type system. Concerning this point, we observe that other proposals suffering from message-not-understood and run-time ambiguities in method selection do not employ such a strong static type discipline.

Finally, let us remark that the conditions of well-formedness rely on the subtyping relation, which is independent from the number of arguments. Indeed, there is no conceptual problem in extending our solution to multiple dispatch as discussed in section ‘Towards Multiple Dispatch’. From the typing point of view, all we have to do is to consider tuple types, instead of pair types, for input types and the corresponding subtype relation (which is the natural extension of subtyping from pair types to tuple types). Thus, all the discussions above naturally apply to multiple arguments, still ensuring the absence of message-not-understood and message-ambiguous in well-typed programs. Indeed the formal treatment in [12, 14] already considers multiple dispatch.

TRANSLATING DOUBLE DISPATCH INTO C++

In this section we present the translation of the new constructs into standard C++ code that uses only static overloading and dynamic binding (i.e., single dispatch). We adopt the following terminology: we refer to classes defining and redefining branches of a multi method as the clients (e.g., visitor classes) of the target classes (e.g., element classes), i.e., those used for declaring the type of the parameter of a branch.

We would like to observe that the translation is defined on well-typed programs, so we assume properties concerning well-typed programs in defining the algorithm, in particular, we know that all multi methods are well formed.

The key idea

The translation consists in modifying both the client and the target classes by appropriately adding methods that implement the double dispatch semantics. In particular, for any multi method \( m \) some additional \( m_{\_DB} \) methods are introduced associated to the invocation of \( m \) by double dispatch. Moreover, \( m_{\_DB} \)’s body uses methods \( \text{disp}_m \) introduced in the target classes. In a sense, our translated code is inspired by the visitor design pattern: the double dispatch is indeed implemented by dispatching the method invocation twice. Firstly the invocation is dispatched to the target element (the one to be visited in the visitor pattern) that dispatches it back to the original receiver of the method. This back-and-forth dispatching, thanks to dynamic binding and static overloading, allows to correctly and dynamically select the most specialized method.

We give an informal idea of the translation, by considering the example of Listing 1. The generated standard C++ code is illustrated in Listing 4. Now let us consider the following code:

```c++
A *a = new A;
T1 *t = new T2;
a->m_DB(t);
```

and interpret its execution:
class A {
    T m_DB(T1 *t)
    { return t->disp_m(this); }
    T m(T1 *t) { ... }
    T m(T2 *t) { ... }
    Sn_DB(S1 *t)
    { return t->disp_n(this); }
    S n(S1 *t) { ... }
    S n(S2 *t) { ... }
};

class B : public A {
    T m_DB(T1 *t)
    { return t->disp_m(this); }
    using A::m;
    T m(T2 *t) { ... }
    T m(T3 *t) { ... }
    S n_DB(S1 *t)
    { return t->disp_n(this); }
    using A::n;
    S n(S3 *t) { ... }
};

class T1 {
    T disp_m(A *a)
    { return a->m(this); }
    T disp_m(B *b)
    { return b->m(this); }
};

class T2 : public T1 {
    T disp_m(A *a)
    { return a->m(this); }
    T disp_m(B *b)
    { return b->m(this); }
};

class T3 : public T2 {
    T disp_m(B *b)
    { return b->m(this); }
};

class S1 {
    T disp_n(A *a)
    { return a->n(this); }
    T disp_n(B *b)
    { return b->n(this); }
};

class S2 : public S1 {
    T disp_n(A *a)
    { return a->n(this); }
    T disp_n(B *b)
    { return b->n(this); }
};

class S3 : public S2 {
    T disp_n(B *b)
    { return b->n(this); }
};

Listing 4: The standard C++ code generated from the code of Listing 1. The other parts and methods of the classes are not shown here. All methods are virtual.

1. A::m_DB(T1 *t) will execute t->disp_m(this); in the method, this is of (static) type A*, thus among the disp_m of class T1, method disp_m(A *) has already been (statically) selected. The (pointer to) object t is statically of type T1, but dynamically is of type T2; since dynamic binding is used in method selection, then the method T2::disp_m(A *) will be dynamically executed.

2. Inside T2::disp_m(A *) the method invocation a->m(this) is performed; in this method, this is of (static) type T2*, and a is of (static) type A*, thus, according to static overloading, the method A::m(T2 *) will be executed.

Thus, the generated C++ code actually implements dynamic overloading semantics, since the most specialized version of an overloaded method is dynamically selected according to the dynamic type of its argument.
Let us now consider the following code:

```java
A *a = new B;
T1 *t = new T3;
a->m_DB(0);
```

its execution proceeds as follows:

1. by dynamic binding, the method `B::m_DB(T1 *)` is executed;
2. inside this method `this` is of (static) type `B*`, thus, when invoking `t->disp_m(this)`, the compiler has already statically selected the branch `disp_m(B *)` of the overloaded method `disp_m`; since `t` is dynamically of type `T3*`, due to dynamic binding, the method `T3::disp_m(B *)` is dynamically selected;
3. inside this method, `this` is of (static) type `T3*` and `b` is of (static) type `B*`, thus, `b->m(this)` statically selects `B::m(T3 *)`.

Once again, the most specialized version of the method has been dynamically selected, not only according to the run-time type of the receiver but also to the run-time type of the argument.

The key idea is that the dynamic overloading semantics can be obtained by exploiting dynamic binding (i.e., single dispatch) and static overloading twice: this way the right method is selected dynamically by exploiting the run time types of both the receiver of the message and the argument.

**The translation algorithm**

For simplicity, we do not present the full translation algorithm in a formal way, we just define its main steps concerning the new constructs.

(i) In each class definition `C`, for each multi method `m`, additional methods `m_DB` are inserted: instead of generating one `m_DB` for each branch of `m` we reduce their number according to the following procedure. Let `ArgTypes(m, C)` be the set of argument types in all the branches of `m` (both defined explicitly in `C` and inherited from the superclasses). The subset `DBTypes(m, C)` is so defined:

\[
DBTypes(m, C) \triangleq \{ T_i \in ArgTypes(m, C) | \exists T_j \in ArgTypes(m, C). T_i \leq T_j \} \cup \{ T_i \in ArgTypes(m, C) | \exists T_j, T_k \in ArgTypes(m, C) . T_i \leq T_j \land T_i \leq T_k \land T_j \land i \neq j \neq k \land unrelated(T_j, T_k) \}
\]

where `unrelated(T_j, T_k) \triangleq (T_j \nleq T_k) \land (T_k \nleq T_j)`. Then, for any `T_i` belonging to `DBTypes(m, C)` the following method is introduced in the transformed `C`, where `T` is the return type:

```java
T m_DB(Ti *x) { return x->disp_m(this); }
```

Notice that we use the parameter types of `m` for deciding the number of the `m_DB` to be generated; however, as a consequence of condition (ii) of well-formedness (see section ‘Typing Issues’) the return type of each `m_DB` is determined without ambiguities.
class Point {
    bool equals_DB (Point *p)
    { return p->disp.equals(this); }
    bool equals(Point *p) {...}
    bool disp.equals(Point *p)
    { return p->equals(this); }
    bool disp.equals(Point *p)
    { return p->equals(this); }
    bool disp.equals(ColorPoint *p)
    { return p->equals(this); }
    bool equals_DB (Point *p)
    { return p->disp.equals(this); }
    bool equals(ColorPoint *p)
    { return p->equals(this); }
};

class ColorPoint :
    public Point {
        bool equals_DB (Point *p)
        { p->disp.equals(this); }
        using Point::equals;
    }
    bool equals(ColorPoint *p)
    { return p->equals(this); }
};

Listing 5: The translation of the binary method equals of Listing 3. All methods are virtual.

(ii) For each target class $T_i$ such that $T_i$ is an argument type of a branch of a multi method $m$ in the class $C$, say $T \in \pi(T_i \times x)$, then the following method is added to the class $T_i$:

$$T \text{ disp}_m(C \times x) \{ \text{return } x \rightarrow m(\text{this}); \}$$

(iii) If $T_j$ is a superclass of $T_i$ and also a target of the same multi method $m$ in $C$ or in any superclass of $C$, the same method $\text{disp}_m$ is added to $T_j$ too.

Summarizing, $m_{DB}$ and $\text{disp}_m$ complement and complete each other to implement double dispatch: the method $m_{DB}$ aims at using statically the type of $C_i$ and dynamically the type of the $T_j$, while the method $\text{disp}_m$ has exactly the opposite goal.

We point out that binary methods can be seen as a degenerate case of the more general case presented above: client and target classes belong to the same hierarchy. For instance, the code shown in Listing 3 will be translated as shown in Listing 5. You may have noticed that in the translated Point class there is dependence on a subclass ColorPoint, which is an odd thing in object-oriented design. However, this is only in the generated code, i.e., invisible to the programmer as it is an implementation issue.

Concerning cyclic dependences, we remark that the translation is made in such a way that the target classes depend on the client classes minimally, as in Listing 6: in the target’s header file, client classes are declared through forward declarations and the actual generated body of $\text{disp}_m$ methods are inserted in an additional source file. This way, the target class header file of $T_1$ is changed by the preprocessor either when a client uses that class as a target in a branch, or when a client specializes an existing multi method, using $T_1$ as a target, with a subclass of $T_1$. Also other existing sources using the header file of $T_1$ have to be recompiled only in such a situation. With this respect, the header file of $T_1$ does not actually depend on the header file of the client classes, thus there is no cyclic dependence.

We proved that any C++ program obtained by translating a well typed program written in the extended language is still well typed (type correctness of the translation). A formal proof of this property requires a complete formalization of the language (both kernel C++ and its extension) and its type system, which is out of the scope of the present paper. However, informally, it is easy to check the property at issue. Firstly, we verify that all the new methods generated by the translation are well typed C++ methods. Secondly, we observe that points (i), (ii) and (iii) modify classes simply...
by adding those new methods. Thus, modified classes are still well-typed, while types decrease during the translation. Then, the code that is not affected by the translation remains well typed, since any expression can be used in every context where one of greater type is required in a type safe way. Moreover, the translation procedure uses neither RTTI nor, more importantly, type downcasts. As a consequence, the code generated by the translation is type safe, which is a distinguishing feature of our approach.

EVALUATION

In the introduction we have already pointed out the main features and advantages of our proposal. In this section we want to discuss what seems the major drawback of our translation, i.e., the lack of modularity, and show some performance evaluations.

Our solution sacrifices modularity in the sense that also the target classes are modified by the translation into standard C++ (although, in the actual implementation cyclic dependences among client and target classes are not introduced, as noted in the previous section). However, we believe that generated code is efficient in the sense that it exploits dynamic binding twice, thus the invocation of a branch is independent from the number of branches of a multi method and from the hierarchy of target classes. The rational behind this choice is the same of the implementation of dynamic binding in mainstream object-oriented languages such as C++ and Java: the dynamic selection of the “right” version of a method is not performed by inspecting bottom up the class hierarchy of objects; a virtual method is invoked by accessing the virtual method table shared by objects of the same class and containing pointers to the most specialized methods. This allows to select efficiently methods at run-time in constant time (i.e., independently from the number of branches and from the class hierarchy). Following a similar approach, we do not select the right branch at run-time by checking the dynamic type of parameters (using RTTI information) but we employ the dynamic binding mechanism provided by the language twice, by dispatching the method invocation to both the client object and the target object (i.e., we actually perform double dispatch).

In order to asset the performance of our implementation of double dispatch in C++, we compare our approach with a “modular” solution that basically exploits run-time type information checks and type casts, without modifying target classes. Our preprocessor doublecpp, when given the command line

<table>
<thead>
<tr>
<th>// file T1.h, modified</th>
<th>// additional file T1_add.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>// by the preprocessor</td>
<td>// generated by the preprocessor</td>
</tr>
<tr>
<td>class A;</td>
<td>T T1::disp_m(A *a)</td>
</tr>
<tr>
<td>class B;</td>
<td>{ return a-&gt;m(this); }</td>
</tr>
<tr>
<td>class T1 {</td>
<td>T1::disp_m(B *b)</td>
</tr>
<tr>
<td>T disp_m(A *a);</td>
<td>{ return b-&gt;m(this); }</td>
</tr>
<tr>
<td>T disp_m(B *b);</td>
<td>}</td>
</tr>
</tbody>
</table>

Listing 6: The actual translation of the target class T1 of Listing 1 as performed by doublecpp. All methods are virtual.
option --modular, generates such a modular code, instead of the one described by our translation. The generated code is still based on the generation of _DB methods, but the body of these generated methods detects the right branch by using RTTI. For instance, the method m_DB of class B of the example in Listing 1 will be as follows:

```cpp
T B::m_DB(T1 *t) {
    if (T3 *j = dynamic_cast<T3 *>(t))
        return m(j);
    if (T2 *j = dynamic_cast<T2 *>(t))
        return m(j);
    return m(t);
}
```

Notice that, in such generated code, the order of the if statements matters and they have to follow the order of the hierarchy so that the most derived classes are tried first.

Since our multi methods are not encapsulated, in that the receiver and the parameter participate together in method selection by double dispatch (without using any priority), the generated _DB method checks against each target class declared in the branches of the multi method in any parent class. For instance, the method n_DB for the class B of Listing 1 will perform a dynamic type check also for the class S2 even though that branch of the multi method n is not defined in B (it is inherited from the base class):

```cpp
S B::n_DB(S1 *t) {
    if (S3 *j = dynamic_cast<S3 *>(t))
        return n(j);
    if (S2 *j = dynamic_cast<S2 *>(t))
        return n(j);
    return n(t);
}
```

This way the receiver object will not have a priority over the parameter in the dynamic branch selection.

It is easy to verify that code generated in such a way actually selects the most specialized branch of a multi method. We skip further details of this generation mechanism and its formal properties because they are not relevant to the main subject of the paper, but it is quite straightforward to show that the code generated with this approach has the same semantics of the one shown in Listing 4.

Concerning the performance evaluation, we then generated some test source programs where the depth and width of the class hierarchy of target classes vary. The client class, in such tests, declares a multi method with a branch for each target class in the hierarchy and the main procedure will invoke all the branches of the multi method (each time with a different target object). We then measured the average performance of such a code processed with doublecpp with the two different strategies that we now call optimized (the one presented in section ‘Translating Double Dispatch into C++’) and modular (the one presented in this section). The tests were executed on a Pentium III, 1Ghz, running Linux. In Figure 1 we show the time that it took to complete the tests when the target class hierarchy width is fixed to 3 (3 subclasses for each class) and the depth varies from 2 to 6 levels. Notice that in spite of the last case being a quite degenerate case (the hierarchy contains a huge amount of classes) the
optimized code does not show dramatic performance loss while the modular code does. In Figure 2 the class hierarchy depth is fixed to 2, while the width varies from 10 to 35; once again the performance loss is big only in the modular generated code. Let us remark that the class hierarchies used in these tests are not unrealistic: as discussed later in this section, a wide class hierarchy is used, for instance, for representing the nodes of an abstract syntax tree in a compiler for a programming language.

We have also computed the average time it takes to invoke a branch of a multi methods with the above mentioned tests. The results are shown in Figures 3 and 4. It should not surprise that, while for the modular programs this average time increases when the dimension of target class hierarchy increases, for the optimized programs the average time is basically constant.

As expected, the code generated according to the translation presented in this paper outperforms the one generated with the modular approach. On the other hand, the latter generated code does not need to modify the target classes; this has the advantage of not requiring recompilation of target classes. These dependences do not show a high impact on the compilation process, thanks to the generation strategy summarized in Listing 6. However, the addition of a new branch in a client class may require the recompilation of many target sources too. Summarizing, the modular approach requires less compilation but has a lower performance.

These factors concern two different kinds of users: the programmer is interested in modularity and separate compilation since less compilations will increase productivity time. On the other hand, the
end user does not care about modularity and would appreciate a better performance. doublecpp can help both kinds of users:

- the programmer can use the modular code generation strategy during the development of his programs; less compilations will be required, indeed even less than if he had used the visitor pattern directly;
- upon deployment of the application, the whole code can be re-preprocessed with doublecpp generating the faster code.

This strategy is quite typical in development contexts: the programmer usually disables compiler optimizations during the development and test phase. Indeed, these optimizations, apart from making the compilation process slower, make debugging harder. Debugging an optimized program can be difficult since, for instance, some parts of code are re-arranged by the optimizations, other parts can be inlined, and following the flow of a program with a debugger is more complicated in these situations. On the other hand, compiler optimizations are activated upon the deployment of the application. Furthermore, we observe that for small class hierarchies (say 3 classes) the difference between the two approaches is really not relevant, so, in these cases, one can always use the modular approach.

We used doublecpp in the development and refactoring of the compiler for the language X-KLAIM, an object-oriented programming language for code mobility, available on line at...
Varying class hierarchy width - average time for method invocation

Figure 3. Average time to call a branch of a multi method, when varying the depth of target class hierarchy. Time is expressed in seconds $\times 10^{-5}$.

http://music.dsi.unifi.it. This compiler used to rely heavily on the visitor pattern (indeed, visitor is often applied to compilers when showing examples of use), in order to perform many semantic checks: type checking, identifier initializations, access rights, and so on. The use of this pattern had revealed quite useful in order to have a clean program design and a flexible mechanism for activating some operations during compilation (e.g., optimizations, further checks, etc.). However, adding a new visitor or adding a new element in the syntax tree had always been a burden. We then used doublecpp for re-engineering the compiler with multi methods (applied to the visitor classes).

We benefit from doublecpp features during the development of new features of X-KCLAIM with many respects:

- using the modular code generation during the development phase shortened the compilation time significantly;
- having multi methods directly in the language allowed us to easily and quickly add new features to the language without worrying about manual implementation of visitors;
- many parts of the compiler have been made “cleaner”: during program analysis, inside visitors’ methods, we have to perform some specific actions according to the type of the program node we are visiting; in such cases we should resort to using further visitors (possibly by creating...
new visitor hierarchies, since the visit method should have a different signature). Since these parts of the compiler were small and due to laziness, in the previous versions of the compiler we simply resorted to RTTI for these parts. With doublecpp we were able to easily modify these parts by employing multi methods thus obtaining cleaner and more re-usable code.

The class hierarchy representing the abstract syntax tree nodes in the X-KLAIM compiler is quite large (about 35 classes), and not quite deep (no more than two levels) and it is implemented according the Composite pattern [3]. Thus, our context is very close to the extreme test cases discussed above. Indeed the “optimized” version of the X-KLAIM compiler (where double dispatch is achieved according to the translation presented in this paper) compiles all the sources of the compiler test suite (about 150 files) in less than 15 seconds, while it takes more than 21 seconds with the “modular” version of the compiler.

Moreover, since doublecpp itself is a sort of compiler, in many parts of the source, it has to behave differently according of the specific part of code it is analyzing and preprocessing. Then, after a first development phase (bootstrapping) we used multi methods in doublecpp as well: we used doublecpp to preprocess the source code of doublecpp itself.

Let us observe that even the modular generated code is still type safe. This is the main advantage of having a language extension together with a static type system and a preprocessor. Even in this case “message-not-understood” errors can never take place: the type casts are always checked and if no
dynamic type check succeeds in the generated _DB method, we can still call the branch with the highest target class used in this multi method (that is also the type used to declare the formal parameter of the _DB method). This does not happen in other approaches where no language extension is introduced, such as, e.g., [15, 20, 21, 22]: in such frameworks run-time exceptions may be thrown when no branch is selectable.

We think that the transformation presented in this paper, which actually is more a method addition strategy, could be performed at linking time by a (modified) C++ compiler, i.e., without actually modifying the sources. These methods could then be added directly to the file objects produced by the compiler. Notice that these method additions would not alter the memory space needed to store objects since it would only modify their method virtual tables (to which they refer to by a pointer) and, furthermore, target classes have to be polymorphic anyway in order to exploit either dynamic binding or RTTI. Notice that, at linking time, all compiled files are available. We are investigating on this issue. Furthermore, we are quite confident that the code generated by our translation, relying on dynamic binding, can further benefit from possible optimizations for virtual method invocation, such as those proposed in [23, 24].

Finally, as for code inspection, when doublecpp inspects a client class, it has to examine also the hierarchy of target classes. The inspection concerns only header files (doublecpp handles #include directives). With this respect, it may have to process many header files. This has minimal impact on the overall compilation process; indeed doublecpp inspects about 150 files in less than one second.

PRIVATE, PROTECTED AND VIRTUAL METHODS

For simplicity, in the previous sections we did not consider private and protected methods. Indeed, doublecpp can handle also these standard C++ features. This introduces a few technical complications in the translation procedure that can be solved as explained in the following. When the branches of a multi method have restricted access (private or protected) in a client class, then the target class is not able to dispatch to that branch. This issue is dealt with by adding to the client class a public version of the restricted branches and by making the target classes dispatch to these additional methods (these additional methods are actually inline so they do not introduce overhead). Thus, for instance, suppose the class A in Listing 1 defines the multi method m as private (the same holds for protected methods), then the generated code will be as in Listing 7.

Notice that the added public versions are present in the generated code only, not in the original source. If this may seem to undermine the information hiding of the client classes, we remind that the transformation could be implemented at link time, thus enabling only the target classes to access the restricted methods. Something similar happens when compiling classes with friend classes. Indeed, we may also think of a “smarter” translation, introducing a friend class of the client classes, say F, that can then access the restricted methods. We introduce the additional methods as private in this class, and we make the target classes friend of F; thus, only the target classes can access the additional methods, that in turn are the only ones allowed to access the original private methods in the client classes.

As we said in section ‘The Extended C++’, all branches of the superclass are implicitly inherited in the derived class; of course this takes place consistently with the C++ rules, i.e., only if the branches are not private in the superclass.
class A {
private:
T m_DB(T1 *t)
{
  return
  t->disp_m(this);
}
T m(T1 *t) { ... }
T m(T2 *t) { ... }
public:
T m(T1 *t) { return m(t); }
T m(T2 *t) { return m(t); }
};

class T1 {
public:
T disp_m(A *a)
{
  return
  a->m(this);
};
};

Listing 7: The translation in the presence of private methods in the client class (The multi method m is private). All methods are virtual.

In spite of considering all methods as virtual up to now, doublecpp actually requires to explicitly declare whether a multi method and its branches are virtual or not (just like in C++). Thus, the full syntax of multi methods, with respect to the one already presented, is as follows:

\[
\begin{align*}
\text{multi-method} & ::= \text{virtual branches name branches endbranches} \\
\text{branches} & ::= \text{branch} \\
\text{branch} & ::= \text{virtual type (type } \ast \text{ arg ) ;} \\
\text{virtual} & ::= \text{virtual } | \varepsilon
\end{align*}
\]

Declaring a multi method as virtual ensures that all the branches in the class hierarchy are considered during branch selection to execute the most specialized one. Declaring a single branch as virtual, instead, has exactly the same meaning of declaring a (overloaded) method as virtual in C++. Finally, we notice that, by using `virtual` for multi method and branches in appropriate combinations, one can implement several flavors of multi method semantics, even encapsulated multi methods (e.g., by not using `virtual` at all).

**TOWARDS MULTIPLE DISPATCH**

In this paper we limited our multi methods to double dispatch, so that only one parameter is considered during dynamic method selection. We did not see this as a strong limitation from a pragmatic point of view. Indeed, most of the examples found in the literature dealing with multi methods consider only two parameters: we observe that these works provide multi methods as overloaded functions, instead of overloaded methods ([12, 14] provide both forms), where the first parameter is the implicit invocation object. It is quite unusual to have to deal with more than double dispatch. Other related approaches, such as, e.g., [25, 26], focus on double dispatch only, saying that two-argument dispatch occurs much more frequently than three or more argument dispatch (e.g., as reported in [27], 84% of the multiple dispatch messages consider only two arguments in Cecil). For instance, all design patterns...
class C {
  branches m
  void (T1 ∗t, S1 ∗s);
  void (T2 ∗t, S2 ∗s);
endbranches
};

Listing 8: An example of multiple dispatch.

class C {
  void m_DB(T1 ∗t, S1 ∗s) {
    t->disp_m(s, this);
  }
  void m(T1 ∗t, S1 ∗s);
  void m(T2 ∗t, S2 ∗s);
};

class T1 {
  void disp_m(S1 ∗s, C ∗c) {
    s->disp_disp_m(c, this);
  }
  void disp_m(S2 ∗s, C ∗c) {
    s->disp_disp_m(c, this);
  }
};

class S1 {
  void disp_disp_m(T1 ∗t, C ∗c) {
    c->m(t, this);
  }
};

class S2 : public S1 {
  void disp_disp_m(T2 ∗t, C ∗c) {
    c->m(t, this);
  }
};

Listing 9: Translated code for Listing 8. All methods are virtual.

[3] that would benefit from multi methods involve two class hierarchies only. Furthermore, also the other important example of usage of multi methods, i.e., binary methods, deals with only two objects (of the same class hierarchy).

In spite of our practical restriction to double dispatch, the approach presented in this paper can be generalized in order to deal with multiple dispatch in a smooth way, without conceptual problems. Although we are still working on this subject, namely on technical details and implementation, in the following we provide some hints about this generalization.

Let us consider, for instance, the code in Listing 8, where the branches of the multi method \textit{m} uses two parameters for dynamic overloading, where \textit{T2} \leq \textit{T1}, \textit{S2} \leq \textit{S1}, and \textit{T1} and \textit{S1} are unrelated. Now, the informal idea to achieve multiple dispatch is that the added method \textit{m_DB} in \textit{C} will invoke a \textit{disp}_\textit{m} method in \textit{T1}, which in turn will invoke a \textit{disp}_\textit{disp}_\textit{m} method in \textit{S1}, which will finally select the right branch among the ones of the multi method \textit{m}.

The code generated by the preprocessor would be like in Listing 9. Notice that, as in the case of double dispatch, the basic idea of our approach is still to use both dynamic binding and static
overloading in order to achieve the same semantics of multiple dispatch. Let us now consider the following code:

```cpp
C *c = new C;
T1 *t = new T2;
S1 *s = new S2;

c->m_DB(t, s);
```

and interpret its execution:

1. `c::m_DB(t, s)` will execute `t->disp_m(s, this)`; since `t` is of type `T2` dynamically, due to dynamic binding, `T2::disp_m(S1 *s, C *c)` will be selected;
2. `T2::disp_m(S1 *s, C *c)` will execute `s->disp_disp_m(c, this)`; `s` is of dynamic type `S2` and `this` is of static type `T2`, thus `S2::disp_disp_m(T2 *, C *)` will be selected;
3. when `S2::disp_disp_m(T2 *, C *)` executes `c->m(t, this)`, since `t` is of static type `T2` and `this` is of static type `S2`, due to static overloading `C::m(T2 *, S2 *)` will be selected.

Thus, we achieved the semantics of multiple dispatch.

If we complicate the example of Listing 8 by introducing a derived client class, as in Listing 10, then the code generated by the preprocessor would be as in Listing 11, where, for brevity, we omit the translation of `T2` and `S2` that are the same as `T1` and `S1` respectively.

Again, it is straightforward to verify that the following code:

```cpp
C1 *c = new C2;
T1 *t = new T3;
S1 *s = new S3;

c->m_DB(t, s);
```

will select dynamically the most specialized branch, i.e., `C2::m(T3 *, S3 *)`.

As for extending the type system to handle multiple dispatch, this is basically even more straightforward: since our typing is based on the one of [12, 14], and since the languages and type systems presented therein already deal with multiple dispatch, all we have to do is to extend the syntax of our types in order to deal with tuple types instead of pair types. All the other typing issues will then naturally extend to multiple dispatch. Let us stress that static well-typedness will continue to guarantee the absence of run-time errors.

Finally, if we consider multiple dispatch, it would be convenient to let the programmer specify which parameters should be considered during dynamic overloading selection, and which should not. With
### Listing 11: Translated code for Listing 10. All methods are virtual.

```cpp
class C1 {
    void m_DB(T1 *t, S1 *s) {
        t->disp_m(s, this);
    }
    void m(T1 *t, S1 *s);
    void m(T2 *t, S2 *s);
};

class C2 : public C1 {
    void m_DB(T1 *t, S1 *s) {
        t->disp_m(s, this);
    }
    void m(T3 *t, S3 *s);
};

class T1 {
    void disp_m(s, C1 *c) {
        s->disp_m(s, C1 *c);
    }
    void disp_m(S2 *s, C1 *c) {
        s->disp_m(s, C1 *c);
    }
    void disp_m(S2 *s, C2 *c) {
        s->disp_m(s, C2 *c);
    }
    void disp_m(S3 *s, C2 *c) {
        s->disp_m(s, C2 *c);
    }
};

class T2 {
    void disp_m(S1 *s, C2 *c) {
        s->disp_m(s, C2 *c);
    }
    void disp_m(T1 *t, C2 *c) {
        c->m(t, this);
    }
};

class S1 {
    void disp_m(T1 *t, C1 *c) {
        c->m(t, this);
    }
    void disp_m(T2 *t, C2 *c) {
        c->m(t, this);
    }
};

class S2 {
    void disp_m(T1 *t, C3 *c) {
        c->m(t, this);
    }
};

class S3 {
    void disp_m(T1 *t, C3 *c) {
        c->m(t, this);
    }
};
```

In this respect, we might think to adopt the syntax proposed in [16, 17] of declaring the former with the keyword `virtual` in front of them.

With respect to multi methods as functions as proposed in [16, 17], we could easily add them to our implementation by simply considering a multi method in the shape of a function, that is

```cpp
void f(virtual C *c, virtual T *t);
```

as a method to be added in the class C taking as parameter a pointer to T; from this point on, the translation scheme applies just the same.

Let us observe that in the generalization to multiple dispatch hinted above the time complexity of a multi method invocation remains constant in the sense that it is independent from the number of branches and from the class hierarchy (while it is linear in the number of arguments).
Multiple dispatch table techniques

We recall that problems of space and time complexity arise in implementing dynamic dispatch. Indeed, extending single dispatch to multiple dispatch, the total size of the structures that are needed for recording all the method branches (for all possible combinations of the argument types, not only of the parameter types) dramatically increases. To this aim, the papers mentioned in the following propose non-trivial data structures, of tractable memory size, that can be dealt with in an efficient way.

We just mention that other proposals such as [28] are cache based approaches mainly concerning dynamically typed languages and then they are unrelated to our statically typed context.

Compressed dispatch tables. By using an $n$-dimensional table, indexed by all the possible $n$ argument types, the main disadvantage is the huge amount of memory required to store these tables. The size is given by $T^n$, where $T$ is the number of types in the system, and $n$ is the number of the arguments.

For example, for a large project with 1000 types, a 3-argument message would have a table with 1000$^3$ entries. If a memory address takes four bytes, then about four gigabytes would be necessary just to store the table of this message. Since, on average, there are four definitions for each message, the potential redundancy in this table is enormous. Each entry refers to one of these definitions, or to an error routine if the message was not defined for the type combination under consideration. Many of the table rows are identical. These problems are the starting point for many proposed optimizations:

- **selector coloring** [29, 30] aims at eliminating null entries in tables by slicing the set of messages. Each slice must satisfy the following property: two messages in the slice cannot be recognized by the same type, so in each sub-table a row can have at most one null entry. This property makes it possible to merge all the columns in a sub-table. The performance of selector coloring improves when the number of slices decreases. Since it is computationally heavy to find an optimal slicing, then the slices must be found using a heuristic.

- **by row displacement** [31], the rows are displaced in the dispatching matrix by different offsets so that they can be merged together in a master array. In [32], a displacement of columns rather than rows (called selector based row displacement) is used to obtain a better compression.

- **by compact selector tables** [33, 34], the compression results of row displacement are improved on some hierarchies. This technique aims at duplicates elimination. The idea is to partition the set of messages into disjoint subsets: slicing breaks the dispatching table into sub-tables. Identical rows within each sub-table are then merged.

Look-up automata. [35] and [36] create a look-up automaton for each multi method. In order to avoid backtracking and thus exponential dispatch time, the automata must include more types than those that are explicitly listed in method definitions. The most efficient implementation of the above automata is via a $n$-dimensional table compressed using the techniques listed above. A disadvantage of this proposals is that they concern languages that satisfy *inheritance order precedence* [37] that is not supported by common object oriented languages.

Single receiver projections. The proposal of [38] maintains single receiver dispatch tables and projects multi methods definitions of arity $k$ onto such $k$ tables. Each table maintains a bit vector of applicable method indices so dispatch consists of logically anding bit vectors, finding the index of
the right-most on-bit and returning the method associated with this index. These vectors are typically compressed to reduce space complexity using the techniques listed above.

Discussion. All the above proposals, based on a direct implementation of multiple dispatch, have the crucial issue of optimizing the compression of the $n$-dimensional dispatch tables in order to achieve a good performance of any multi-method invocation. Furthermore, other proposals (such as [39]) address another complexity metric: the creation time, i.e., the time required for generating the dispatching data structures. This is crucial in a context of dynamic recompilation.

Completely different, our approach implements multiple dispatch by only using single dispatch and static overloading. Thus, all the time and space complexity problems of the $n$-dimensional tables discussed above do not concern our proposal. Indeed we only use standard 1-dimensional virtual tables (VTBL) and our tables do not suffer from all the problems that arise in implementing $n$-dimensional tables such as empty or repeated entries. In our approach, we can reuse the natural subtyping relations on types, thus avoiding to add an entry for all the possible argument types (those used in a multi method invocation): we can restrict to any possible parameter types (those used in multi method branch declarations), whose number is, of course, no greater than the number of all the possible argument types. Instead, in table-based multiple dispatch approaches, since there is no subtyping relation on the indexes of the tables, they have to take into consideration all the possible types of the arguments for any method invocation. In particular, let us consider the case when a new class, which is a subtype of an existing multi method branch parameter type, is added into an existing program, and this new type is used only as an argument for a multi method invocation (i.e., not for defining a new branch). Then everything will continue to work without any further program transformation, in our solution. Instead, in the approaches using $n$-dimensional dispatch tables, these tables should be re-computed. This also clarifies that, although our approach is not modular, it does not require the whole program code, but only the parts that define the types involved in multi method branch definitions. On the contrary, dispatch table based approaches, by their own nature, require that all the types of the program are considered when generating entries of the tables.

Furthermore, we can benefit of all the techniques that can be used to gain a better performance in the case of single dispatch [40, 41, 24, 42]. For instance, dispatch overhead can be reduced by eliminating some dispatches: when a branch is provably monomorphic, its target procedure can be inlined. Static analysis [43, 41, 24] can determine the concrete receiver types of calls, possibly eliminating dynamic dispatch for many invocations. According to [40], a low average-case dispatch cost with a low worst case cost can be achieved by combining caches with standard virtual function tables. Thus, the efficiency depends on the percentage of method invocations that are handled by the cache. For example, [44] shows that at least 66% of all virtual calls in C++ could be handled without cache misses.

Finally, we want to stress what we consider a crucial feature of our approach from the point of view of the efficiency: there is no overhead in parts of programs that do not use method invocation by dynamic overloading. Thus, in the extended C++, the parts of the programs that only use single dispatch have the same cost as in standard C++. Moreover, as explained in [8], if a program in a single-dispatching language is written by using additional dispatching after methods are called to resolve problems caused by binary methods, then such a program will be no faster than the equivalent multi-method program. This makes our solution a fully conservative extension of C++ and, in general, of
standard object oriented languages. Further discussions on performance issues have been presented in section ‘Evaluation’.

RELATED WORKS

In the literature there are many other proposals for implementing multi methods. For simplicity, we discuss different approaches dividing them into three main categories. The list of works referred in this section is however not to be considered exhaustive.

Languages with multi methods

We just cite three main languages with multi methods and multiple dispatch that, however, due to their features, are quite different from C++ and Java paradigm, which is the setting of our study. CLOS [45] is a class-based language with a linearization approach to multi methods: they are ordered by prioritizing argument position with earlier argument positions completely dominating later ones. This automatic handling of ambiguities may lead to programming errors. In Dylan [46] methods are not encapsulated in classes but in generic functions which are first class objects. When a generic function is called it finds the methods that are applicable to the arguments and selects the most specific one. BeCecil [47] is a prototype based language with multi methods. Multi methods are collected in first-class generic function objects which can extend other objects. Even if this language is object-based, it provides a static type system, scoping and encapsulation of all declarations; however, its approach, being object-based is radically different from our class-based setting.

Extensions of existing languages

Among the papers that add multiple dispatch to existing languages, the work that is closer to ours is the one of parasitic methods [13], which are a variant of encapsulated multi methods [8, 14] applied to Java. This extension is rather flexible and indeed provides multiple dispatch instead of double dispatch. It is thought to be modular (thus it does not introduce many dependences among classes) a choice that has influenced many aspects of the design:

- parasitic methods are encapsulated so the receiver is evaluated before the argument, in method selection;
- the selection of the most specialized method takes place through instanceof checks and consequent type casts, thus it does not perform constantly as in our solution, but essentially linearly on the number of branches;
- parasitic methods are complicated by the use of textual order of methods in order to resolve ambiguities for selecting the right branch;
- all methods must be declared in the class of the receiver in order to eliminate class dependences.

The price to pay is that the class hierarchies of the multi methods arguments must be anticipated limiting flexibility.

MultiJava [19] is an extension of Java to support open classes (classes to which new methods can be added without editing the class directly) and (not encapsulated) multi methods. Once again
it provides a modular type checking at some cost. To avoid paying modularity with lack of flexibility (as in parasitic methods), MultiJava allows the use of multi methods only with open classes syntax and only for programs which import open classes definitions. If subclasses wish to specialize multi methods, then additional open classes are required and the compilation may result in multiple layers of type-case double dispatch. The method selection is performed through a cascade of if statements to test types of arguments at run-time showing a bad performance (as analyzed in [48]).

Cmm [49] is a preprocessor providing multi methods in C++. Cmm uses a log(N) cache of dispatches (where N is the number of different combinations of dynamic types seen for a particular multi-method) and also supports an amortized constant-time mode (that uses RTTI only to initialize the function table). Cmm is based on the proposal of [17]. The drawback of this solution is that exceptions can be raised due to missing branches and ambiguities, thus losing the advantage of having a type-safe linguistic extension. We plan to execute performance evaluations comparing the amortized constant time approach of Cmm with the constant time of double dispatch, in order to see whether there are situations where one performs better than the other and, in such cases, whether a combined approach can be employed to implement dynamic overloading.

Other approaches

Other approaches add multiple dispatch to Java without type checking the proposed extension: JMMF [22] is a framework implemented using reflection mechanism, while in [21] a new construct is created using ELIDE (a framework implemented to add high level features to Java). The major drawback of these proposals is that type errors, due to missing or ambiguous branches, are caught at run time by exceptions.

[20] proposes an extension of the JVM to provide multi dispatch in Java without modifying neither the syntax nor the type system: the programmer directly selects the classes which should use multiple dispatch. The problem of this approach is that code written for single dispatch is roughly switched to multiple dispatch so arising problems for ambiguous method calls and return types.

Other proposals on the same subject, such as [50, 51], are characterized by the fact that they do not provide any automatic means for preprocessing the code, thus they are more similar to a pattern or idiom (as for the Visitor). Moreover some of them are targeted to a specific scenario, e.g., binary methods, or they require the programmer much manual programming, typically without static checks for correctness. Also [48] follows this approach by providing a library extension to implement a limited form of multiple dispatch in Java, based on a visitor-like code. Again these approaches suffer from possible run-time exceptions.

Chapter 11 of [15] presents some approaches that allow to implement double dispatch in C++ through a smart use of generic programming (templates). This approach does not extend the language and run-time errors due to missing branches can still be raised. Furthermore, while the use of templates decreases the amount of code that has to be written explicitly (w.r.t. to other solutions based on Java), the programmer is still required to write some code to achieve double dispatch. In some cases, he even has to provide the hierarchy order of target classes. Moreover, some of the approaches presented in [15] are not able to handle objects of classes derived from the target classes specified in the multi methods (i.e., they do not work correctly with inheritance).
CONCLUSIONS

We have proposed an extension of C++ with multi methods and double dispatch for method invocation. The translation into standard C++ shows how dynamic overloading can be implemented by using only static overloading and dynamic binding in a type safe way. This translation essentially consists in modifying client and target classes in such a way that it can be performed by a preprocessor. The approach is general enough to be applied to other OO languages, such as Java and C#.

Concerning further developments, theoretical issues of our proposal will be presented in a companion paper. In this formal setting: (i) a toy language is defined, including a kernel C++ and the proposed extension; (ii) the semantics of both the kernel C++ and the toy language is defined by translation into \( \lambda_{\text{object}} \) [12, 14], which is a meta language for modeling object-oriented languages; (iii) metatheory of our proposal is so stated in the formal setting of \( \lambda_{\text{object}} \), which provides a formal suitable framework yet very close to actual implementations of object-oriented languages.

We are also working on implementing a generalization of our approach to multiple dispatch, based on a chain of dispatch invocations that involve all the parameters of a multi method as hinted in section ‘Towards Multiple Dispatch’.

ACKNOWLEDGEMENTS

We thank the anonymous referees for many helpful remarks and suggestions.

REFERENCES

3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


51. B. Fraser. Implementing a Double Dispatcher that Respects Inheritance. Available at http://www.apmaths.uwo.ca/~bfraser/c++. 