A safe implementation of dynamic overloading in Java-like languages*

Lorenzo Bettini[1], Sara Capecchi[1], Betti Venneri[2]

[1]Dipartimento di Informatica, Università di Torino
[2]Dipartimento di Sistemi e Informatica, Università di Firenze
[1]bettini,capecchi@di.unito.it,[2]venneri@dsi.unifi.it

Abstract. We present a general technique for extending Java-like languages with dynamic overloading, where method selection depends on the dynamic type of the parameter, instead of just the receiver. To this aim we use a core Java-language enriched with encapsulated multi-methods and dynamic overloading. Then we define an algorithm which translates programs to standard Java code using only basic mechanisms of static overloading and dynamic binding. The translated programs are semantically equivalent to the original versions and preserve type safety.

Key words: Language extensions, Multi-methods, Dynamic Overloading

1 Introduction

A *multi-method* can be seen as a collection of overloaded methods, called *branches*, associated to the same message, but the method selection takes place dynamically according to the run-time types of both the receiver and the arguments, thus implementing *dynamic overloading*. Though multi-methods are widely studied in the literature, they have not been added to mainstream programming languages such as Java, C++ and C#, where overloading resolution is a static mechanism (the most appropriate implementation of an overloaded method is selected statically by the compiler according to the static type of the arguments).

In this paper we present a general technique for extending Java-like languages with dynamic overloading. To this aim we use the core languages FDJ and FSJ which are extensions of Featherweight Java [7] with multi-methods and static overloading, respectively. Then, we define an algorithm which translates FDJ programs to FSJ code using only basic mechanisms of static overloading and dynamic binding. Both FDJ and FSJ are based on [2], where we studied an extension of FJ with static and dynamic overloading from the linguistic point of view; thus, in that paper, we focused on the type system and the crucial conditions to avoid statically any possible ambiguities at run-time. The multi-methods we are considering are *encapsulated* in classes and not external functions, and the branch selection is *symmetric*: during dynamic overloading selection the receiver type of the method invocation has no precedence over the argument types.

* This work has been partially supported by the MIUR project EOS DUE.

The translation here presented can be regarded as a formalization of a general technique to implement dynamic overloading in mainstream object-oriented languages. The translation is type safe (i.e., the generated code will not raise type errors) and the translated code will have the same semantics of the original program using dynamic overloading. In particular, since the translated code uses only static overloading and dynamic binding, it does not introduce a big overhead (it performs method selection in constant time, independently from the width and depth of inheritance hierarchies) as in other approaches in the literature [4, 6]. In [3] we presented `doublecpp`, `http://doublecpp.sf.net`, a preprocessor for C++ which is based on the approach here presented ([3] also sketches a very primordial and informal version of the translation algorithm); the translation presented in this paper is the first formalization of our technique for implementing dynamic overloading through static overloading.

We briefly present the parts of the core language FDJ (*Featherweight Java + Dynamic overloading*), which is an extension of FJ (*Featherweight Java*) [7] with multi-methods, that are relevant for the translation algorithm (we refer to [2] for further details). In the following we assume that the reader is familiar with FJ, then we will concentrate on the features of FDJ. The syntax of FDJ is the following:

L ::= class C extends C {$\overline{\texttt{C f}}$; K; $\overline{\texttt{M}}$}	classes
K ::= C($\overline{\texttt{C f}}$){super($\overline{\texttt{f}}$); this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;}	constructors
M ::= C m (C x){return e;}	methods
e ::= x | e.f | e.m(e) | new C($\overline{\texttt{e}}$)	expressions
v ::= new C($\overline{\texttt{v}}$)	values

The distinguishing feature, w.r.t. FJ, consists in the definition of multi-methods: the programmer is allowed to specify more than one method with the same name and different signatures; any definition of a method `m` in a class `C` is interpreted as the definition of a new branch of the multi-method `m` (this difference w.r.t. FJ is evident in the typing [2]). The new branch is added to all the other branches of `m` that are inherited (if they are not redefined) from the superclasses of `C` (*copy semantics of inheritance* [1]). In FDJ we also permit method redefinition with a covariant return type, a feature that has been recently added to Java. In the present paper we limit multi-methods to one parameter. We do 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 one parameter.

A program is a pair $(CT, \texttt{e})$ of a class table (mapping from class names to class declarations) and an expression `e` (the program's main entry point). The subtyping relation $<:$ on classes (types) is induced by the standard subclass relation. The types of FDJ are the types of FJ extended with *multi-types*, representing types of multi-methods. A multi-type is a set of arrow types associated to the branches of a multi-method, and is of the shape: $\{\texttt{C}_1 \to \texttt{C}'_1, \ldots, \texttt{C}_n \to \texttt{C}'_n\}$. We will write multi-types in a compact form, by using the sequence notation: $\{\overline{\texttt{C} \to \texttt{C}'}\}$. Σ will range over multi-types. We extend the sequence notation also to multi-method definitions: $\overline{\texttt{C}'\ \texttt{m}\ (\texttt{C}\ \texttt{x})\{\texttt{return e;}\}}$ represents a sequence of method definitions, each with the same name `m` but with different signatures (and possibly different bodies): $\texttt{C}'_1\ \texttt{m}\ (\texttt{C}_1\ \texttt{x})\{\texttt{return e}_1\texttt{;}\}\ \ldots\ \texttt{C}'_n\ \texttt{m}\ (\texttt{C}_n\ \texttt{x})\{\texttt{return e}_n\texttt{;}\}$. The multi-type of the above multi-method will be denoted by $\{\overline{\texttt{C} \to \texttt{C}'}\}$.

Multi-types are constrained by two crucial consistency conditions, formulated in [5], which must be checked statically, in order to be well-formed: a multi-type $\{\overline{\mathtt{B} \to \mathtt{B}'}\}$ is *well-formed* if $\forall\ (\mathtt{B}_i \to \mathtt{B}'_i), (\mathtt{B}_j \to \mathtt{B}'_j) \in \{\overline{\mathtt{B} \to \mathtt{B}'}\}$ the following conditions are verified: 1) $\mathtt{B}_i \neq \mathtt{B}_j$, 2) $\mathtt{B}_i <: \mathtt{B}_j \Rightarrow \mathtt{B}'_i <: \mathtt{B}'_j$. The first condition requires that all input types are distinct. The second condition guarantees that a branch specialization is safe: if statically a branch selection has a return type, and if dynamically a more specialized branch is selected, the return type is consistent with the static selection (it is a subtype). The original definition of well-formedness of [5] also contained a third condition, which is always implied by the other two conditions in our context where we only have single inheritance and one parameter.

The $mtype(\mathtt{m},\mathtt{C})$ lookup function returns the type of `m` in the class `C` which is a multi-type. In particular, since we consider copy semantics of inheritance, the multi-type contains both the signatures of the branches defined (or redefined) in the current class and the ones inherited by the superclass:

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \qquad \overline{\mathtt{B}'\ \mathtt{m}\ (\mathtt{B}\ \mathtt{x})\{\mathtt{return\ e};\}} \in \overline{\mathtt{M}}}{mtype(\mathtt{m},\mathtt{C}) = \{\overline{\mathtt{B} \to \mathtt{B}'}\} \cup \{\mathtt{B}_i \to \mathtt{B}'_i \in mtype(\mathtt{m},\mathtt{D}) \mid \forall \mathtt{B}_j \to \mathtt{B}'_j \in \{\overline{\mathtt{B} \to \mathtt{B}'}\}, \mathtt{B}_i \neq \mathtt{B}_j\}}$$

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mtype(\mathtt{m},\mathtt{C}) = mtype(\mathtt{m},\mathtt{D})}$$

Note that we cannot implement $mtype(\mathtt{m},\mathtt{C})$ simply as $\{\overline{\mathtt{B} \to \mathtt{B}'}\} \cup mtype(\mathtt{m},\mathtt{D})$ due to possible method overriding with covariant return types.

The semantics of method invocation is type driven in that it uses *mtype* to select the most specialized version among the set of matching branches (it is unique, by well-formedness, if that set is not empty). The selected method body is not only the most specialized w.r.t. the argument type but also the most redefined version associated to that signature. This way, we model standard method overriding inside our mechanism of dynamic overloading. The language FSJ has the same syntax as FDJ but the multi-methods are intended as standard overloaded methods and then the overloading resolution is static.

2 From FDJ to FSJ: the Translation Algorithm

We now use FDJ and FSJ to formalize the transformation from an extended Java with dynamic overloading to standard Java (with static overloading): in this section we show how *multi-methods* can be implemented by static overloading and dynamic binding. Our goal is to define a general technique to extend a language with dynamic overloading. The solution presented here is inspired by the one described by Ingall in [8] (on which also the Visitor pattern is based), but it does not suffer from possible implementation problems when implementing manually this technique.

We provide a translation algorithm that, given an FDJ program using dynamic overloading, produces an equivalent FSJ program only using 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 actual language compiler. Note that the code generated by our translation uses neither RTTI nor, more importantly, type downcasts which are very common in other proposals and that are notoriously sources of type

safety violations. Thus, we provide a formal framework to reason about correctness of compilers when adding dynamic overloading to a language.

In order to give an informal idea of the proposed translation, let us consider the following example (for simplicity, in the following, we will use the full Java syntax, e.g., assignments and sequentialization).

Suppose we have the following FDJ program, where $\mathtt{B}_2 <: \mathtt{B}_1$ are not shown:

```
class A1{
    C m (B1 x){return e1;}
    C m (B2 x){return e2;}
};
```

```
A1 z = new A1();
B1 y = new B2();
z.m(y);
```

Then we consider the semantics of the method invocation `z.m(y)`: it will select the second branch of `m` in $\mathtt{A}_1$ since, in spite of being declared statically as $\mathtt{B}_1$, `y` is of type $\mathtt{B}_2$ dynamically.

Let classes $\mathtt{A}_1$, $\mathtt{B}_1$ and $\mathtt{B}_2$ be written in FSJ as follows:

```
class A1{
  C m (B1 x){return x.disp_m(this);}
  C m (B2 x){return x.disp_m(this);}
  C _m (B1 x){return e1;}
  C _m (B2 x){return e2;}
}
```

```
class B1{
  // original contents
  C disp_m (A1 x){return x._m(this);}
}
```

```
class B2 extends B1{
  // original contents
  C disp_m (A1 x){return x._m(this);}
}
```

Summarizing, all method definitions are renamed by adding the (reserved) prefix `_` and all the branches of the original multi-methods (including the ones implicitly inherited with copy semantics) are modified using the forward invocation `x.disp_m(this)`. Now let us analyze how the method invocation `z.m(y)` proceeds in FSJ:

1. `z.m(y)` will select the branch `C m (B1 x){return x.disp_m(this);}` in $\mathtt{A}_1$ (remember that `y` is statically of type $\mathtt{B}_1$);
2. `x.disp_m(this)` will select the (only) branch of `disp_m` in $\mathtt{B}_2$, since dynamic binding is employed also in the static overloading invocation;
3. `x._m(this)` in $\mathtt{B}_2$ will use static overloading, and thus will select a branch of `_m` in $\mathtt{A}_1$ according to the static type of the argument: the argument `this` is of type $\mathtt{B}_2$ and thus the second branch of `_m` in $\mathtt{A}_1$ will be selected.

Therefore, `z.m(y)` in the translated FSJ program, where classes are modified as above, has the same behavior as in the FDJ original program (since they execute the same method body $\mathtt{e}_2$). Consider the same classes of the previous example and the following additional classes (where $\mathtt{B}_3 <: \mathtt{B}_2$ and $\mathtt{C}' <: \mathtt{C}$):

```
class A2 extends A1{
  C' m (B3 x){return e3;}
};
```

```
A1 z = new A2();
B1 y = new B3();
z.m(y);
```

The dynamic overloading semantics will select the branch $\texttt{C}'\ \texttt{m}(\texttt{B}_3\ \texttt{x})$ in $\texttt{A}_2$. In this case the program would be translated in FSJ as follows (the translation of class $\texttt{A}_1$ is the same as before):

```
class A2 extends A1{
  C m (B1 x){return x.disp_m(this);}
  C m (B2 x){return x.disp_m(this);}
  C' m (B3 x){return x.disp_m(this);}
  C' _m (B3 x){return e3;}
}
```

```
class B1{
  // original contents
  C disp_m (A1 x){return x._m(this);}
  C disp_m (A2 x){return x._m(this);}
}
```

```
class B2 extends B1{
  // original contents
  C disp_m (A1 x){return x._m(this);}
  C disp_m (A2 x){return x._m(this);}
}
```

```
class B3 extends B2{
  // original contents
  C' disp_m (A2 x){return x._m(this);}
}
```

Let us interpret the method invocation $\texttt{z.m(y)}$ in FSJ:

1. As in the previous example, $\texttt{z.m(y)}$ will select the branch $\texttt{C m (B}_1\ \texttt{x)\{return} \ldots\texttt{;\}}$ (remember that $\texttt{x}$ is statically of type $\texttt{B}_1$);
2. since dynamic binding is employed, the implementation of $\texttt{m}$ in $\texttt{A}_2$ will be selected dynamically;
3. the method invocation $\texttt{x.disp\_m(this)}$ will select statically the second branch of $\texttt{disp\_m}$ in $\texttt{B}_1$, since $\texttt{this}$ is (statically) of type $\texttt{A}_2$, but since dynamic binding is employed, the version of such method provided in $\texttt{B}_3$ will be actually invoked dynamically (note that $\texttt{disp\_m}$ in $\texttt{B}_3$ is an override of $\texttt{disp\_m}$ in $\texttt{B}_2$ with covariant return type, which is sound);
4. the method invocation $\texttt{x.\_m(this)}$ in $\texttt{B}_3$ will select a branch of $\texttt{\_m}$ in $\texttt{A}_2$ according to the static type of the argument: the argument $\texttt{this}$ is of type $\texttt{B}_3$ and thus the branch $\texttt{C}'\ (\texttt{B}_3\ x)$ of $\texttt{\_m}$ in $\texttt{A}_2$ will be selected.

Again, $\texttt{z.m(y)}$ in FSJ has the same behavior as in FDJ (they both execute the method body $\texttt{e}_3$). The reader can easily verify that if $\texttt{y}$ is assigned an instance of $\texttt{B}_2$ we would execute $\texttt{e}_2$, just as in the first example, i.e., the body of the branch with parameter $\texttt{B}_2$ as defined in $\texttt{A}_1$, implicitly inherited by $\texttt{A}_2$. 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 and static overloading twice: this way the dynamic selection of the right method is based on the run time types of both the receiver and the argument of the message.

Note that the key point in our translation is to rename by $\texttt{\_m}$ every method $\texttt{m}$ and then introduce a new overloaded method $\texttt{m}$, which is the *entry point* for dynamic overloading interpretation. The branches of this new multi-method $\texttt{m}$ in a FSJ class $\texttt{A}_i$ are built starting from the branches of the original FDJ $\texttt{m}$ by considering the set of all the parameters types $\texttt{B}_j$ of $\texttt{m}$ in $\texttt{A}_i$ (including the ones inherited by copy semantics). One might be tempted to say that the added methods $\texttt{C m (B}_{1,2}\ \texttt{x)\{return x.disp\_m(this);\}}$ in $\texttt{A}_2$ are useless since they would be inherited from $\texttt{A}_1$: however, their presence is fundamental

to make our translated code work, since what matters in this context is the static type of `this`; thus, these methods must be present also in $\mathtt{A}_2$.

Our translation also relies on a new multi-method `disp_m` introduced in the $\mathtt{B}_i$'s classes. Let us consider how the branches of `disp_m` are built. Regarding the definition of the multi-method `m` in a FDJ class $\mathtt{A}_i$, for each type $\mathtt{B}_i$, such that $\mathtt{B}_i$ is the type of the parameter of a branch of `m`, we add a branch to `disp_m` in the class $\mathtt{B}_i$ with parameter of type $\mathtt{A}_i$. We add the same branch to `disp_m` in each class $\mathtt{B}_j$ such that $\mathtt{B}_i <: \mathtt{B}_j$ and $\mathtt{B}_j$ is the type of the parameter of a branch of `m` in $\mathtt{A}_i$ or in a superclass $\mathtt{A}_k$ ($\mathtt{A}_i <: \mathtt{A}_k$).

Let us consider the classes in the second example:

- $\mathtt{B}_1$ and $\mathtt{B}_2$ are the parameter types of the branches of `m` in $\mathtt{A}_1$, thus we add a branch to `disp_m` in $\mathtt{B}_1$ and $\mathtt{B}_2$, with parameter of type $\mathtt{A}_1$.
- $\mathtt{B}_3$ is the type of the parameter of the branch of `m` in $\mathtt{A}_2$ so we add a branch to `disp_m` in $\mathtt{B}_3$, with parameter of type $\mathtt{A}_2$. Moreover, since $\mathtt{B}_3 <: \mathtt{B}_2$ and since $\mathtt{B}_2$ is the type of parameter of a branch of `m` in $\mathtt{A}_1$ ($\mathtt{A}_2 <: \mathtt{A}_1$), we add a branch to `disp_m` in $\mathtt{B}_2$, with parameter of type $\mathtt{A}_2$; recursively, we add such a branch to $\mathtt{B}_1$, since $\mathtt{B}_2 <: \mathtt{B}_1$ and since $\mathtt{B}_1$ is the type of parameter of a branch of `m` in $\mathtt{A}_1$ ($\mathtt{A}_2 <: \mathtt{A}_1$).

To insert the correct `disp_` methods following the strategy above, it is enough to retrieve all the branches of a multi-method in a class (including those inherited with copy semantics); for each of such methods in a class $\mathtt{A}_i$, say $\mathtt{B}'\ \mathtt{m}(\mathtt{B}_j\ \mathtt{x})\ \{\ldots\}$, we insert in $\mathtt{B}_j$ a method $\mathtt{B}'\ \mathtt{disp\_m}(\mathtt{A}_i\ \mathtt{x})\ \{\mathtt{return\ x.\_m(this);}\}$. It is easy to verify that this procedure adds exactly all and only the `disp_` that are required to make our translation work. Summarizing, the method `m`, in the FSJ translated version, aims at statically using the type of $\mathtt{A}_i$ and dynamically using the type of the $\mathtt{B}_j$, while the method `disp_m` has exactly the opposite task. Together these two methods realize the dynamic overloading semantics.

Let us now present formally our translation, which is defined on well-typed FDJ programs, so assuming properties concerning well typedness of programs. In the following we introduce some auxiliary notations: given a set of method definitions $\overline{\mathtt{M}}$, a method definition `M`, a method name `m`, a class table *CT* and $\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\mathtt{;}\ \overline{\mathtt{M}}\} \in \mathit{dom}(CT)$ we use the following notations:

- $\overline{\mathtt{M}} \setminus \mathtt{m} = \{\mathtt{B}'\ \mathtt{m}'(\mathtt{B\ x})\ \{\mathtt{return\ e;}\} \in \overline{\mathtt{M}} \mid \mathtt{m}' \neq \mathtt{m}\}$
- $\mathit{rename}(\overline{\mathtt{M}},\mathtt{m}) = (\overline{\mathtt{M}} \setminus \mathtt{m}) \cup \{\mathtt{B}'\ \mathtt{\_m}(\mathtt{B\ x})\ \{\mathtt{return\ e;}\} \mid \mathtt{B}'\ \mathtt{m}(\mathtt{B\ x})\ \{\mathtt{return\ e;}\} \in \overline{\mathtt{M}}\}$
- $\mathit{rename}(CT,\mathtt{C},\mathtt{m})$ is the class table CT' that is obtained from *CT* by renaming the methods with name `m` in the class `C`, i.e., CT' is such that:
  - $CT'(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\mathtt{;}\ \mathit{rename}(\overline{\mathtt{M}},\mathtt{m})\}$
  - $\forall \mathtt{C}' \in \mathit{dom}(CT)$ such that $\mathtt{C}' \neq \mathtt{C}$, $CT'(\mathtt{C}') = CT(\mathtt{C}')$
- $CT(\mathtt{C}) \leftarrow\!\!\!\!\leftarrow \mathtt{M}$ is the class table CT' obtained from *CT* by adding to `C` the method definition `M`, i.e., CT' is such that:
  - $CT'(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{M}\}$
  - $\forall \mathtt{C}' \in \mathit{dom}(CT)$ such that $\mathtt{C}' \neq \mathtt{C}$, $CT'(\mathtt{C}') = CT(\mathtt{C}')$

Definition 1 (Translation algorithm from FDJ to FSJ). *Let $p = (CT, \mathtt{e})$ be well typed* FDJ *program, then the corresponding* FSJ *translated version, denoted by $(\overline{CT}, \mathtt{e})$, is obtained from p by performing the following algorithm:*

1. $\overline{CT} := CT$
2. $\forall$`class C extends D` $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \in dom(CT)$
   (a) $\forall \texttt{m} \in Names(\overline{\texttt{M}})$
      i. $\overline{CT} := rename(\overline{CT}, \texttt{C}, \texttt{m})$
      ii. $\forall \texttt{B} \to \texttt{B}' \in mtype(\texttt{m}, \texttt{C})$, *using CT*
         A. $\overline{CT} := \overline{CT}(\texttt{B}) \leftarrow\!\!- \texttt{B}'\ \texttt{disp\_m(C x)}\ \{\texttt{return x.\_m(this);}\}$
         B. $\overline{CT} := \overline{CT}(\texttt{C}) \leftarrow\!\!- \texttt{B}'\ \texttt{m(B x)}\ \{\texttt{return x.disp\_m(this);}\}$

Note that the algorithm needs both the original *CT* and a new class table $\overline{CT}$, which contains the translated classes; the former is used to drive the translation, in particular for the lookup functions and for retrieving the original method definitions, while the latter is updated at every step (we use the assignment operator $:=$ to update the class table $\overline{CT}$). Thus, we start by initializing $\overline{CT}$ with a copy of the original class table *CT* (this way we also copy all the class fields and constructor definitions that will not be changed during the translation). Then, for every class in *CT* we perform the three steps that act on method names and method definitions. Note that the renaming (step 2(a)i) is always performed starting from $\overline{CT}$, thus incrementally renaming method names one by one. Step 2(a)iiA generates the `disp_` methods in target classes; the branches to be generated are collected starting from the original client class in *CT*, using the *mtype* function that implements the copy semantics of inheritance; this is crucial to make the translation algorithm work correctly. Step 2(a)iiB adds the *entry point* branches, that have the same name of the multi methods in the original program, that forward to the `disp_` methods. Note that in Steps 2(a)iiA and 2(a)iiB it is crucial to start from the original class, otherwise, the algorithm would treat also the added or renamed methods in successive steps.

Finally, we observe that the translated program $(\overline{CT}, \texttt{e})$ differs from the original one $(CT, \texttt{e})$ only with respect to method definitions, affecting neither the body of the original methods (i.e., the translation never acts on method bodies) nor the main expression `e`. The translation algorithm here defined preserves both type safety and semantics of the original programs, then it implements dynamic overloading in a type safe and correct way.

3 Conclusions and Related Work

Some object-oriented languages and calculi have been proposed to study multi-methods and dynamic overloading; we refer to [2] for an extensive discussion of these approaches. Here we list only a few recent works which are more related to the issues of the present paper.

Parasitic methods [4] are a linguistic extension of Java with asymmetric multi-methods, with the main goal of retaining modularity. The approach does not use copy semantics, the selection of the most specialized method relies on `instanceof` checks and consequent type casts (thus it does not perform constantly as in our solution, but essentially linearly on the number of branches), the dispatching semantics is complicated by the use of textual order of method declarations.

MultiJava [6] is a backward-compatible extension to Java supporting multi-methods and open classes. New methods in a class `C` can be added by defining external *method*

families in a different compilation unit w.r.t. to the one containing `C` definition. The drawback of this approach is that extending, or modifying an existing method, can only be done by explicitly subclassing all affected variants and overriding the corresponding branches. This complicates the extensibility and can lead to an inconsistent distribution of code. *MultiJava* is directly compiled into Java bytecode. *Relaxed MultiJava* [9] increases flexibility of MultiJava w.r.t. overloading; however, it is obtained by allowing a function call to be ambiguous; these ambiguities are caught at class load time.

In [10] C++ is extended with open multi-methods and symmetric dispatch. Differently from our approach, open multi-methods are external to classes, as the external added methods of MultiJava.

The translation algorithm presented in this paper can be extended in two directions. Firstly, we could consider methods with more than one parameter. In this case, the translation gets more complicated, and it is the subject of an ongoing work. The second extension, which has a stronger practical impact, consists in handling multiple inheritance. Multiple inheritance introduces subtle ambiguity problems that have always been the drawback of introducing dynamic overloading in mainstream languages. From the point of view of the language, in [2] we showed how the type-checking can rule out all possible ambiguities due to multiple inheritance; then, we have only to extend the translation algorithm.

References

1. D. Ancona, S. Drossopoulou, and E. Zucca. Overloading and Inheritance. In *FOOL 8*, 2001.
2. L. Bettini, S. Capecchi, and B. Venneri. Featherweight Java with Dynamic and Static Overloading. *Science of Computer Programming*. To appear. Available at `http://www.dsi.unifi.it/~bettini/multifj.pdf`.
3. L. Bettini, S. Capecchi, and B. Venneri. Double Dispatch in C++. *Software: Practice and Experience*, 36(6):581–613, 2006.
4. J. Boyland and G. Castagna. Parasitic Methods: Implementation of Multi-Methods for Java. In *Proc. of OOPSLA*, pages 66–76. ACM, 1997.
5. G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, 1997.
6. C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Prog. Lang. Syst.*, 28(3), May 2006.
7. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
8. D. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proc. OOPSLA*, pages 347–349. ACM Press, 1986.
9. T. Millstein, M. Reay, and C. Chambers. Relaxed multijava: balancing extensibility and modular typechecking. *SIGPLAN Not.*, 38(11):224–240, 2003.
10. P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Open multi-methods for C++. In *GPCE*, pages 123–134. ACM, 2007.