Featherweight Java with Multi-Methods *

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

[1]Dipartimento di Sistemi ed Informatica, Università di Firenze, {bettini,venneri}@dsi.unifi.it
[2]Dipartimento di Matematica e Informatica, Università di Catania, capecchi@dmi.unict.it

ABSTRACT

Multi-methods (collections of overloaded methods associated to the same message, whose selection takes place dynamically instead of statically as in standard overloading) are a useful mechanism since they unleash 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., Java, do not provide it, resorting to only static overloading.

In this paper we propose an extension, we call FMJ (Featherweight Multi Java), of Featherweight Java with encapsulated multi-methods thus providing dynamic overloading. The extension is conservative and type safe: both "message-not-understood" and "message-ambiguous" are statically ruled out. Our core language can be used as the formal basis for an actual implementation of dynamic overloading in Java-like languages.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages, Theory

Keywords: Language extensions, Featherweight Java, Multi-methods, Dynamic Overloading.

1. INTRODUCTION

Method and function overloading is typically a static mechanism in mainstream languages, such as C++ and Java, in that the most appropriate implementation is selected statically by the compiler according to the static type of the arguments (static overloading). With this respect, in [15], this kind of polymorphism is called "ad-hoc".

It is generally recognized that static overloading is misleading in Java-like languages [34, 9]. Indeed, overloading interaction with inheritance differs among different languages, thus originating a confusing situation in programming in practice: it might become difficult to figure out which method implementation will be executed at run time when different signatures match with the type of the actual parameters (by subtyping). Instead dynamic overloading (run-time selection mechanism based on the dynamic types of the receiver and the arguments) does not suffer from this problem. Namely, our proposal guarantees that the selection of an overloaded method is as specific as possible, also with respect to the parameter types, by using the standard method lookup starting from the dynamic class of the receiver.

The need of adopting dynamic overloading arises in many situations. A typical scenario is when there are two separate class hierarchies and the classes of a hierarchy have to operate on instances of classes of the other hierarchy according to their dynamic types. Quoting from [2], "*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.*" This is quite a recurrent situation in object-oriented design, since separation of responsibilities enhances class decoupling and thus re-usability.

So we end up having a class hierarchy for a specific operation with a superclass, say, `Operation`, that is separate from the hierarchy of the elements with a superclass, say, `Elem`. This separation easily permits changing 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 [26]). For instance, we may structure the class `Operation` as follows, where `ElemB` is a subclass of `ElemA` and `ElemC` is a subclass of `ElemB`.

```
class Operation {
    public void op(ElemA e) { ... }
    public void op(ElemB e) { ... }
    public void op(ElemC e) { ... }
    ...
}
```

in order to perform a specific operation according to the actual type of the element. However, such methods `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 `e`. So, the following code:

```
Operation o = new Operation();
ElemA e = new ElemB();
...
o.op(e);
```

would not dynamically select the most appropriate method according to the actual (run-time) type of `e`.

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

PPPJ 2007, September 5–7, 2007, Lisboa, Portugal.

Furthermore, multi-methods provide safe *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 [33, 6]. Its most expressive application appears with *binary methods* [13], 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 runtime according to the actual type of the argument and the receiver.

Covariant specialization, overriding, overloading and binary methods have often participated together in making the relation between overloading and subtyping more complex than it is due. With this respect [16] provides an insightful study of these problems and concludes that once the above concepts are given the correct interpretation, everything turns out to be clear and, most importantly, safe. However, the overloading needs to be dynamic in order to have the full flexibility without problems. Also for this reason, we believe that problems, skepticism and fear in using static overloading should disappear when one can exploit dynamic overloading.

The concept of dynamic overloading is interchangeable with the one of *multi-methods* [22, 36, 17]. A multi-method can be seen as a collection of overloaded methods, called *branches*, associated to the same message, but the selection takes place dynamically according to run-time types of the receiver and of the arguments. Though multi-methods are widely studied in the literature and supported in languages such as *CLOS* [11], *Dylan* [39] and *BeCecil* [20], they have not been added to mainstream programming languages such as Java, C++ and C#. 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 overcome lacks of the language [31].

The problems of solutions based on run-time checks is that, besides of requiring manual programming, they are error prone; e.g., in the following code, that tries to manually simulate dynamic overloading, the case for `ElemC` should have come before the case for `ElemB`:

```
public void op(ElemA e) {
    if (e instanceof ElemB)
        op((ElemB)e);
    else if (e instanceof ElemC)
        op((ElemC)e);
    else {
        // code for case ElemA
    }
}
```

As it is, the case for `ElemC` will never be executed.

Another problem is that, when combining two object-oriented mechanisms, crucial issues arise that lead the designers of the language to choose directions that simplify the implementation of the language itself at the cost of sacrificing its flexibility. This typically happens when combining static overloading and inheritance [9]. With this respect, [4] tackles this issue and shows how the design choices of languages such as Java (earlier than version 1.4) are counter intuitive and too restrictive. In particular, [4] points out that semantics of overloading and inheritance is rather "clean" if it is interpreted through a *copy semantics of inheritance*, whereby all the inherited overloaded methods are intended to be directly copied into the subclass (apart for those explicitly redefined by the subclass itself). This avoids a too restrictive overloading resolution policy that leads to "strange" compilation errors or unexpected behaviors.

In this paper we propose an extension of Featherweight Java (FJ) [29, 37] with multi-methods thus providing dynamic overloading. The extension is conservative and type safe (both "message-not-understood" and "message-ambiguous" are statically ruled out) and can be used as the formal basis for an actual implementation of dynamic overloading in Java or in another object-oriented language. Featherweight Java was proposed as a minimal core calculus for modelling Java's type system and it is useful for studying extensions of Java-like languages (see, e.g., [29, 28]).

We believe our language extension is interesting since, at the best of our knowledge, this is the first attempt to extend FJ with multi-methods (although some proposals for multi-methods in Java are already present, see, e.g., [12, 21]).

The multi-methods we are considering are *encapsulated* multi-methods, i.e., they are actual methods of classes and not external functions (as, e.g., in [39]), and the branch selection is *symmetric*: during the dynamic overloading selection the receiver type of the method invocation has no precedence over the argument types (differently from the encapsulated multi-methods of [13] and [12]).

Symmetry of branch selection is tightly related to copy semantics: if the receiver of a method invocation had the precedence over the parameters, it would mean that we search for the best matching branch only on the class of the receiver (or the first superclass that defines some branch in case the class of the receiver does not define branches), i.e., it does not inspect also the superclasses' branches (which could provide a more specialized version). Thus, copy semantics would not be implemented.

For instance, consider a subclass of the above class `Operation`, where we redefine the branch for `ElemB` and we add a specialization to the multi-method with the branch for `ElemE` (where `ElemE` is a subclass of `ElemD`)[1]:

```
class ExtendedOperation extends Operation {
    public void op(ElemB e) { ... } // redefinition
    public void op(ElemE e) { ... } // specialization
    ...
};
```

If we had no copy semantics, then the following method invocation

```
Operation o = new ExtendedOperation();
ElemA e = new ElemC();
...
o.op(e);
```

would select the branch for `ElemB` in `ExtendedOperation`, while we should have selected the branch for `ElemC` in `Operation` which would be more specific.

With copy semantics instead, and thus symmetric selection, we get the desired behavior. This way, the programmer of the subclasses is not required to redefine all branches just to avoid hiding the branches of the superclasses (this would also force the programmer to be aware of too many details of the superclasses).

Furthermore, the overloading resolution policy in Java has followed copy semantics since version 1.4. We further discuss copy semantics issues in Section 4 where we also propose some alternative versions of FMJ without copy semantics.

[1]Throughout the paper, in the examples, we will use the full Java syntax instead of the smaller FMJ one (e.g., we will use imperative features). These examples could be written also in FMJ but they would only be more verbose.

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

Figure 1: FMJ syntax

The paper is organized as follows: in Section 2 we present the syntax, typing and operational semantics of our extension of FJ, FMJ. In section 3 we present the main properties of our language. In Section 4 we consider some variations of FMJ without copy semantics. Section 5 resume some works on multi-methods in the literature. Finally, Section 6 concludes the paper with some evaluations and proposes some future directions.

2. FEATHERWEIGHT MULTI JAVA

In this section we present syntax, typing and operational semantics of our proposal, the core language FMJ (*Featherweight Multi Java*), which is an extension of FJ (*Featherweight Java*) to multi-methods. FJ [29, 37] is a lightweight version of Java, which focuses on a few basic features: mutually recursive class definitions, inheritance, object creation, method invocation, method recursion through `this`, subtyping and field access[2]. Thus, the minimal syntax, typing and semantics make the type safety proof simple and compact, in such a way that FJ is a handy tool for studying the consequences of extensions and variations with respect to Java ("FJ's main application is modeling extensions of Java", [37], pag. 248). Although we assume the reader is familiar with FJ, we will give a brief introduction to FJ and then we will focus on the novel aspects of FMJ w.r.t. FJ. The full FMJ definition (typing and semantics) is presented in Figure 5.

The abstract syntax of FMJ constructs is given in Figure 1 and it is just the same of FJ. The programmer, however, is allowed to specify more than one method with the same name and different signatures (multi-methods), and this difference w.r.t. FJ will be evident in the typing and in the semantics. The metavariables `B`, `C`, `D` and `E` range over class names; `f` and `g` range over attribute names; `x` ranges over method parameter names; `e` and `d` range over expressions and `v` and `u` range over values.

As in FJ, we will use the overline notation for possibly empty sequences (e.g., "$\overline{\texttt{e}}$" is a shorthand for a possibly empty sequence "$\texttt{e}_1,\ldots,\texttt{e}_n$"). We abbreviate pair of sequences in similar way, e.g., $\overline{\texttt{C}}$ $\overline{\texttt{f}}$ stands for $\texttt{C}_1$ $\texttt{f}_1,\ldots,\texttt{C}_n$ $\texttt{f}_n$. The empty sequence is denoted by $\bullet$.

Following FJ, we assume that the set of variables includes the special variable `this` (implicitly bound in any method declaration), which cannot be used as the name of a method's formal parameter (this restriction is imposed by the typing rules). Note that since we treat `this` in method bodies as an ordinary variable, no special syntax for it is required.

A class declaration class C extends D {$\overline{\texttt{C}}$ $\overline{\texttt{f}}$; K; $\overline{\texttt{M}}$} consists of its name `C`, its superclass `D` (which must always be specified, even if it is `Object`), a list of field names $\overline{\texttt{C}}$ $\overline{\texttt{f}}$ with their types, the constructor `K`, and a list of method definitions $\overline{\texttt{M}}$. The instance variables of `C` are added to the ones declared by `D` and its superclasses and are assumed to have distinct names. The constructor declaration shows how to initialize all these fields with the received values. A method definition `M` specifies the name, the signature and the body of a method; a body is a single `return` statement since FJ is a functional core of Java. The same method name can occur with different signatures in $\overline{\texttt{M}}$. Differently, in FJ all method names in the same class are assumed to be distinct; moreover, typing rules check that if a method with the same name is declared in the superclass, then it must have the same signature in all subclasses. The key feature of FMJ consists in interpreting any definition of a method `m` in a class `C` as the definition of a new branch of the multi-method `m`; 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*); thus `m` is a multi-method that is associated to different branch definitions for different signatures. Clearly, the notion of standard method is subsumed by our definition as a multi-method with only one branch. In the following, we will write $\texttt{m} \in \overline{\texttt{M}}$ ($\texttt{m} \notin \overline{\texttt{M}}$) to mean that at least one branch definition (no branch definition) of `m` belongs to $\overline{\texttt{M}}$. Finally, values, denoted by `v` and `u`, are fully evaluated object creation terms new C($\overline{\texttt{v}}$) (i.e., expressions that are passed to the constructor are values too) and will be the results of completely evaluated well-typed expressions.

As in FJ, a class table *CT* is a mapping from class names to class declarations. Then a program is a pair $(CT, \texttt{e})$ of a class table (containing all the class definitions of the program) and an expression `e` (the program's main entry point). The class `Object` has no members and its declaration does not appear in *CT*. We assume that *CT* satisfies some usual sanity conditions: (*i*) $CT(\texttt{C}) = \texttt{class C} \ldots$ for every $\texttt{C} \in dom(CT)$ (*ii*) for every class name `C` (except `Object`) appearing anywhere in *CT*, we have $\texttt{C} \in dom(CT)$; (*iii*) there are no cycles in the transitive closure of the `extends` relation. Thus, in the following, instead of writing $CT(\texttt{C}) = \texttt{class} \ \ldots$ we will simply write class C ....

The subtyping relation $<:$ on classes (types) is induced by the standard subclass relation: $\texttt{C} <: \texttt{D}$ holds if and only if either class C extends D or class C extends $\texttt{D}_1$ and $\texttt{D}_1 <: \texttt{D}$. Thus, a subtyping relation $<:$ is defined for any class table *CT*.

In the rest of the section, in order to focus the attention on our extension of FJ, we will discuss only the parts of typing and semantics that we introduced or that we modified w.r.t. FJ. However, for completeness, the full FMJ definition (typing and semantics) is presented in Figure 5.

2.1 Typing

The types of FMJ 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:

$$\{\overline{\texttt{C}}_1 \to \texttt{C}_1, \ldots, \overline{\texttt{C}}_n \to \texttt{C}_n\}$$

We will write multi-types in a compact form, by extending the sequence notation to multi-types:

$$\{\overline{\overline{\texttt{C}} \to \texttt{C}}\}$$

Multi-types can be constrained by three crucial consistency conditions (adapted from [17, 18]), which are checked statically, presented in the following definition of well-formedness.

DEFINITION 2.1 (WELL-FORMEDNESS OF MULTI-TYPES).

A multi-type $\{\overline{\overline{\texttt{B}} \to \texttt{B}}\}$ *is* well-formed*, denoted by*

[2]FJ also includes up and down casts; however, since these features are completely orthogonal to our extension to multi-methods, they are omitted in FMJ.

$$wellformed(\{\overline{\overline{B} \to B}\})$$

if $\forall\ (\overline{B}_i \to B_i), (\overline{B}_j \to B_j) \in \{\overline{\overline{B} \to B}\}$ *the following conditions are verified:*

1. $\overline{B}_i \neq \overline{B}_j$
2. $\overline{B}_i <: \overline{B}_j \Rightarrow B_i <: B_j$
3. *For all* $\overline{E}$ *maximal types in* $LB(\overline{B}_i, \overline{B}_j)$ *there exists* $\overline{E} \to E \in \{\overline{\overline{B} \to B}\}$ *for some* E.

Where $LB()$ is the standard lower bound set, i.e., $LB(\overline{B}_i, \overline{B}_j) = \{\overline{C} \mid \overline{C} <: \overline{B}_i \wedge \overline{C} <: \overline{B}_j\}$. The first condition is quite natural, in that it 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 third condition is crucial to rule out statically any possible ambiguities. Notice that, if no ambiguity is present at compile time, no ambiguity is guaranteed to be present also at run-time.

We extend the sequence notation also to multi-method definitions:

$$\overline{C\ m\ (\overline{C}\ \overline{x})\{\texttt{return}\ e;\}}$$

represents a sequence of method definitions, each with the same name m but with different signatures (and possibly different bodies).

$$C_1\ m\ (\overline{C}_1\ \overline{x})\{\texttt{return}\ e_1;\}\ \ldots\ C_n\ m\ (\overline{C}_n\ \overline{x})\{\texttt{return}\ e_n;\}$$

The multi-type of the above multi-method will be denoted by $\{\overline{C} \to C\}$.

To lighten the notation, in the following, we will assume a fixed class table *CT* and then $<:$ is the subtype relation induced by *CT*. We will write $\overline{C} <: \overline{D}$ as a shorthand for $C_1 <: D_1 \wedge \ldots \wedge C_n <: D_n$.

As in FJ, we define some auxiliary functions to look up fields and method branches from *CT*; these functions are used in the typing rules and in the operational semantics.

The look up function *fields*(C) returns the sequence of the field names, together with the corresponding types, for all the fields declared in C and in its superclasses.

The new lookup functions concerning multi-methods are in Figure 2. The *mtype*(m, C) lookup function (where m is the method name we're looking for, and C is the class where we are performing the lookup) differs from the one of FJ in that it does not select a method's signature, but it represents 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. We introduce an additional lookup function, *mtypesel*(m, C, $\overline{C}$), that selects the signature of the branch of the multi-method m in the class C, that matches best the specified argument types $\overline{C}$ (function *minsel*). This consists in selecting, among the branches whose parameter types are supertypes of $\overline{C}$, the one that is the most specific. The function *minsel* is defined formally in Definition 2.3

DEFINITION 2.2. *Given a set of arrow types* $\{\overline{\overline{B} \to B}\}$*, we denote by* $MIN(\{\overline{\overline{B} \to B}\})$ *the set of minimal arrow types computed w.r.t. to the input types only, i.e.,*

$$MIN(\{\overline{\overline{B} \to B}\}) \stackrel{def}{=} \{\overline{B}_i \to B_i \in \{\overline{\overline{B} \to B}\} \mid$$

$$\forall (\overline{B}_j \to B_j) \in \{\overline{\overline{B} \to B}\}\ s.t.\ \overline{B}_i \neq \overline{B}_j$$

$$\overline{B}_j \not<: \overline{B}_i\}$$

Method type lookup

$$\frac{\texttt{class C extends D}\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} \qquad \overline{B\ m\ (\overline{B}\ \overline{x})\{\texttt{return e;}\}} \in \overline{M}}{mtype(m, C) = \{\overline{\overline{B} \to B}\} \cup mtype(m, D)}$$

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

$$\frac{mtype(m, C) = \{\overline{\overline{B} \to B}\} \qquad \overline{D} \to D = minsel(\overline{C}, \{\overline{\overline{B} \to B}\})}{mtypesel(m, C, \overline{C}) = \overline{D} \to D}$$

Method body lookup

$$\frac{\texttt{class C extends D}\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} \qquad B\ m\ (\overline{B}\ \overline{x})\{\texttt{return e;}\} \in \overline{M}}{mbody(m, C, \overline{B}) = \overline{x}.e}$$

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

$$\frac{mtypesel(m, C, \overline{C}) = \overline{D} \to D \qquad mbody(m, C, \overline{D}) = (\overline{x}, e)}{mbodysel(m, C, \overline{C}) = (\overline{x}, e)}$$

Figure 2: Lookup functions

Notice that this set is computed only on the parameter types, i.e., the return type is not considered (in fact, the return type does not participate in overloading selection).

DEFINITION 2.3 (MOST SPECIALIZED SELECTION). *Given some parameter types* $\overline{C}$ *and a multi-type* $\{\overline{\overline{B} \to B}\}$*, then* $minsel(\overline{C}, \{\overline{\overline{B} \to B}\}) \stackrel{def}{=} \overline{B}_i \to B_i$ *if and only if*

$$MIN(\{\overline{B}_j \to B_j \in \{\overline{\overline{B} \to B}\} \mid \overline{C} <: \overline{B}_j\}) = \{\overline{B}_i \to B_i\}$$

i.e., $MIN()$ *returns a singleton.*

Notice that, if a multi-method $\{\overline{\overline{B} \to B}\}$ is well-formed (Definition 2.1), then $minsel(\overline{C}, \{\overline{\overline{B} \to B}\})$ is defined if and only if $\overline{C} <: \overline{B}_i$ for some $\overline{B}_i \to B_i \in \{\overline{\overline{B} \to B}\}$; in fact, the condition 3 of well-formedness is the weakest condition for ensuring that the subset of branches matching with $\overline{C}$ is either empty or has a least element (see the formal proof in [18], p. 119), thus there can be no ambiguity. Hence, once typing has successfully checked all the classes of a program (i.e., all multi-methods are well-formed), then every method invocation in the program is safe in that not only a method definition exists for every method call, but it is also unique. Furthermore, well-formedness guarantees that during the evaluation, well-typedness is preserved, while types of both the receiver and the arguments possibly decrease (see Section 3).

Accordingly, *mbody*(m, C, $\overline{C}$) selects the body of the branch of the multi-method m in the class C that has the parameter types $\overline{C}$ (since branches are inherited, this function may have to inspect the superclass of C in case that body is not defined in C). *mbody* returns a pair where the first element is the parameter sequence, and the second one is the actual body of the selected branch. Differently from FJ, where the parameter types were not needed, since only one signature for a method can exist in a class hierarchy, here we also need to

Expression typing

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \mathit{mtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}) = \overline{\mathtt{B}} \to \mathtt{B}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{B}} \quad \text{(T-INVK)}$$

Method typing

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e} : \mathtt{E} \qquad \mathtt{E} <: \mathtt{B}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\}\ \mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}\ \mathtt{OK\ IN\ C}} \quad \text{(T-METHOD)}$$

Class typing

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this.}\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \{\overline{\mathtt{B}} \to \mathtt{B}\} \wedge \mathit{wellformed}(\{\overline{\mathtt{B}} \to \mathtt{B}\}) \text{ for all } \mathtt{m} \in \overline{\mathtt{M}}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \quad \text{(T-CLASS)}$$

Figure 3: Typing rules

$$\frac{\mathit{mbodysel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{D}}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{new\ D}(\overline{\mathtt{u}})}) \longrightarrow [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{new\ D}(\overline{\mathtt{u}})}, \mathtt{this} \leftarrow \mathtt{new\ C}(\overline{\mathtt{v}})]\mathtt{e}_0} \quad \text{(R-INVK)}$$

Figure 4: Reduction rule for multi-method selection

specify the parameter types in order to perform the body selection. We introduce the additional lookup function, $\mathit{mbodysel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}})$ that searches for the signature of the branch that matches best the argument types $\overline{\mathtt{C}}$, by using the above mentioned *mtypesel* lookup function, and then selects the body of the obtained signature, by using *mbody*.

As usual, a type judgment of the form $\Gamma \vdash \mathtt{e} : \mathtt{C}$ states that "e has type C in the type environment Γ". A type environment is a finite mapping from variables (including this) to types, written $\overline{\mathtt{x}} : \overline{\mathtt{C}}$. Again, we use the sequence notation for abbreviating $\Gamma \vdash \mathtt{e}_1 : \mathtt{C}_1, \ldots, \Gamma \vdash \mathtt{e}_n : \mathtt{C}_n$ to $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$.

Typing rules that are relevant for multi-methods are presented in Figure 3. Concerning expressions, the only rule that changes w.r.t. FJ is (T-INVK): method invocation concerns dynamic overloading, i.e., multi-method selection; in fact we use the *mtypesel* function, in order to obtain the return type of the selected branch.

The rule for typing a branch, (T-METHOD), differs from the original (T-METHOD) of FJ in that we do not check for signature redefinition (the *override* predicate of [37]). In FJ we need to check that if a subclass overrides a method then the signature is not modified, i.e., that we are overriding a method definition by preserving the signature. In our case, any new signature is legal: if it is the same as the signature of a branch already defined, then this redefinition overrides the implementation of that branch, otherwise it is intended as the definition of a new branch for that method.

Finally, (T-CLASS) checks also that any multi-method defined in the class is well-formed (the function *wellformed*, Definition 2.1).

2.2 Operational semantics

The operational semantics is defined by the reduction relation $\mathtt{e} \longrightarrow \mathtt{e}'$, read "e reduces to e′ in one step". The standard reflexive and transitive closure of $\longrightarrow$ defines the reduction relation in many steps. We adopt a deterministic call-by-value semantics, analogous to the call-by-value strategy of FJ presented in [37]. The congruence rules (Figure 5) formalize how operators (method invocation, object creation and field selection) are reduced only when all their subexpressions are reduced to values (call-by-value).

The expression $[\overline{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \leftarrow \mathtt{u}]\mathtt{e}$ denotes the expression obtained from e by replacing $\mathtt{x}_1$ with $\mathtt{v}_1$, ..., $\mathtt{x}_n$ with $\mathtt{v}_n$ and this with u.

We focus on the only rule that needs to be changed w.r.t. FJ, i.e., the method selection rule (R-INVK) (Figure 4). This rule employs dynamic overloading for selecting the most specific body (branch) of the invoked method, with respect to the actual types of both the receiver and the arguments. The semantics of method invocation is type driven: the function *mtypesel* is employed to obtain the signature of the most specialized branch, then the function *mbodysel* searches for the body that is associated to this signature starting from the actual type of the receiver object. Thus, the selected method body is not only the most specialized w.r.t. the argument types but also the most redefined version associated to that signature. This way, we model standard method overriding inside our mechanism of dynamic overloading.

3. PROPERTIES

In this section we address the issue of type safety for FMJ. In the proofs, we explicitly deal with crucial points involving typing and semantics rules for multi-methods, while we omit the parts of proofs that are unchanged with respect to the corresponding proofs in FJ. Indeed, we fully exploit one of the most appealing features of FJ, i.e., when FJ is employed for a language extension, properties concerning those language parts that are not changed/extended can be re-used together with their proofs.

Some preliminary lemmas are useful to simplify the proof of type preservation.

LEMMA 3.1. *Let* $\mathtt{C}$ *and* $\mathtt{C}_1$ *be such that* $CT(\mathtt{C})$ *and* $CT(\mathtt{C}_1)$ *are well-typed (*OK*). If* $\mathit{mtypesel}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}}) = \overline{\mathtt{B}} \to \mathtt{B}$ *then, for any* $\mathtt{C}_1$ *and* $\overline{\mathtt{C}}_1$ *such that* $\mathtt{C}_1 <: \mathtt{C}$ *and* $\overline{\mathtt{C}}_1 <: \overline{\mathtt{C}}$*, we have that* $\mathit{mtypesel}(\mathtt{m}, \mathtt{C}_1, \overline{\mathtt{C}}_1) = \overline{\mathtt{E}} \to \mathtt{E}$ *where* $\overline{\mathtt{E}} <: \overline{\mathtt{B}}$ *and* $\mathtt{E} <: \mathtt{B}$.

PROOF. By induction on the derivation of $\mathtt{C}_1 <: \mathtt{C}$. Notice that

- $\overline{\mathtt{B}} \to \mathtt{B} \in \mathit{mtype}(\mathtt{m}, \mathtt{C}_1)$ since $\mathtt{C}_1 <: \mathtt{C}$ and then by definition of *mtype*.
- $\overline{\mathtt{C}}_1 <: \overline{\mathtt{C}}$ and $\overline{\mathtt{C}} <: \overline{\mathtt{B}}$ (by definition of *mtypesel*) imply $\overline{\mathtt{C}}_1 <: \overline{\mathtt{B}}$

Then, by definition of $\mathit{mtypesel}(\mathtt{m}, \mathtt{C}_1, \overline{\mathtt{C}}_1)$ we have that $\overline{\mathtt{E}} \to \mathtt{E}$ is such that $\overline{\mathtt{E}} <: \overline{\mathtt{B}}$. Finally, since $\overline{\mathtt{E}} \to \mathtt{E} \in \mathit{mtype}(\mathtt{m}, \mathtt{C}_1)$ and $\overline{\mathtt{B}} \to \mathtt{B} \in \mathit{mtype}(\mathtt{m}, \mathtt{C}_1)$, by well-formedness of multi-types (Definition 2.1) we have $\mathtt{E} <: \mathtt{B}$. □

The previous Lemma is a crucial property to prove the Substitution Lemma. Notice that if m is a standard method, whose redefinition in subclasses preserves the signature (standard method overriding), then, our definitions of *mtypesel* and the proof of the following Substitution Lemma collapse to the proof of the analogous Lemma given for FJ. Indeed, our language extension is conservative.

LEMMA 3.2 (SUBSTITUTION LEMMA). *If* $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e} : \mathtt{D}$ *and* $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}}$ *where* $\overline{\mathtt{E}} <: \overline{\mathtt{B}}$*, then* $\Gamma \vdash [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{e}}]\mathtt{e} : \mathtt{C}$ *for some* $\mathtt{C} <: \mathtt{D}$.

PROOF. By induction on a derivation of $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e} : \mathtt{D}$. The only interesting case is when the last applied rule is (R-INVK) which follows from Lemma 3.1. □

THEOREM 3.3 (TYPE PRESERVATION). *If* $\Gamma \vdash \mathtt{e} : \mathtt{E}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$ *then* $\Gamma \vdash \mathtt{e}' : \mathtt{E}'$ *for some* $\mathtt{E}' <: \mathtt{E}$.

PROOF. By induction on a derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$. The only crucial case is when the last applied rule is (R-INVK):

Syntax

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

Subtyping

$$\text{C} <: \text{C} \qquad \frac{\text{C} <: \text{D} \qquad \text{D} <: \text{E}}{\text{C} <: \text{E}} \qquad \frac{\text{class C extends D } \{\ldots\}}{\text{C} <: \text{D}}$$

Field lookup

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\text{class C extends D } \{\overline{\text{C f}}; \text{K}; \overline{\text{M}}\} \qquad \mathit{fields}(\text{D}) = \overline{\text{D g}}}{\mathit{fields}(\text{C}) = \overline{\text{D g}}, \overline{\text{C f}}}$$

Method type lookup

$$\frac{\text{class C extends D } \{\overline{\text{C f}}; \text{K}; \overline{\text{M}}\} \qquad \overline{\text{B m } (\overline{\text{B x}})\{\text{return e;}\}} \in \overline{\text{M}}}{\mathit{mtype}(\text{m}, \text{C}) = \{\overline{\overline{\text{B}} \to \text{B}}\} \cup \mathit{mtype}(\text{m}, \text{D})}$$

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

$$\frac{\mathit{mtype}(\text{m}, \text{C}) = \{\overline{\overline{\text{B}} \to \text{B}}\} \qquad \overline{\text{D}} \to \text{D} = \mathit{minsel}(\overline{\text{C}}, \{\overline{\overline{\text{B}} \to \text{B}}\})}{\mathit{mtypesel}(\text{m}, \text{C}, \overline{\text{C}}) = \overline{\text{D}} \to \text{D}}$$

Method body lookup

$$\frac{\text{class C extends D } \{\overline{\text{C f}}; \text{K}; \overline{\text{M}}\} \qquad \text{B m } (\overline{\text{B x}})\{\text{return e;}\} \in \overline{\text{M}}}{\mathit{mbody}(\text{m}, \text{C}, \overline{\text{B}}) = \overline{\text{x}}.\text{e}}$$

$$\frac{\text{class C extends D } \{\overline{\text{C f}}; \text{K}; \overline{\text{M}}\} \qquad \text{m} \notin \overline{\text{M}}}{\mathit{mbody}(\text{m}, \text{C}, \overline{\text{B}}) = \mathit{mbody}(\text{m}, \text{D}, \overline{\text{B}})}$$

$$\frac{\mathit{mtypesel}(\text{m}, \text{C}, \overline{\text{C}}) = \overline{\text{D}} \to \text{D} \qquad \mathit{mbody}(\text{m}, \text{C}, \overline{\text{D}}) = (\overline{\text{x}}, \text{e})}{\mathit{mbodysel}(\text{m}, \text{C}, \overline{\text{C}}) = (\overline{\text{x}}, \text{e})}$$

Expression typing

$$\Gamma \vdash \text{x} : \Gamma(\text{x}) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \text{e}_0 : \text{C}_0 \qquad \mathit{fields}(\text{C}_0) = \overline{\text{C f}}}{\Gamma \vdash \text{e}_0.\text{f}_i : \text{C}_i} \qquad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash \text{e} : \text{C} \qquad \Gamma \vdash \overline{\text{e}} : \overline{\text{C}} \qquad \mathit{mtypesel}(\text{m}, \text{C}, \overline{\text{C}}) = \overline{\text{B}} \to \text{B}}{\Gamma \vdash \text{e.m}(\overline{\text{e}}) : \text{B}} \qquad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(\text{C}) = \overline{\text{D f}} \qquad \Gamma \vdash \overline{\text{e}} : \overline{\text{C}} \qquad \overline{\text{C}} <: \overline{\text{D}}}{\Gamma \vdash \text{new C}(\overline{\text{e}}) : \text{C}} \qquad \text{(T-NEW)}$$

Method and Class typing

$$\frac{\overline{\text{x}} : \overline{\text{B}}, \text{this} : \text{C} \vdash \text{e} : \text{E} \qquad \text{E} <: \text{B} \qquad \text{class C extends D } \{\overline{\text{C f}}; \text{K}; \overline{\text{M}}\}}{\text{B m } (\overline{\text{B x}})\{\text{return e;}\} \text{ OK IN C}} \qquad \text{(T-METHOD)}$$

$$\frac{\begin{array}{c}\text{K} = \text{C}(\overline{\text{D g}}, \overline{\text{C f}})\{\text{super}(\overline{\text{g}}); \text{this.}\overline{\text{f}} = \overline{\text{f}};\} \\ \mathit{fields}(\text{D}) = \overline{\text{D g}} \qquad \overline{\text{M}} \text{ OK IN C} \\ \mathit{mtype}(\text{m}, \text{C}) = \{\overline{\text{B}} \to \text{B}\} \ \wedge \ \mathit{wellformed}(\{\overline{\text{B}} \to \text{B}\}) \text{ for all } \text{m} \in \overline{\text{M}}\end{array}}{\text{class C extends D } \{\overline{\text{C f}}; \text{K}; \overline{\text{M}}\} \text{ OK}} \qquad \text{(T-CLASS)}$$

Reduction

$$\frac{\mathit{fields}(\text{C}) = \overline{\text{C f}}}{(\text{new C}(\overline{\text{v}})).\text{f}_i \longrightarrow \text{v}_i} \qquad \text{(R-FIELD)}$$

$$\frac{\mathit{mbodysel}(\text{m}, \text{C}, \overline{\text{D}}) = (\overline{\text{x}}, \text{e}_0)}{\text{new C}(\overline{\text{v}}).\text{m}(\overline{\text{new D}(\overline{\text{u}})}) \longrightarrow [\overline{\text{x}} \leftarrow \overline{\text{new D}(\overline{\text{u}})}, \text{this} \leftarrow \text{new C}(\overline{\text{v}})]\text{e}_0} \qquad \text{(R-INVK)}$$

Congruence rules

$$\frac{\text{e} \longrightarrow \text{e}'}{\text{e.f} \longrightarrow \text{e}'.\text{f}}$$

$$\frac{\text{e} \longrightarrow \text{e}'}{\text{e.m}(\overline{\text{e}}) \longrightarrow \text{e}'.\text{m}(\overline{\text{e}})}$$

$$\frac{\text{e}_i \longrightarrow \text{e}_i'}{\text{v}_0.\text{m}(\overline{\text{v}}, \text{e}_i, \overline{\text{e}}) \longrightarrow \text{v}_0.\text{m}(\overline{\text{v}}, \text{e}_i', \overline{\text{e}})}$$

$$\frac{\text{e}_i \longrightarrow \text{e}_i'}{\text{new C}(\overline{\text{v}}, \text{e}_i, \overline{\text{e}}) \longrightarrow \text{new C}(\overline{\text{v}}, \text{e}_i', \overline{\text{e}})}$$

Figure 5: Definition of FMJ

$$\frac{\mathit{mbodysel}(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = (\overline{\mathtt{x}},\mathtt{e}_0)}{\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{new\ D}(\overline{\mathtt{u}})}) \longrightarrow [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{new\ D}(\overline{\mathtt{u}})}, \mathtt{this} \leftarrow \mathtt{new\ C}(\overline{\mathtt{v}})]\mathtt{e}_0}$$

It is easy to verify that if $\mathit{mbodysel}(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = (\overline{\mathtt{x}},\mathtt{e}_0)$ then

1. $\mathit{mtypesel}(\mathtt{m},\mathtt{C},\overline{\mathtt{D}}) = \overline{\mathtt{E}} \rightarrow \mathtt{E}$ for some $\overline{\mathtt{E}}$, $\mathtt{E}$ such that $\overline{\mathtt{D}} <: \overline{\mathtt{E}}$.
2. $\overline{\mathtt{x}} : \overline{\mathtt{E}}, \mathtt{this} : \mathtt{B} \vdash \mathtt{e}_0 : \mathtt{D}$ for some $\mathtt{B}$, $\mathtt{D}$ such that $\mathtt{C} <: \mathtt{B}$ and $\mathtt{D} <: \mathtt{E}$.

Then, the result follows from the Substitution Lemma, by replacing

1. $\overline{\mathtt{new\ D}(\overline{\mathtt{u}})}$ of type $\overline{\mathtt{D}} <: \overline{\mathtt{E}}$ to $\overline{\mathtt{x}} : \overline{\mathtt{E}}$;
2. $\mathtt{new\ C}(\overline{\mathtt{v}})$ of type $\mathtt{C} <: \mathtt{B}$ to $\mathtt{this} : \mathtt{B}$.

□

THEOREM 3.4 (PROGRESS). *Let* $\mathtt{e}$ *be a closed expression. If* $\vdash \mathtt{e} : \mathtt{B}$ *for some* $\mathtt{B}$*, then either* $\mathtt{e}$ *is a value or* $\mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$.

PROOF. Straightforward induction on $\vdash \mathtt{e} : \mathtt{B}$. □

Theorems 3.3 and 3.4 show how type safety of FJ can be preserved when adding multi-methods with dynamic overloading with our approach, i.e., any well-typed FMJ program cannot get stuck.

4. WITHOUT COPY SEMANTICS

Motivations for adopting copy semantics come for different purposes (we have already cited [4]). From a foundational point of view, following Meyer [34], inheritance nicely fits a framework where classes are viewed as functions, a special case of sets; the basic idea for modeling inheritance is that a class is the "union" of its own definition and those of its parent(s) (following the model of [1]). The "flattening" of classes is also adopted in engineering of object oriented systems in particular to study the impact of inheritance on object oriented metrics [10].

Although copy semantics is a useful mechanism in most situations, there might still be cases where the programmer of the derived class would actually like to "shadow" the branches of the superclass with the ones in the derived class. For instance, consider the following class that inherits from the `Operation` class we showed in the Introduction:

```
class DecoratedOperation extends Operation {
    Operation forwarder;

    public void op(ElemA e) {
        // do some stuff
        forwarder->op(e);
    }
}
```

This class wants to "capture" all the invocations of the multi-method `m` (for instance, it might want to log the call, or print debugging information) and then delegates the call to the `forwarder` field (e.g., it might be an implementation of the *Decorator* pattern [26]). In this context, the copy semantics would force the programmer of `DecoratedOperation` to redefine all the branches of the superclass, otherwise, it would "intercept" only the most general branch. Notice also that such a solution would not scale to future modifications of superclasses with additional branches.

Another context concerns binary methods; for instance, consider the two "classical" classes:

```
class Point {
    int x; int y;

    public boolean isEqual(Point p) {
        return x == p.x && y == p.y;
    }
}

class ColorPoint extends Point {
    String color;

    public boolean isEqual(ColorPoint p) {
        return super.isEqual(p) && color.equals(p.color);
    }
}
```

With copy semantics the version of `Point` is automatically inherited, namely, a `ColorPoint` considers a standard `Point` equal if the coordinates are equal, although the standard `Point` has no color. Differently, we may want to adopt a different semantics where a `ColorPoint` and a `Point` are not equal in any case.

Thus, we think that a mechanism that allows the programmer to explicitly inhibit copy semantics would be useful[3].

However, if we do not enable copy semantics, we need to consider further type checks, otherwise the program would not be type safe. As illustrated in [16] it is necessary to provide, in the subclass that redefines a multi-method, a definition for the most general branch. Thus, for instance, the `DecoratedOperation` above is type safe, but not the `ColorPoint` (we refer to [16] for further details), that should be modified, e.g., as follows:

```
class ColorPoint extends Point {
    String color;

    public boolean isEqual(Point p) { return false; }

    public boolean isEqual(ColorPoint p) {
        return super.isEqual(p) && color.equals(p.color);
    }
}
```

Thus, the only modification that we should apply to FMJ[4] is to the first *mtype* rule of Figure 2.

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \qquad \mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\} \in \overline{\mathtt{M}} \qquad \{\overline{\overline{\mathtt{B}} \rightarrow \mathtt{B}}\} \supseteq \mathit{MAX}(\mathit{mtype}(\mathtt{m},\mathtt{D}))}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \{\overline{\overline{\mathtt{B}} \rightarrow \mathtt{B}}\}}$$

where $\mathit{MAX}()$ is the set of maximal types, i.e., similarly to $\mathit{MIN}()$:

$$\mathit{MAX}(\{\overline{\overline{\mathtt{B}} \rightarrow \mathtt{B}}\}) \stackrel{def}{=} \{\overline{\mathtt{B}}_i \rightarrow \mathtt{B}_i \in \{\overline{\overline{\mathtt{B}} \rightarrow \mathtt{B}}\} \mid \forall(\overline{\mathtt{B}}_j \rightarrow \mathtt{B}_j) \in \{\overline{\overline{\mathtt{B}} \rightarrow \mathtt{B}}\}$$

$$\forall(\overline{\mathtt{B}}_j \rightarrow \mathtt{B}_j) \in \{\overline{\overline{\mathtt{B}} \rightarrow \mathtt{B}}\} \text{ s.t. } \overline{\mathtt{B}}_i \neq \overline{\mathtt{B}}_j$$

$$\overline{\mathtt{B}}_i \not<: \overline{\mathtt{B}}_j\}$$

[3]For instance, C++ [42] adopts the opposite approach for static overloading: by default there is no copy semantics for overloaded methods and the programmer can explicitly inherit the branches of a superclass by using the keyword `using`.

[4]In this section, for the sake of simplicity, we will *not* consider a version of FMJ with both inheritance mechanisms, thus, here, we take into account only the absence of copy semantics.

This definition of *mtype* is actually too restrictive, in that it does not allow the subclass to define a more general branch w.r.t. all the branches of the superclasses. For instance, the class `DecoratedOperation` could not specify a branch for `op` with parameter `Object`.

To permit such situations, we must relax the above check $\{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\} \supseteq MAX(mtype(\mathtt{m},\mathtt{D}))$ with this one:

$$\forall(\overline{\mathtt{B}}_i \to \mathtt{B}_i) \in MAX(mtype(\mathtt{m},\mathtt{D})).$$

$$\exists(\overline{\mathtt{B}}_j \to \mathtt{B}_j) \in \{\overline{\overline{\mathtt{B}} \to \mathtt{B}}\} \text{ such that } \overline{\mathtt{B}}_i <: \overline{\mathtt{B}}_j$$

However, this raises another problem: since a subclass can specify a more general branch that will hide all the branches of the superclasses, then, it can also make the return type more general (this is allowed by the condition 2 of well-formedness, Definition 2.1).

It is then easy to write an example that shows that this version of the language would not be type safe. For instance, consider this slight variation of the `Operation` example:

```
class Elem {
    public void m() { ... }
}

class Operation {
    public Elem op(Elem e) { ... }
}

class DecoratedOperation extends Operation {
    public Object op(Object e) { ... }
}

Operation o = new DecoratedOperation();
Elem e = new Elem();
...
o.op(e).m();
```

This program is statically well-typed, but at run-time, the `op` in `DecoratedOperation` will be selected, which returns an `Object` and the invocation of `m` on the returned object will generate a "message-not-understood" error.

In order to make the language sound, we must restrict the second condition of well-formedness (see Definition 2.1) by not allowing covariant return types:

$$\overline{\mathtt{B}}_i <: \overline{\mathtt{B}}_j \Rightarrow \mathtt{B}_i = \mathtt{B}_j$$

With this definition of well-formedness the above program would be rejected during the static type checking.

5. RELATED WORKS

Various object-oriented (core) languages have been proposed to study multi-methods and dynamic overloading. In the following we mention most of the them, by focusing on their features that differ from our approach.

CLOS [11] 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* [39] 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* [20] 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.

The language KOOL [18] integrates multi-methods and overloaded functions (external to classes) in a functional kernel object-oriented language in a type safe way. The notion of well formedness for FMJ types is widely inspired by the one defined in [18]. KOOL includes both encapsulated multi-methods and overloaded functions external to classes (not included in FMJ) thus unifying in a single language two different styles of programming; on the other hand KOOL does not permit mutually recursive class definitions, namely, it does not allow the programmer to write a method with a parameter of type *A* in a class if the class *A* is defined after the current class. Finally, KOOL does not interpret overloading by copy semantics, so many multi-method definitions which are well typed in FMJ are discarded in KOOL. Thus FMJ permits a less restrictive overloading resolution policy still maintaining the primary goal of type safety.

Tuple [32] is a simple object-oriented language designed to integrate dynamic overloading in a language by adding tuples as primitive expressions. Tuple is statically typed and it is proved to be type safe based on the type checking algorithm presented in [19]. In Tuple methods can be defined either inside classes or in *tuple classes* (external to standard classes). Message lookup on methods defined in a tuple supports dynamic overloading: messages are sent to tuples of arguments dynamically involved in method selection. There are some key differences between Tuple and FMJ. Differently from FMJ, Tuple supports dynamic overloading only for methods external to classes.

Dubious [35] is a core calculus designed to study modular static type checking of multi-methods in modules. Dubious is classless and includes multi-methods with symmetric dispatch in the form of generic functions defined in modules that can be checked separately and then linked in a type safe way. In [35] several type systems are discussed in order to find the right balance between flexibility and type safety. Dubious and FMJ share some basic features such as the symmetry of method dispatch and static type safety.

Fortress [3] is an object-oriented language supporting methods within *traits* [38] and functions defined outside traits. It also provides components (units of compilation) which contain declarations of objects, traits and functions. Fortress differs from mainstream languages since it is not class-based.

Parasitic multi-methods [12] are a variant of the encapsulated multi-methods of [13, 18] applied to Java. This extension is rather flexible and indeed provides modular dynamic overloading. The goal of modularity has influenced many aspects of the design:

- method selection is asymmetric, i.e., the receiver's type is evaluated before the argument, in method selection;
- the selection of the most specialized method takes place through `instanceof` checks and consequent type casts
- 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;

- the rules concerning the interaction between standard and parasitic methods w.r.t. overriding and overloading are quite complex thus requiring some extra effort to the programmer.

MultiJava [21] is an extension of Java to support open classes (classes to which new methods can be added without editing the class directly) and multi-methods. Once again this solution 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 with open classes syntax: 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. MultiJava modular type system is based on the one of Dubious.

Cmm [41] is a preprocessor providing multi-methods in C++. *Cmm* is based on the proposal of [40]. 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.

Other approaches add dynamic overloading to Java without type checking the proposed extension: *JMMF* [25] is a framework implemented using reflection mechanism, while in [14] 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.

[24] proposes an extension of the JVM to provide dynamic overloading in Java without modifying neither the syntax nor the type system: the programmer directly selects the classes which should use dynamic overloading. The problem of this approach is that ambiguous method calls can arise dynamically.

[27] provides a library extension to implement a limited form of dynamic overloading in Java, based on a visitor pattern-like code. Again, this approach suffers from possible run-time exceptions.

Chapter 11 of [2] presents some solutions to implement dynamic overloading 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. Moreover, some of the approaches presented in [2] do not correctly interact with inheritance.

6. CONCLUSIONS AND FUTURE WORK

The contribution of this paper is to provide a formal framework for reasoning about extensions of Java-like languages with multi-methods. Distinguishing features of our proposal are: (i) multi-methods are encapsulated in classes, (ii) dynamic selection of the multi-method branch is symmetric, (iii) we adopt a copy semantics of inheritance, (iv) the extended language FMJ preserves type safety.

In this formal perspective, we do not deal with efficiency issues: the dynamic overloading selection here presented is not efficient due to the recurrent use of the lookup functions (Figure 2), which basically inspect the classes in the hierarchy and all the available branches. On the contrary, in the actual implementation of a language, it is crucial that the selection takes place efficiently; for instance, dynamic binding is usually implemented in constant time, i.e., independently from the number of classes of the hierarchy.

Indeed, from the implementation point of view, the lookup functions must be intended as lookup tables that are built once and for all, possibly by the compiler. However, this would still require a run-time inspection of these tables, and the branch selection might not be constant (see, e.g., some works on the optimizations of these tables, such as [5, 23, 43]). Moreover, concerning the type checking of multi-method signatures with respect to a given list of types, a polynomial-time algorithm is developed in [19] which is applicable to many contexts, including FMJ.

We already did some work on multi-methods as a language extension of C++ (see [7, 8]). In particular, the software `doublecpp`, `http://doublecpp.sf.net`, is based on a preprocessor that, given a program written in the extended C++, produces an equivalent program in standard C++, by applying a program transformation based on the double dispatch technique ([30, 26]). The translated code implements dynamic overloading selection by using only static overloading and dynamic binding (in particular, it never uses RTTI checks) and thus the selection takes place in constant time, independently from the class hierarchy and from the number of branches.

We plan to follow the same technique to extend Java with multi-methods: we will adapt the transformation technique used for C++ to Java, we will formalize the transformation from an extended Java to standard Java using the extension of FJ presented in this paper. The formalization will prove not only that the translated program is still type safe, but also that the semantics of the translated program is the same as the semantics of the original one.

Acknowledgments. We thank the anonymous referees for providing helpful hints for improving the paper.

7. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

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

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification Version 1.0, 2006. Sun Microsystems, available on line.

[4] D. Ancona, S. Drossopoulou, and E. Zucca. Overloading and Inheritance. In *FOOL 8*, 2001.

[5] P. André and J.-C. Royer. Optimizing method search with lookup caches and incremental coloring. In *Proc. of OOPSLA '92*, pages 110–126, New York, NY, USA, 1992. ACM Press.

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

[7] L. Bettini, S. Capecchi, and B. Venneri. Translating double dispatch into single dispatch. *Electr. Notes Theor. Comput. Sci*, 138(2), 2005.

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

[9] A. Beugnard. Method overloading and overriding cause encapsulation flaw: an experiment on assembly of heterogeneous components. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1424–1428. ACM Press, 2006.

[10] D. Beyer, C. Lewerentz, and F. Simon. Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems. In *IWSM '00: Proceedings of the 10th International Workshop on New Approaches in Software Measurement*, pages 1–17. Springer, 2000.

[11] D. Bobrow, L. Demichiel, R. Gabriel, S. Keene, and G. Kiczales. Common Lisp Object System Specification.

Lisp and Symbolic Computation, 1(3/4):245–394, 1989.

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

[13] K. B. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[14] P. Carbonetto. An implementation for multiple dispatch in java using the elide framework.

[15] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[16] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[17] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2):297–352, 1995.

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

[19] C. Chambers and G. Leavens. Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst.*, 17(6):805–843, 1995.

[20] C. Chambers and G. T. Leavens. BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing. In *The 4th Int. Workshop on Foundations of Object-Oriented Languages, FOOL 4*, 1996.

[21] C. Clifton, G. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. *ACM SIGPLAN Notices*, 35(10):130–145, 2000.

[22] L. DeMichiel and R. Gabriel. The Common Lisp Object System: An Overview. In *Proc. ECOOP*, volume 276 of *LNCS*, pages 151–170. Springer, 1987.

[23] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proc. of OOPSLA '95*, pages 141–155. ACM Press, 1995.

[24] C. Dutchyn, 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)*, pages 77–92, 2001.

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

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

[27] C. Grothoff. Walkabout revisited: The Runabout. In *Proc. of ECOOP*, number 2743 in LNCS. Springer, 2003.

[28] A. Igarashi and H. Nagira. Union Types for Object-Oriented Programming. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, pages 1435–1441. ACM, 2006.

[29] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[30] D. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proc. OOPSLA*, pages 347–349. ACM Press, 1986.

[31] D. Lea. Run-Time Type Information and Class Design. In *Proc. USENIX C++ Technical Conference*, pages 341–347. USENIX, 1992.

[32] G. Leavens and T. Millstein. Multiple dispatch as dispatch on Tuples. In *OOPSLA '98*, pages 374–387. ACM Press, 1998.

[33] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[34] B. Meyer. Overloading vs. Object Technology. *Journal of Object-Oriented Programming*, pages 3–7, October/November 2001.

[35] T. D. Millstein and C. Chambers. Modular statically typed multimethods. *Inf. Comput*, 175(1):76–118, 2002.

[36] W. Mugridge, J. Hamer, and J. Hosking. Multi-Methods in a Statically-Typed Programming Language. In *Proc. ECOOP '91*, volume 512 of *LNCS*, pages 307–324. Springer, 1991.

[37] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[38] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.

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

[40] J. Smith. Draft proposal for adding Multimethods to C++. Available at `http://std.dkuug.dk/jtc1/sc22/-wg21/docs/papers/2003/n1529.html`.

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

[42] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[43] J. Vitek and R. N. Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *CC '96: Proc. of the 6th Int. Conf. on Compiler Construction*, volume 1060 of *LNCS*, pages 309–325. Springer, 1996.