WOOD 2004 Preliminary Version

Translating Double Dispatch into Single Dispatch

Lorenzo Bettini Sara Capecchi Betti Venneri

Dipartimento di Sistemi e Informatica, Università di Firenze, Via Lombroso 6/17, 50134 Firenze, Italy
{bettini,capecchi,venneri}@dsi.unifi.it

Abstract

Goals of flexibility and re-usability in typed object-oriented languages suggest the requirement of double dispatch, i.e., the mechanism of dynamically selecting 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. However, many mainstream languages, such as, e.g., C++ and Java, do not provide it, resorting to only single dispatch. In this paper we present a general technique for adding double dispatch as a type-safe language feature, so yielding dynamic overloading and covariant specialization of methods, without extending basic semantics. To this aim we introduce a toy core language, extended to a full form of (non encapsulated) multi methods. Then we define a translation algorithm from multi methods to the core language, that implements double dispatch by using only standard mechanisms of static overloading and single dispatch. As a main feature, our translation preserves type safety, it uses neither RTTI nor type downcasts and does not introduce crucial overhead during method selection.

Key words: Double Dispatch, Multi Methods, Dynamic Overloading, Program Transformation.

1 Introduction

Double dispatch is the ability of dynamically selecting 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 (when all arguments are considered, we have *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 manually explore 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, these means are a necessary solution to get around the lack of double dispatch in the language [19]. 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 *Visitor* pattern [16], 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.

On the other hand, it is well known that double dispatch improves the flexibility of object-oriented programming; namely, it enables safe *covariant specialization* of methods [21,3,7], where subclasses are allowed to redefine a method by specializing its argument. For instance, covariant specialization provides a smooth solution to the problem of *binary methods* [5] (i.e., methods that act on objects of the same type: the receiver and the argument). It seems quite natural to specialize binary methods in subclasses and to require that the new code replaces the old definition when performing code selection at run-time according to the actual type of the argument and the receiver.

In this paper we show how *multi methods* [12,22,8,4], supporting the mechanisms of double dispatch and covariant specialization, can be implemented by static overloading and single dispatch. A multi method can be seen as a set of branches associated to the same message name, i.e., an overloaded method, but the selection takes place dynamically according to multiple dispatch. In our approach, we limit multi methods to double dispatch. Our goal is to define a general technique to extend a language with double dispatch (and dynamic overloading). In order to do so, we present a kernel object-oriented language (inspired by C++ and Java) that supports both dynamic and static overloading (i.e., both double and single dispatch) and we provide a translation algorithm that, given a program that uses double dispatch, produces an equivalent program (i.e., with the same semantics) that uses only single dispatch. This translation is thought to be executed automatically by a program translator (a preprocessor) that has to be run before the actual language compiler. We applied this translation algorithm to implement double dispatch in C++ as a language extension that is translated by a preprocessor. The implementation of such a preprocessor, `doublecpp`, (freely available at `http://www.lorenzobettini.it/software/doublecpp`) has been 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 related approaches:

- We provide a full form of multi methods that are not encapsulated (see, e.g., [5,9]) at the cost of sacrificing modularity. With encapsulated multi methods the method selection is performed firstly according to the run-time class of the receiver and then according to the run-time class of the argument. Instead, in our solution, the receiver and the argument participate equally to the method selection by double dispatch without using 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 translation is conservative since it preserves static overloading, while adding the additional form of dynamic overloading. In particular, the new mechanism is semantically consistent with static overloading and standard dynamic binding.
- The code generated by our translation uses neither RTTI nor, more importantly, type downcasts, which are very common in other proposals (discussed in Section 5) and that are notoriously sources of type safety violations. Indeed, our solution is characterized by the crucial issue of preserving type safety.
- 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. The benchmarks applied to our implementation `doublecpp` actually confirmed this.

We would like to stress that our translation is not an automatic implementation of the visitor pattern [16] (although it may remind its structure); instead it is 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) [16]. 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 code in order to obtain a similar effect. Moreover, the visitor pattern tends to simulate the encapsulated multi-method functionality, and this makes this pattern hard to use when derived classes want to implement additional branches (e.g., while with our approach binary methods can be directly implemented, they are difficult to implement using the visitor pattern).

2 toy^{DO}: a minimal core language with double dispatch

In this section we present a kernel language, toy^{DO}, formally defined in Table 1. It is used to:

(i) abstract the basic object-oriented features of mainstream languages, such as Java and C++, that are relevant for our purposes;

(ii) support multi method definitions and multi method selection by dynamic overloading (double dispatch).

Regarding the second point, toy^{DO}, is widely inspired by the language KOOL of Castagna [8,9]. However, differently from the functional nature of KOOL, toy^{DO} is an imperative language dealing with side effects.

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{classdef}^{\,*} \quad \textit{methdef}^{\,*} \quad \textit{main} \\[1em]
\textit{classdef} & ::= & \textbf{class}\ A\ \ : A_1\ \{ \\
 & & \quad T_i\ f_i\ =\ \textit{exp}_i;\ ^{i \in I} \\
 & & \quad \textit{branchmethdecl}_j;\ ^{j \in J} \\
 & & \}; \\[1em]
\textit{branchmethdecl} & ::= & \textit{message}\ \textbf{branches} \\
 & & \quad T'_k\ (T_k\ x);\ ^{k \in K} \\
 & & \textbf{end} \\[1em]
\textit{exp} & ::= & x \\
 & | & \textbf{this} \\
 & | & \textbf{this}.\ell \\
 & | & \textit{methinvok} \\
 & | & \textbf{new}\ \textit{classname} \\[1em]
\textit{methdef} & ::= & T'\ \textit{classname} :: \textit{message}(T\ x)\ \textit{body} \\[1em]
\textit{main} & ::= & \textit{body} \\[1em]
\textit{body} & ::= & \{\textit{localdecl}; \textit{stmnt}; \textbf{return}\ \textit{exp}\} \\[1em]
\textit{localdecl} & ::= & T_1\ x_1; \ldots; T_n\ x_n \\[1em]
\textit{stmnt} & ::= & \textit{methinvok} \\
 & | & \textit{left}\ =\ \textit{exp} \\
 & | & \textit{stmnt}_1; \textit{stmnt}_2 \\[1em]
\textit{left} & ::= & x \\
 & | & \textbf{this}.\ell \\[1em]
\textit{methinvok} & ::= & \textit{receiver} \Leftarrow \textit{message}(\textit{exp}') \\
 & | & \textit{receiver} \leftarrow \textit{message}(\textit{exp}') \\[1em]
\textit{receiver} & ::= & \textit{exp} \\
 & | & \textbf{super}
\end{array}
$$

Table 1
The syntax of the toy language

As an aid to abstraction, we adopt the following simplifications in the syntax of toy^{DO}:

- class fields are all private while methods are all public;
- methods always return a value (since method bodies include a final return statement);
- a program is simply made of a sequence of class definitions, followed by a sequence of method definitions (class definitions only declare the signature of methods, not their implementations), followed by a sequence of statements that play the same role of `main` in C++ and Java;
- we allow method bodies to call the implementation of a method in the superclass with the construct **super** (this superclass method is selected statically);
- since we are interested in studying only double dispatch (and not the more general multiple dispatch), we further simplify our language in that methods accept only one parameter. Moreover, as in Java and C++, where the return type is not considered in the overloaded method selection, a sub-class cannot change the return type of a method;
- **class** $A : A_1$ means that A_1 is the superclass of A. In this paper, for simplicity, we allow only single inheritance. At the end of Section 4.2 we hint on how multiple inheritance can be treated (as it is in the implementation of `doublecpp`). Subclassing implies subtyping, here denoted by $\leq$ (i.e., $A \leq A_1$).

Finally, concerning the key point of the toy^{DO} syntax, multi methods are written according to the following rule: a multi method is defined as a set of branches associated to the same message name, i.e., an overloaded method. Intuitively all branches of a multi method represent different behaviors on different arguments. A subclass can extend the definition of a multi method by providing additional branches and redefining some of them. The crucial issue in our language concerns method invocation. In order to clarify the treatment of double dispatch we distinguish two different linguistic constructs for method invocation, $\leftarrow$ and $\Leftarrow$:

- $\mathit{receiver} \leftarrow \mathit{message}(\mathit{exp}')$ is used for the standard static overloaded method invocation, i.e., the branch of the multi method is selected statically according to the static type of the parameter (*static overloading*), and dynamically according to the run time type of the receiver (*single dispatch*);
- $\mathit{receiver} \Leftarrow \mathit{message}(\mathit{exp}')$ is used for dynamic overloaded method invocation, where the branch of the multi method is selected dynamically according both to the run time type of the receiver and to the run time type of the parameter (*double dispatch*);

where *receiver* can be either an *exp* or **super**.

```
class A {                  class B : A {
  m branches                 m branches
   T (T1 t);                  T (T2 t);
   T (T2 t);                  T (T3 t);
  end                        end

  n branches                 n branches
   S (S1 t);                  S (S3 t);
   S (S2 t);                 end
  end                      };
};
```

Fig. 1. Code in *toy*^{DO} using multi methods

Let us illustrate informally the semantics of method selection by a simple example. Consider the code in Figure 1 (assuming `Tj` $\leq$ `Ti` if $i \leq j$) and the following code:

A a = **new** A;
$T1$ t = **new** $T2$;
$a \leftarrow m(t)$; // *static overloading*
$a \Leftarrow m(t)$; // *dynamic overloading*

Then, the first method invocation is performed according to static overloading semantics, i.e., `A::m(T1)` is selected, while the second method invocation, performed according to dynamic overloading semantics, selects `A::m(T2)`.

The $\Leftarrow$ construct enables *covariant specialization* of the parameter type in the branches of a multi method. Indeed, B redefines the branch `m(T2)` but also specializes the multi method by adding a new branch, `m(T3)`, where `T3` $\leq$ `T2`. Again, the most specialized branch of a multi method will be dynamically selected for invocation on objects belonging to subclasses. Thus, considering the following code:

A a = **new** B;
$T1$ t = **new** $T1$;
$a \Leftarrow m(t)$; // *dynamic overloading*
t = **new** $T2$;
$a \Leftarrow m(t)$; // *dynamic overloading*
t = **new** $T3$;
$a \Leftarrow m(t)$; // *dynamic overloading*

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

Let us remark that the receiver and the parameter participate together in the dynamic selection of the method, without any priority, as in [8] and in the *symmetric* multiple dispatch of [11]. Differently, *encapsulated multi methods* [5,9] are characterized by the fact that the receiver has the precedence over the parameters.

3 Typing multi methods and method invocation

The type system of the toy language is quite standard as far as it concerns the core object oriented constructs. For lack of space, we omit a full treatment of this issue and we restrain our attention to typing issues concerning multi methods and method selection: in the following, main points of this subject are sketched.

We use the notion of multi types (widely inspired by [8,9]) for typing multi methods. A *multi type* Σ is of the form:

$$\Sigma = \{I_1 \to T_1, \ldots, I_n \to T_n\}$$

where each input type I_i is a pair of types $(A \times B)$.

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

$$A \leq A_1, B \leq B_1 \Rightarrow A_1 \to B \leq A \to B_1$$

Then, $\Sigma_1 \leq \Sigma_2$ if and only if for every arrow type in Σ_2 there is at least one smaller (or equal) arrow type in Σ_1.

Well formedness of multi types.

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

(i) all input types are pairwise distinct;

(ii) if $I_i \leq I_j$ then $T_i \equiv T_j$;

(iii) if I_i and I_j have a set of common subtypes, then for each set of common subtypes there must be exactly one arrow type $I_k \to T_k$ in Σ such that I_k is the maximal type of this set.

Condition (i) is quite standard on overloaded definitions. Condition (ii) is specific to multi methods; we require $T_i \equiv T_j$ (instead of $T_i \leq T_j$ as in [8,9]) according to the philosophy of C++ and Java where return types are not used in overloaded method selection. Condition (iii) concerns the absence of ambiguities in well typed method selection (see below).

Notice that one may think of adopting a more liberal type discipline and check the third condition (absence of ambiguity) only at method invocation time. This is the strategy used by, e.g., C++ and Java. This would make the type checking more complex and, while for static overloading this could still be feasible, in the presence of dynamic overloading it would make the type checking also more inefficient, since, every time a method is used, the whole class hierarchy involved in that method invocation should be checked, in order to ensure that no ambiguities will take place dynamically. Moreover, as hinted in Section 5, we develop the formal theory of our translation using λ_object [8,9] that is based on these well-formedness conditions.

3.1 Typing rules

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 type of the parameter and the return type.

Typing rule for multi method declaration.

A multi method m in the class C has the multi type Σ

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

iff

- $A_1 \to T_1, \ldots, A_n \to T_n$ are the types of all the branches of m as defined in C;
- Σ' is the (possibly empty) multi type of m in the superclass of C;
- Σ is well formed.

Remark

Let us notice that, by well-formedness, overriding some branch of a multi method in a subclass can require to override other branches in order to obtain a well formed multi method. For instance, if a class A defines the two branches for the multi method m with the following types $T_1 \to T$ and $T_2 \to T$, where $T_2 \leq T_1$, and the first branch of m is overridden in a subclass B of A, then the resulting multi type of m, $\Sigma = \{(A \times T_1) \to T, (A \times T_2) \to T, (B \times T_1) \to T\}$ is not well formed, due to the third condition. In order to have a well typed overriding of this multi method, the programmer should redefine also the other branch (e.g., by simply calling the superclass implementation) so obtaining the well formed multi type $\Sigma = \{(A \times T_1) \to T, (A \times T_2) \to T, (B \times T_1) \to T, (B \times T_2) \to T\}$. From the practical point of view, a more flexible policy should consist in automatically inserting such trivial overridings when needed. More generally, all branches of a multi method in the superclass could be automatically inserted in the derived class, provided that they are not redefined in the latter, so obtaining a full form of *copy semantics of inheritance* [2]. Our solution is simpler but equivalent from the formal point of view. The implementation `doublecpp` uses the full form of copy semantics, thus the programmer is not forced to write such trivial method overriding.

Typing rule for multi method invocation.

The invocation of m (both with the construct $\leftarrow$ and $\Leftarrow$) on a receiving object of type A with an argument of type B has type T iff there is an arrow type $(A' \times B') \rightarrow T \in \Sigma$ such that $(A \times B) \leq (A' \times B')$.

The above conditions of well-formedness play a crucial role in typing method invocation. Since Σ is well formed, condition (ii) ensures that the type T is unique for any such pair $(A' \times B')$. Most importantly, condition (iii) relates to the question of selecting the correct branch when interpreting the invocation of m both by static and dynamic overloading semantics. Namely by condition (iii) there exists one and only one branch whose input type "best approximates" both the pair of static types $(A \times B)$, and any pair of dynamic types $(A_1 \times B_1)$ such that $(A_1 \times B_1) \leq (A \times B) \leq (A' \times B')$.

Thus, the evaluation of a well typed method invocation has two important features:

(i) preserves the static type (*subject reduction*),

(ii) does not reach a "stuck state" due to ambiguities or message-not-understood, since it always reduces to a selection of a branch which is determined without ambiguities (*progress property*).

4 From double dispatch to single dispatch

In this section we show how to translate any toy^{DO} program that uses double dispatch ($\Leftarrow$) into an equivalent program that uses only single dispatch (dynamic binding) and static overloading ($\leftarrow$). To this aim, we will use toy^{SO} to denote the subset of toy^{DO} defined as in Table 1 except for the clause of method invocation which is reduced to one construct only, i.e., the static overloading method invocation:

$$\mathit{receiver} \leftarrow \mathit{message}(\mathit{exp})$$

The translation algorithm from toy^{DO} to toy^{SO} is defined in Section 4.2, after an informal presentation of the basic idea in Section 4.1.

4.1 The informal basic idea

In order to give a basic idea of the proposed translation we consider an example. For simplicity, we write both the declaration and the definition of branches in the class declaration and we omit **return**.

Suppose we have the following class definition (on the left), where T_i's are in the subtyping relation $T_i \leq T_j$ if $j \leq i$, and the following piece of code (on the right):

```
class $C_1${                          
  $m$ branches                        $C_1\ c\ =\ $ new $C_1$;
    $T\ (T_1\ x)$;                    $T_1\ t\ =\ $ new $T_2$;
    $T\ (T_2\ x)$;                    $c \Leftarrow m(t)$
  end
}
```

In this program the second branch of m in C_1 will be selected, since, in spite of being declared statically as T_1, t is of type T_2 dynamically.

Let classes C_1, T_1 and T_2 be modified as follows:

```
class $C_1${
  $m$_DB branches
    $T\ (T_1\ x)$                       class $T_1${                 class $T_2 : T_1${
    {$x \leftarrow$ disp_$m$(this)}      disp_$m$ branches           disp_$m$ branches
  end;                                   $T\ (C_1\ x)$                  $T\ (C_1\ x)$
  $m$ branches                             {$x \leftarrow m$(this)}       {$x \leftarrow m$(this)}
    $T\ (T_1\ x)$;                       end                           end
    $T\ (T_2\ x)$;                     }                             }
  end
}
```

then we translate the method invocation $c \Leftarrow m(t)$ as $c \leftarrow m$_DB(t). It is easy to verify that:

(i) the method invocation $x \leftarrow$ disp_m(**this**) will select the (only) branch of disp_m in T_2, since dynamic binding is employed also in the static overloading invocation;

(ii) the method invocation $x \leftarrow m$(**this**) in T_2 will use static overloading, and thus will select a branch of m in C_1 according to the static type of the argument: the argument **this** is of type T_2 and thus the second branch of m in C_1 will be selected.

Thus, $c \leftarrow m$_DB(t) in the translated program results in having the same behavior of $c \Leftarrow m(t)$ in the original program.

Now let us make our example a little bit more complex ($T_4 \leq T_2$):

```
class $C_2 : C_1${
  $m$ branches                        $C_1\ c\ =\ $ new $C_2$;
    $T\ (T_4\ x)$;                    $T_1\ t\ =\ $ new $T_4$;
  end                                 $c \Leftarrow m(t)$
}
```

Again the dynamic overloading semantics will select the $T\ (T_4\ x)$; in C_2. In this case the program would be translated as follows (the translation of class

C_1 is just the same as before, and the translation of T_1 and T_2 are identical):

```
class $C_2 : C_1${                       class $T_1${
  $m$_DB branches                          disp_$m$ branches
    $T$ ($T_1$ $x$)                          $T$ ($C_1$ $x$)
    {$x \leftarrow$ disp_$m$(this)}          {$x \leftarrow m$(this)};
  end;                                       $T$ ($C_2$ $x$)
  $m$ branches                               {$x \leftarrow m$(this)}
    $T$ ($T_4$ $x$);                       end
  end                                    }
}

  class $T_2 : T_1${                     class $T_4 : T_2${
    disp_$m$ branches                      disp_$m$ branches
      $T$ ($C_1$ $x$)                        $T$ ($C_2$ $x$)
      {$x \leftarrow m$(this)};              {$x \leftarrow m$(this)}
      $T$ ($C_2$ $x$)                      end
      {$x \leftarrow m$(this)}           }
    end
  }
```

Again, let us interpret the method invocation $c \leftarrow m$_DB(t) (replaced to $c \Leftarrow m(t)$):

(i) since dynamic binding is employed, the implementation of the branch of m_DB in C_2 will be selected dynamically;

(ii) the method invocation $x \leftarrow$ disp_m(**this**) will select statically the second branch of disp_m in T_1, since **this** is (statically) of type C_2, but since dynamic binding is employed, the version of such method provided in T_4 will actually be invoked dynamically;

(iii) the method invocation $x \leftarrow m$(**this**) in T_4 will select a branch of m in C_2 according to the static type of the argument: the argument **this** is of type T_4 and thus the branch T (T_4 x) of m in C_2 will be selected.

Thus, $c \leftarrow m$_DB(t) in the translated program has the same behavior of $c \Leftarrow m(t)$ in the original program.

Summarizing, the idea of our translation is that the dynamic overloading semantics can be obtained, in a static overloading semantics language, 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.

Notice that our translation introduces a new multi method for each multi method m with the same name plus the suffix _DB (for DouBle dispatch). The branches of this new multi method m_DB in a class C_i are built starting from the branches of m as follows:

- we consider the set τ of all the parameters types T_j of m in C_i (including the ones of the branches inherited from all the superclasses);

- we then consider the subset of τ containing the maximal types and for each type T_i in this subset we add a branch to m_DB in C_i with parameter of type T_i. For instance, if the set of parameter types is $\{T_1, T_2, T_3, S_1, S_2\}$, where $T_3 \leq T_2 \leq T_1$ are unrelated with $S_2 \leq S_1$, then the subset of maximal types is $\{T_1, S_1\}$. The correctness of this operation relies on well formedness of multi types.

For instance considering the classes defined above and their translation:

- the multi method m in class C_1 has parameters of type T_1 and T_2. T_1 is the maximal so the overloaded method m_DB in C_1 has a branch with a parameter of type T_1;
- again the multi method m in class C_2 has a parameter of type T_4 and inherits from C_1 two branches with parameters T_1 and T_2. T_1 is the maximal among these types T_2 and T_4 so the overloaded method m_DB in C_2 has a branch with a parameter of type T_1.

Similarly, a new multi method disp_m is introduced in the T_i's classes. The branches of disp_m are built as follows. Let us consider the definition of the multi method m in a class C_i. For each type T_i, such that T_i is the type of the parameter of a branch in m, we add a branch to disp_m in the class T_i with parameter of type C_i. We add the same branch to disp_m in each class T_j such that $T_j \geq T_i$ and T_j is the type of the parameter of a branch of m in C_i or in a superclass C_k ($C_k \geq C_i$).

Let us consider again the classes above:

- T_1 and T_2 are the parameter types of the branches of m in C_1, thus we add a branch to disp_m in T_1 and T_2, with parameter of type C_1.
- T_4 is the type of the parameter of the branch of m in C_2 so we add a branch to disp_m in T_4, with parameter of type C_2. Moreover, since $T_4 \leq T_2$ and since T_2 is the type of parameter of a branch of m in C_1 ($C_2 \leq C_1$), we add a branch to disp_m in T_2, with parameter of type C_2; recursively, we add such a branch to T_1, since $T_2 \leq T_1$ and since T_1 is the type of parameter of a branch of m in C_1 ($C_2 \leq C_1$).

Thus, the method m_DB aims at statically using the type of C_i and dynamically using the type of the T_j, while the method disp_m has exactly the opposite task. Together these two methods implement the dynamic overloading semantics.

Let us observe that in case a derived class, say $C_3 \leq C_2$, does not introduce any new specialized branch in m, but only redefines the inherited ones, then there is no need of such an additional m_DB method: the most redefined version will be anyway selected thanks to dynamic binding. For the same reason branches of disp_m is not required for such a class C_3 in the classes T_i's. Moreover, possible intermediate classes T_j's, that are not used as parameter types in any branches, are not modified by the translation.

4.2 The translation algorithm

We present the translation algorithm in a top-down style, defining in the following each additional procedure we use. We would like to observe that the translation is defined on well-typed programs, so we assume properties concerning correct programs in defining the algorithm. Furthermore, all types and subtyping relations are considered according to the type and subtyping environments collected during type checking algorithm, and are not made explicit in the translation algorithm.

Definition 4.1 [Specializing type] Let m **branches** $T'_l\ (T_l\ x);\ ^{l \in L}$ **end** be the declaration of a multi method. Then we say that each T_l is a *specializing type* for m (in the sense that its type is used for the formal parameter in the specialization of the method m, i.e., in one of the branches of m).

Conventions.
In the definition of the algorithm, we will use the following conventions:

- *MD* represents the set of method definitions in the current program;
- $A \leftarrow\!\!\leftarrow T'\ m(T\ x)$ means that the declaration of the branch $T'\ (T\ x)$ is added to the multi method m in the class A in the current program, of course, if the branch is not already present. If the multi method does not exist in A, it also creates the multi method declaration;
- $MD \leftarrow T'\ A :: m(T\ x)\ body$ means that the method definition $T'\ A :: m(T\ x)\ body$ is added to the current program, of course, if it is not already present;
- *superclass*(A) returns the direct super class of A or **nil** if A has no superclass.

Definition 4.2 [isspecializingtype predicate] *isspecializingtype*(A, m, T, T') checks whether the type T is used by A, or by a superclass of A, as the parameter type in a branch of the multi method m (returning an object of type T'):

$$
\begin{array}{l}
\mathit{isspecializingtype}(A, m, T, T') \overset{def}{=} \\
\quad m : \{(A_i \times T_{j_i}) \to T'_{j_i}\ ^{i \in I\ j_i \in J_i}\} \wedge \\
\qquad \exists i \in I \text{ such that } A \leq A_i \wedge \\
\qquad\quad \exists j_i \in J_i \text{ such that } T \equiv T_{j_i} \wedge T' \equiv T'_{j_i}
\end{array}
$$

Translation algorithm
The translation of a *toy*^{DO} program into an equivalent *toy*^{SO} program is as follows:

```
for each multi method $m_k : \{(A_i \times T_{j_i}) \to T'_{j_i}\ ^{i \in I\ j_i \in J_i}\}$ and for each $i \in I$
and $j_i \in J_i$:
    $\mathit{addmethod}(A_i, m_k, T_{j_i}, T'_{j_i})$
    $\mathit{adddispatch}(A_i, m_k, T_{j_i}, T'_{j_i})$
```

Thus the translation algorithm essentially consists in calling two procedures for each branch of each multi method m: the first one to build m_DB, the second one to build disp_m. These two sub-procedures are defined as follows:

(i) *addmethod*(A, m, T, T') adds a branch to m_DB in the class A; the type of the added branch is not necessarily $T \rightarrow T'$, but it can be $T'' \rightarrow T'$ where $T \leq T''$: indeed, T'' will be the sup among all the types T_l such that $T \leq T_l$ and T_l is used as a specializing type for m in A or in a superclass of A.

```
$addmethod(A, m, T, T') \overset{def}{=}$
    **if** $m : \{(A_i \times T_{j_i}) \rightarrow T'_{j_i}\ {}^{i \in I\ j_i \in J_i}\}$ **then**
    **let** $T_h = \sup_{\leq}\{T_l | T \leq T_l \wedge \exists i \in I, j_i \in J_i . j_i = l \wedge A \leq A_i\}$ **in**
        $A \hookleftarrow T'\ m\_\mathtt{DB}(T_h\ x)$
        $MD \leftarrow T'\ A :: m\_\mathtt{DB}(T_h\ x)\ \{\mathbf{return}\ x \leftarrow \mathtt{disp\_}m(\mathbf{this})\}$
```

Notice that m_DB can have several branches: indeed the branches of the overloaded method m that is being translated may use specializing types coming from unrelated hierarchies: for each of these hierarchies we have to add a branch to m_DB.

(ii) *adddispatch*(A, m, T, T') adds a branch to disp_m with parameter type A in the class T, provided that T is a specializing type for m in A or in a superclass of A. Moreover, it updates all the superclasses of T by calling itself recursively:

```
$adddispatch(A, m, T, T') \overset{def}{=}$
    **if** $isspecializingtype(A, m, T, T')$ **then**
        $T \hookleftarrow T'\ \mathtt{disp\_}m(A\ x)$
        $MD \leftarrow T'\ T :: \mathtt{disp\_}m(A\ x)\ \{\mathbf{return}\ x \leftarrow m(\mathbf{this})\}$
    **endif**
    **if** $superclass(T) \neq$ **nil then**
        $adddispatch(A, m, superclass(T), T')$
    **endif**
```

Notice that the procedure skips (i.e., does not modify) possible intermediate classes in the hierarchy of T that are not specializing type for any branch of m.

Finally, each dynamic overloading method invocation $exp \Leftarrow m(exp')$ is translated into $exp \leftarrow m\_\mathtt{DB}(exp')$.

A relevant property of the translation procedure defined above is *preservation of typing*: every well typed program of toy^{DO} is translated into a well typed program of toy^{SO}. For lack of space, we cannot give a formal account of this issue. We just mention the crucial technical steps:

(i) the new m_DB and disp_m multi methods, added during translation, are well typed (in particular, for any multi method m, of multi type Σ, the corresponding m_DB is given a subtype of Σ);

(ii) any class, which is modified by adding new methods, remains well typed.

Finally, concerning a practical evaluation, the translated code is efficient in the sense that it exploits dynamic binding twice, thus method invocation is independent from the number of branches of the multi method and from the depth and width of class hierarchies. 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; method invocation is performed by accessing the virtual method table shared by objects of the same class and containing pointers to the most specialized methods (see, e.g., [20]). This allows to efficiently select methods at run-time in constant time (i.e., independently from the number of methods 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 the parameter (using RTTI information) but we employ the dynamic binding mechanism provided by the language twice, by dispatching the method invocation both to the receiver and to the parameter (i.e., we actually perform double dispatch).

Remark: Multiple inheritance. The extension of the algorithm to the case of multiple inheritance requires some additional steps in defining the two sub-procedures *addmethod* and *adddispatch*, in order to ensure that the newly created multi methods m_DB and disp_m satisfy condition (iii) of well formedness also in the presence of multiple, unrelated superclasses. Indeed, multiple inheritance is dealt with in `doublecpp`.

5 Conclusion and related works

We proposed a translation algorithm that allows to implement, in a type safe way, double dispatch (i.e., dynamic overloading and covariant specialization) in a language that provides only static overloading and dynamic binding. This translation essentially consists in modifying classes in such a way that it can be performed by a preprocessor. The approach is general enough to be applied to many imperative OO languages, such as Java, C++ and C#.

Concerning further developments, theoretical issues of our proposal will be presented in a companion paper, where: the semantics of the language toy^{DO} is defined by translation into λ_object [8,9], which is a meta language for modeling object-oriented features. Metatheory of our proposal is so stated in the formal setting of λ_object, which provides a formal suitable framework yet very close to actual implementations of object-oriented languages. At the best of our knowledge, λ_object is the only theoretical model that directly treats multi methods and dynamic overloading, so it seems to be a suitable framework for our purposes.

We are also working on how to generalize our approach to implement multiple dispatch instead of only double dispatch. This generalization will be based on a chain of dispatch invocations that involve all the parameters of a multi method.

A drawback of our approach is the absence of modularity that also affects separate compilation. This is due to the fact that also classes used as parameters of multi method branches are modified by our translation. Solutions based on run time type checking and dynamic casts do not suffer from this problem; conversely these solutions are characterized by a method selection complexity that depends on the number of branches and on the number of classes in the hierarchy, while with our solution dynamic method overloading selection is constant (as further discussed later in this section). Modularity and optimized code 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 would appreciate a better performance. We took these requirements in consideration when implementing `doublecpp`: when given the command line option `--modular`, it 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. Of course, the two strategies generate equivalent code. Then, we can imagine the following development scenario:

- 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.

We have also used the two different strategy to asset the performance of our approach and, as expected, the code generated by our translation outperforms the one that uses dynamic type checking and casts.

We think that the transformation presented in this paper, which is 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, at linking time, all compiled files are available. We are investigating on this issue.

Concerning related works, we just mention languages that directly support multi methods and multiple dispatch as a language feature, *Dylan* [23], *BeCecil* [10], *CLOS* [18], and we concentrate on those that are based on program transformation.

The work that is closer to ours is the one of *parasitic methods* [4], an extension of Java allowing to implement multi methods. It is thought to be modular, 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 methods 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 has to 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 has to be anticipated limiting flexibility. By comparison, our approach implements non encapsulated multi methods (at cost of sacrificing modularity), does not use RTTI mechanisms and results in a better performance.

MultiJava [11] is an extension of Java language to support open classes (classes to which new methods can be added without editing the class directly) and (non encapsulated) multi methods. At cost of many restrictions, MultiJava allows the use of multi methods only with open classes syntax and only for programs which import open classes definitions; method selection is performed through a cascade of if statements to test types of arguments at run-time.

Cmm [24] is a preprocessor providing CLOS-style multi methods in C++. Also in this case, type casts and RTTI are heavily exploited thus decreasing the performance during these method invocations. Furthermore, run-time exceptions can be raised due to missing branches.

Other approaches provide multiple dispatch in Java without extending the type system: *JMMF* [14] is a framework implemented using reflection mechanism, while in [6] a new construct is created using ELIDE (a framework implemented to add hight 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.

[13] 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 rising problems for ambiguous method calls and return types.

Other proposals on the same subject, such as [17,15] 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 an idiom (as for the Visitor). Moreover, some of them are targeted to a specific scenario, such as basically designed for binary methods, or they require the programmer much manual programming, typically without static checks for correctness.

Chapter 11 of [1] presents some solutions 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 explicitly 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 [1] 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).

Acknowledgments

We thank the anonymous referees for their helpful suggestions to clarify some aspects of the paper.

References

[1] Alexandrescu, A., "Modern C++ Design, Generic Programming and Design Patterns Applied," Addison Wesley, 2001.

[2] Ancona, D., S. Drossopoulou and E. Zucca, *Overloading and Inheritance*, in: *FOOL 8*, 2001.

[3] Bancilhon, F., C. Delobel and P. Kanellakis (eds.), "Implementing an Object-Oriented database system: The story of O_2," Morgan Kaufmann, 1992.

[4] Boyland, J. and G. Castagna, *Parasitic Methods: Implementation of Multi-Methods for Java*, in: *Proc of OOPSLA '97*, ACM SIGPLAN Notices **32(10)**, ACM, 1997, pp. 66–76.

[5] Bruce, K. B., L. Cardelli, G. Castagna, T. H. O. Group, G. Leavens and B. C. Pierce, *On binary methods*, Theory and Practice of Object Systems **1** (1995), pp. 217–238.

[6] Carbonetto, P., *An implementation for multiple dispatch in java using the elide framework*, available at `http://www.cs.ubc.ca/~pcarbo/`.

[7] Castagna, G., *Covariance and contravariance: conflict without a cause*, ACM Transactions on Programming Languages and Systems **17** (1995), pp. 431–447.

[8] Castagna, G., *A meta-language for typed object-oriented languages*, Theoretical Computer Science **151** (1995), pp. 297–352.

[9] Castagna, G., "Object-Oriented Programming: A Unified Foundation," Progress in Theoretical Computer Science, Birkhauser, 1997.

[10] Chambers, C. and G. T. Leavens, *BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing*, in: *FOOL 4*, 1996.

[11] Clifton, C., G. Leavens, C. Chambers and T. Millstein, *MultiJava: modular open classes and symmetric multiple dispatch for Java*, ACM SIGPLAN Notices **35** (2000), pp. 130–145.

[12] DeMichiel, L. and R. Gabriel, *The Common Lisp Object System: An Overview*, in: *Proc. ECOOP*, LNCS **276** (1987), pp. 151–170.

[13] Dutchyn, C., P. Lu, D. Szafron, S. Bromling and W. Holst, *Multi-Dispatch in the java virtual machine: Design and implementation*, in: *Proc. of the 6th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS '01)*, 2001, pp. 77–92.

[14] Forax, R., E. Duris and G. Roussel, *Java multi-method framework*, in: *Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS '00)*, 2000.

[15] Fraser, B., *Implementing a Double Dispatcher that Respects Inheritance*, available at `http://www.apmaths.uwo.ca/~bfraser/c++/`.

[16] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[17] Ingalls, D., *A Simple Technique for Handling Multiple Polymorphism*, in: *Proc. OOPSLA* (1986), pp. 347–349.

[18] Keene, S., "Object-Oriented Programming in Common Lisp," Addison-Wesley, 1989.

[19] Lea, D., *Run-Time Type Information and Class Design*, in: *Proc. USENIX C++ Technical Conference* (1992), pp. 341–347.

[20] Lippman, S., "Inside the C++ Object Model," Addison-Wesley, 1996.

[21] Meyer, B., "Eiffel: The Language," Prentice-Hall, 1991.

[22] Mugridge, W., J. Hamer and J. Hosking, *Multi-Methods in a Statically-Typed Programming Language*, in: *Proc. ECOOP '91*, LNCS **512** (1991), pp. 307–324.

[23] Shalit, A., "The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language," Addison-Wesley, Reading, Mass., 1997.

[24] Smith, J., *Cmm - C++ with Multimethods*, Association of C/C++ Users Journal (2001), `http://www.op59.net/cmm/readme.html`.