Overloading and Inheritance

Davide Ancona[1], Sophia Drossopoulou[2], Elena Zucca[1]

[1] DISI, University of Genova, [2] Department of Computing, Imperial College
E-mail: {zucca, davide}@disi.unige.it sd@doc.ic.ac.uk

Abstract

Overloading allows several function definitions for the same name, distinguished primarily through different argument types; it is typically resolved at compile-time. Inheritance allows subclasses to define more special versions of the same function; it is typically resolved at run-time.

Modern object-oriented languages incorporate both features, usually in a type-safe manner. However, the combination of these features sometimes turns out to have surprising, and even counterintuitive, effects.

We discuss why we consider these effects inappropriate, and suggest alternatives. We explore the design space by isolating the main issues involved and analyzing their interplay and suggest a formal framework describing static overloading resolution and dynamic function selection, abstracting from other language features. We believe that our framework clarifies the thought process going on at language design level.

We introduce a notion of soundness *and* completeness *of an overloading resolution policy w.r.t. the underlying dynamic semantics, and a way of comparing the* flexibility *of different resolution policies. We apply these concepts to some non-trivial issues raised in concrete languages.*

We also argue that the semantics of overloading and inheritance is "clean" only if it can be understood through a copy semantics, whereby programs are transformed to equivalent programs without subclasses, and the effect of inheritance is obtained through copying.

1 Introduction

Overloading allows for functions that execute different code for arguments of different types, and is supported by many programming languages [17, 15]. It is also called *ad hoc polymorphism*, in contrast to *parametric polymorphism*, as, *e.g.*, in [14], which allows for functions whose code works uniformly on arguments of different types [5]. Earlier languages offer only a form of overloading where the appropriate body for a function call is selected at compile time, that is, *overloading resolution* is static (*early* or *static* binding). This form of overloading can be reduced to a syntactic abbreviation.

In languages where types may be computed at run-time, indeed, the appropriate body for a function can be selected depending on the *dynamic type* of the arguments; hence we have *late* or *dynamic binding*. This happens typically with *inheritance* in object-oriented languages; usually run-time selection is adopted only for the receiver (*i.e.*, the first, implicit, parameter) of a method call. The same policy can also be applied to all, or some of, function arguments, as, *e.g.*, in *multimethods* [12, 7]. Foundations of languages supporting run-time selection are investigated in [6].

In many widely-used languages, *e.g.*, Java [4] and C++ [15], a combination of the two mechanisms is supported, and the terms "overloading" and "inheritance" are used to denote the static and the dynamic selection, respectively. This is the sense in which we use these terms in our paper.

In such languages, a class can both *overload* methods, that is, have methods with the same name but different argument types, and *override* methods inherited from parent classes. Hence, given a method call, the choice of the code to be executed requires two steps: first a static selection, performed on the basis of the static types of the receiver and the arguments, then a dynamic selection, performed on the basis of the dynamic type of the receiver and some information, which we call *(method) descriptor*, stored with the call at compile-time.

Obviously, the primary aim of a language design needs to be type-safety. Achieving it in the context of both inheritance and overloading is non-trivial [8, 13]. However, there are further concerns, to do with how permissive and intuitive the ensuing semantics is. In previous work [3] we discussed cases where Java overloading resolution is too restrictive, and suggested alternatives.

In this paper we provide a more rigorous foundation to our earlier observations. We define a framework formalizing the mechanisms of static and run-time selection, and their interaction, abstracting from orthogonal features. Within this framework, we introduce two notions evaluating overloading resolution policies.

The first notion is *soundness* and *completeness w.r.t.* the underlying dynamic semantics. Soundness requires all descriptors which are not distinguishable by overloading resolution not to be distinguishable when used for method selection; completeness requires the converse. Soundness guarantees determinism; completeness guarantees that overloading resolution is not more discriminating than necessary. Completeness is not satisfied by the 1996 Java version [11] nor by C++ [15].

The second notion is *flexibility*: an overloading resolution policy is more flexible than another iff it gives the same meaning to all method calls which are unambiguous for the latter. We describe alternatives which are sound and complete, and strictly more flexible than Java.

A plausible expectation is that inheritance should be explainable through a *copy semantics*, whereby any program should be transformable into an equivalent program without inheritance, inherited methods should be copied into the bodies of the transformed subclasses. Indeed, similar mechanisms for object based calculi are used in [1, 10]. Interestingly, it turned out that some approaches, *e.g.*, the 1996 Java version, do not have a natural copy semantics, and the approach that has the best properties in terms of soundness and completeness has the simplest copy semantics.

In the next section we explore the above issues through examples. In section 3 we give a formal framework for overloading and inheritance. In section 4 we apply the framework to analyze the policy of Java, describe two alternatives and demonstrate their properties. We also show an extension allowing multiple inheritance and compare it with the C++ approach. In section 6 we draw conclusions and describe further work.

2 Examples

As outlined earlier, a method call has an associated static *overloading resolution*, and a dynamic *method selection* step. Overloading resolution consists of two phases: first, determine the associated set of *candidate methods*, *i.e.*, methods which could correspond to the call; then, select from this set the method (or, in general, the methods; as shown later, sometimes more than one method can be selected) which *fits* best, and store its descriptor with the method call. Based on the dynamic class of the receiver and the stored method descriptor, method selection finds the appropriate method body, which needs not be the same as that chosen at compile-time.

Unless explicitly stated, in all following examples B' is assumed to be a subclass of A' and the set of candidate methods is determined according to the Java rule (see 15.11.2.1 in [11] or 15.12.2.1 in [4]): for a call of a method m whose receiver and actual arguments have static types c and $\mathsf{t}_1 ... \mathsf{t}_n$, candidates are all methods either declared or inherited[1] by c with name m and parameter types $\mathsf{t}'_1 ... \mathsf{t}'_n$, where t'_i supertype of t_i, for all $1 \leq i \leq n$.

2.1 Simple cases of overloading resolution

We illustrate first the simpler cases of overloading resolution. Consider the following example.

```
class A {               int m(A' x) { return 1; } }
class B extends A {     int m(B' x) { return 2; } }
```

Consider method call $\mathsf{b.m(b')}$ where b, b' are expressions of type B, B', respectively. There are two candidate methods, since both could be correctly associated to the given call, but there is no doubt that the method declared in B "fits better" than that declared in A. Java terminology says that the method from B is *more specific* than that from A.

In the example above, both the argument type of the more specific method and the class where it is declared are a subtype of those of the other. The choice between candidate methods is also straightforward when either they are declared in the same class and one has more specific argument types or both have the same argument types but one is declared in an heir class (hence overrides the other).

Finally, the following shows that overloading resolution may flag static errors, when there is no "reasonable" way to choose between candidate methods.

```
class C {   int m(A' x, B' y) { return 1; }
            int m(B' x, A' y) { return 2; } }
```

In this case, in the call $\mathsf{c.m(b', b')}$, where c and b' have types C and B', respectively, neither method fits better, hence this is a static error, and the call is said to be *ambiguous*.

We now turn to examples where the choice as to which method fits better is less obvious.

2.2 Resolution too restrictive – possibly

The following example demonstrates that languages may be too restrictive in overloading resolution.

Example 1

```
class A {               int m(B' x) { return 1; } }
class B extends A {     int m(A' x) { return 2; } }
class C extends B {     }
```

In this case, there is contravariance between inheritance and argument subtyping, hence the meaning of the call $\mathsf{c.m(b')}$ is debatable. In C++ it resolves to the method in class B (see later). In Java the call is ambiguous, since a method is more specific than another only if it has more specific argument types *and* is declared in a subclass/subinterface.

[1] In Java inherited means not overridden (see 8.4.6 in [11, 4]).

People are puzzled when first confronted with this behavior in Java. There are counter-arguments both on the methodological and the descriptive level.

On the methodological level, the Java approach has the implication (bad for modular software development) that clients of a class need to know not only which methods are provided by it, but also from which class each method has been inherited. For instance, the specification of class C states that it has two methods int m(B′ x) and int m(A′ x); but this information is insufficient to predict the effect of the call c.m(b′).

On the descriptive level, this rule does not support a simple copy semantics (in the sense explained below).

We now show four different approaches where c.m(b′) is not ambiguous: the former two approaches adopt the Java rule for candidate methods, but are based on two alternative rules for selecting the method that fits best, whereas the latter two approaches are based on different rules for determining the set of candidates.

First Alternative: arguments-only rule A better alternative would consider the argument types only, and select for c.m(b′) the method returning 1. This solution corresponds to a cleaner view both on the methodological and the descriptive level.

With this approach, indeed, inheritance is explainable as a mechanism for code sharing without duplication. It can be seen as a linguistic mechanism allowing to get for free the same effect as if one duplicated "by hand" in the heir all parent's methods (unless overridden).

With copy semantics our example is transformed to:

```
class A {               int m(B' x)  { return 1;  } }
class B extends A {     int m(A' x)  { return 2;  }
                        int m(B' x)  { return 1;  } }
class C extends B {     int m(A' x)  { return 2;  }
                        int m(B' x)  { return 1;  } }
```

In the above, the call c.m(b′) would be unambiguous and would return 1. In Sect.4.1 (Prop.4), we formally express this fact by showing that the arguments-only rule is strictly more flexible than the Java (componentwise) rule.

Second Alternative: subclasses first rule Another possible solution would be for the method call c.m(b′) to return 2, selecting from the set of candidates the method declared in the first superclass.

This alternative can be explained in terms of a different copy semantics: we copy each method of the parent in the heir, unless the heir contains a method with the same name and argument types which are supertypes of those of the parent's method. For instance, example 1 would be transformed as follows.

```
class A {               int m(B' x)  { return 1;  } }
class B extends A {     int m(A' x)  { return 2;  } }
class C extends B {     int m(A' x)  { return 2;  } }
```

This solution has the methodological advantage of bringing static overloading resolution in line with dynamic method selection, which selects the *first appropriate* method.

Third Alternative: C++ rule C++ differs from the two previously described approaches in the definition of the candidate methods. For a call of a method m whose receiver and actual arguments have static types c and $t_1 \ldots t_n$, respectively, candidates are all methods which have name m and parameters types $t'_1 \ldots t'_n$, with t'_i supertype of t_i, for all $1 \leq i \leq n$, and which are declared (or imported via using clauses) in the first superclass of c which has *some* method named m. According to this rule, the method defined in B is the only candidate for the call c.m(b′). More details can be found in appendix B.

The C++ rule can be explained in terms of a different copy semantics: We copy each method of the parent in the heir, unless the heir contains a method with the same name; methods of classes mentioned in *using* clauses are copied in any case. Examples are given in appendix B.

Fourth Alternative: Contravariant methods If we allow overriding of contravariant methods, then the method declared in B overrides that declared in A; as a consequence, if overridden methods are not candidates as happens in Java, then again the method defined in B is the only candidate for the call c.m(b′).

In this case the copy semantics is obtained by duplicating parent's methods (unless overridden) in the heir, as happens for the first alternative.

```
class A {               int m(B' x)  { return 1;  } }
class B extends A {     int m(A' x)  { return 2;  } }
class C extends B {     int m(A' x)  { return 2;  } }
```

2.3 Resolution unduly restrictive – undoubtedly

In the following case, overloading resolution is unduly restrictive, because it flags an ambiguous call where, in fact, in any execution there is exactly one matching method body.

In a language which supports *abstract* methods, like, *e.g.*, Java, it is possible for a class/interface to inherit more than one method with the same argument and return types (see [4] 8.4.6.4). This leads to a particular case of ambiguity, illustrated below.

Example 2

```
interface I1 {                     int m();  }
interface I2 {                     int m();  }
interface I3 extends I1, I2 {        }
```

Consider the call $\mathsf{i.m()}$ with i of type I. Following the Java rule based on componentwise ordering, from chapter 15.11.2.2 in [11], this call should be ambiguous (even though different Java compilers have different, and sometime obscure, behavior on this and similar examples). The second alternative gives the same result as well.

In our view, this choice is unduly restrictive for even stronger reasons than those from 2.2: Any class implementing interface I will either declare or inherit *exactly one* method $\mathsf{int\ m()}$; thus, execution of $\mathsf{i.m()}$ will always find exactly one method body. Thus, overloading resolution is unduly discriminating. In Sect.4.2 (Prop.6) we formalize this by proving that the Java rule is not *complete*, that is, it makes ambiguous calls for which the ambiguity is not semantically relevant. Overloading resolution in C++ has the same drawback (a similar example to the above can be constructed using abstract methods).

On the contrary, the first alternative makes the call unambiguous. Both the Java rule and the second alternative can be changed so that the call becomes unambiguous, by treating abstract methods specially (see Def.8 in Sect.4.2). For Java this corresponds to the overloading rule given in the revised Java specification ([4] 15.12.2.2).

2.4 Overloading and multiple inheritance

We now consider the case where classes may inherit more than one *non-abstract* method with the same argument and return types. This is possible in languages with *multiple inheritance*[2], as, *e.g.*, C++.

Let us consider first the case in which a class inherits more than one *non virtual*[3] method with the same argument and return types, as in the following example[4].

Example 3

```
class C  {                  int m()  { return 1;  }  }
class C1 : virtual C  {     int m()  { return 2;  }  }
class C2 : virtual C  {     int m()  { return 3;  }  }
class C3 : virtual C1, virtual C2  {  }
```

Here, the the call $\mathsf{c3.m()}$, with $\mathsf{c3}$ of type $\mathsf{C3}$ *should* be ambiguous.[5] However, there is no reason to forbid the declaration of $\mathsf{C3}$ – calls of other methods with receiver $\mathsf{c3}$ could be valid, and subclasses of $\mathsf{C3}$ may define their own method $\mathsf{int\ m()}$. On the other hand, if a class inherits different *virtual* methods which do not override each other (as $\mathsf{C3}$ in the example above if all methods are virtual), then the class is ill-formed, since there would be no way to prevent run-time ambiguities by type-checking method calls: for instance, the call $\mathsf{c.m()}$, where c has static class C, would cause a run-time ambiguity on a receiver with dynamic type $\mathsf{C3}$.

In Sect.4.3 we suggest a formal overloading resolution rule which takes these problems into account, and we prove its soundness and completeness.

3 Language and Overloading frameworks

We develop an abstract framework distilling the essential components from overloading and inheritance as in several programming languages. We consider languages with overloading, inheritance, classes, abstract classes, interfaces, where classes are types, and subclasses are subtypes.

For any programming language we assume an associated set of well-formed programs P. For any $\mathsf{p}{\in}\mathsf{P}$, we assume associated sets of method names $\mathsf{M_p}$, classes $\mathsf{C_p}$, interfaces $\mathsf{I_p}$, and abstract classes $\mathsf{AC_p}$ with $\mathsf{AC_p} \subseteq \mathsf{C_p}$. The sets of class and interface names must be disjoint; their union represents possible types $\mathsf{T_p}$ and is ordered by a subtype relation $\sqsubseteq_\mathsf{p}$. Note that, even though both have no instances (differently from non-abstract classes), abstract classes and interfaces are separate entities, since the former are involved in run-time method selection.

The set of all possible method bodies is described by $\mathsf{B_p}$.

Method calls involve two processes: At compile time, the method call is enriched by additional information, which we call *descriptor*, formalized as the set Dscr. At run-time, the method body is selected based on the dynamic class of the receiver, the name of the method, and the method descriptor. This is formalized by the function $MSel_\mathsf{p}\colon \mathsf{C_p} \times \mathsf{M_p} \times \mathsf{Dscr_p} \longrightarrow \mathsf{B_p} \cup \{\mathsf{none}\}$, which returns a method body, or none if no body is found.[6]

The compile-time calculation of descriptors involves two stages: In the first stage, the candidate methods are collected, based on the static type of the receiver, the method name, and the static types of the arguments, *i.e.*, the function $CndMs_\mathsf{p}\colon \mathsf{T_p} \times \mathsf{M_p} \times \mathsf{T_p^*} \longrightarrow \mathcal{P}(\mathsf{Dscr_p})$[7] is applied. In the second stage, the maximally specific candidates (according to a *more specific* relation $\preceq_\mathsf{p}$) are collected. The call is *un-ambiguous* if this set contains equivalent descriptors (in the sense of $\preceq_\mathsf{p}$) only.

The predicate $:_\mathsf{p}$, defined on $\mathsf{B_p} \times (\mathsf{C_p} \backslash \mathsf{AC_p}) \times \mathsf{Dscr_p}$, determines whether a body $\mathsf{b} \in \mathsf{B_p}$ satisfies the descriptor $\mathsf{d} \in \mathsf{Dscr_p}$ when executed by objects of class $\mathsf{c} \in \mathsf{C_p} \backslash \mathsf{AC_p}$ (dynamic class of the receiver).

Since, as already noted, overloading resolution consists of two distinct phases, we distinguish the *programming language framework*, as in Def.1, which may underly different more specific relations, thus giving different *overloading frameworks*, as in Def.2.

[2] There, selection among methods inherited from different parent classes is called conflict rather than overloading resolution, but obviously the problem is the same.

[3] That is, a method for which binding is static.

[4] Notice that we are using virtual inheritance; more in Sect.4.3.

[5] As a consequence, the arguments-only rule is no longer sound; this corresponds to the fact that there is no simple copy semantics of multiple inheritance.

[6] $MSel_\mathsf{p}$ is defined on all classes, therefore on abstract classes, in particular. This allows a neat, recursive definition of $MSel_\mathsf{p}$ in the particular frameworks introduced in Sect. 4.

[7] $\mathcal{P}(A)$ denotes the powerset of set A.

Definition 1 *We call a tuple* $\mathcal{L} = (\mathsf{P}, \mathsf{M}, \mathsf{C}, \mathsf{AC}, \mathsf{I}, \sqsubseteq, \mathsf{Dscr}, \mathit{CndMs}, \mathsf{B}, \mathit{MSel}, :)$ *a* language framework *iff, for any* $\mathsf{p} \in \mathsf{P}$*:*

- M_p *is a set,*
- C_p *and* I_p *are disjoint sets,* $\mathsf{AC}_\mathsf{p} \subseteq \mathsf{C}_\mathsf{p}$,
- $\sqsubseteq_\mathsf{p}$ *is a subtype relation on* T_p, *where* $\mathsf{T}_\mathsf{p} = \mathsf{I}_\mathsf{p} \cup \mathsf{C}_\mathsf{p}$,
- Dscr_p *is a set,*
- $\mathit{CndMs}_\mathsf{p}\colon \mathsf{T}_\mathsf{p} \times \mathsf{M}_\mathsf{p} \times \mathsf{T}_\mathsf{p}^* \longrightarrow \mathcal{P}(\mathsf{Dscr}_\mathsf{p})$ *is a total function,*
- $\mathit{MSel}_\mathsf{p}\colon \mathsf{C}_\mathsf{p} \times \mathsf{M}_\mathsf{p} \times \mathsf{Dscr}_\mathsf{p} \longrightarrow \mathsf{B}_\mathsf{p} \cup \{\mathsf{none}\}$ *is a total function,*
- $:_\mathsf{p} \subset \mathsf{B}_\mathsf{p} \times (\mathsf{C}_\mathsf{p} \backslash \mathsf{AC}_\mathsf{p}) \times \mathsf{Dscr}_\mathsf{p}$.

From now on, $\mathcal{L}$ denotes given language framework, with $\mathcal{L} = (\mathsf{P}, \mathsf{M}, \mathsf{C}, \mathsf{AC}, \mathsf{I}, \sqsubseteq, \mathsf{Dscr}, \mathit{CndMs}, \mathsf{B}, \mathit{MSel}, :)$.

Definition 2 *We call a pair* $\mathcal{O} = \langle \mathcal{L}, \preceq \rangle$ *an* overloading framework *iff:*

- $\mathcal{L}$ *is a language framework,*
- $\preceq_\mathsf{p}$ *is a relation on* Dscr_p,
- *set* $\mathit{MaxMs}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}) =$
  $\{\, \mathsf{d} \mid \mathsf{d} \in \mathsf{CM},$ *and* $\mathsf{d}' \in \mathsf{CM}, \mathsf{d}' \preceq_\mathsf{p} \mathsf{d} \implies \mathsf{d} \preceq_\mathsf{p} \mathsf{d}' \,\}$,
  for any $\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}$, *and where* $\mathsf{CM} = \mathit{CndMs}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$,
  then $\preceq_\mathsf{p}$ *is a pre-order*[8] *on* $\mathit{MaxMs}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$,
  for any $\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}$.

We call $\mathcal{L}$ the *underlying* language framework of $\mathcal{O}$, and the relation $\preceq_\mathsf{p}$ a *more specific relation.* In appendix A we apply the above definition to the first example in 2.1. In section 4 we define different frameworks corresponding to Java overloading resolution and alternatives, and a framework including multiple inheritance.

$\preceq_\mathsf{p}$ is a pre-order only on the sets $\mathit{MaxMs}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$, and not on Dscr_p. The reason for that is that the requirement for $\preceq_\mathsf{p}$ to be pre-order on Dscr_p is not always met in programming languages (see Def.8 and the counter-example in App.C).

From now on, $\mathcal{O} = \langle \mathcal{L}, \preceq \rangle$ denotes some given overloading framework.

Definition 3 *We define the relation* $\equiv^{\preceq}_\mathsf{p}$ *on* Dscr_p, *and the predicate* $\mathit{UnAmb}^{\preceq}_\mathsf{p}$ *on* $\mathsf{T}_\mathsf{p} \times \mathsf{M}_\mathsf{p} \times \mathsf{T}_\mathsf{p}^*$ *as follows:*

- $\mathsf{d}' \equiv^{\preceq}_\mathsf{p} \mathsf{d}$ *iff* $\mathsf{d}' \preceq_\mathsf{p} \mathsf{d}$ *and* $\mathsf{d} \preceq_\mathsf{p} \mathsf{d}'$,
- $\mathit{UnAmb}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$ *iff*
  $\forall \mathsf{d}, \mathsf{d}' \in \mathit{MaxMs}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}) : \mathsf{d} \equiv^{\preceq}_\mathsf{p} \mathsf{d}'$.

[8] *i.e.*,, transitive and reflexive, but not necessarily anti-symmetric.

$\equiv^{\preceq}_\mathsf{p}$ is an equivalence relation on $\mathit{MaxMs}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$, since $\preceq_\mathsf{p}$ is a pre-order on that set. Also, a call is ambiguous only if there are non comparable maximally specific candidates, since the following lemma trivially holds.

Lemma 1 *For all* $\mathsf{p}, \mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}, \mathsf{d}, \mathsf{d}'$:
$\mathsf{d}, \mathsf{d}' \in \mathit{MaxMs}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}), \mathsf{d} \preceq_\mathsf{p} \mathsf{d}' \implies \mathsf{d} \equiv^{\preceq}_\mathsf{p} \mathsf{d}'$.

3.1 Properties of frameworks

In order for an overloading framework to be part of a safe type system, we require that method selection starting from a (non-abstract) class c and based on the descriptor d of any candidate method provides a method body (*i.e.*, $\mathit{MSel}_\mathsf{p}(\mathsf{c}, \mathsf{m}, \mathsf{d}) \neq \mathsf{none}$, which guarantees progress), and that this method body when executed by objects of class c satisfies the given descriptor (*i.e.*, $\mathit{MSel}_\mathsf{p}(\mathsf{c}, \mathsf{m}, \mathsf{d}) :_\mathsf{p} \langle \mathsf{c}, \mathsf{d} \rangle$, which guarantees type preservation).

Definition 4 $\mathcal{L}$ *is* type safe *iff, for all* $\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}, \mathsf{c} \in \mathsf{C}_\mathsf{p} \backslash \mathsf{AC}_\mathsf{p}$:

$\mathsf{d} \in \mathit{CndMs}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}), \mathsf{c} \sqsubseteq_\mathsf{p} \mathsf{t}_0 \implies$

- $\mathit{MSel}_\mathsf{p}(\mathsf{c}, \mathsf{m}, \mathsf{d}) \neq \mathsf{none}$,
- $\mathit{MSel}_\mathsf{p}(\mathsf{c}, \mathsf{m}, \mathsf{d}) :_\mathsf{p} \langle \mathsf{c}, \mathsf{d} \rangle$.

As we argued in section 2.2, there are further requirements: Overloading resolution should make as many distinctions as can be perceived through dynamic method selection, but no more. Thus, we define the notions of *sound* and *complete* overloading resolution.

Definition 5 *For any* $\mathsf{p} \in \mathsf{P}$, $\mathsf{t}_0 \in \mathsf{T}_\mathsf{p}$, *we define the* semantic equivalence $\approx^{\mathsf{t}_0}_\mathsf{p}$ *on* Dscr_p *as*

- $\mathsf{d} \approx^{\mathsf{t}_0}_\mathsf{p} \mathsf{d}'$ *iff*
  $\forall \mathsf{c} \sqsubseteq_\mathsf{p} \mathsf{t}_0 : \mathit{MSel}_\mathsf{p}(\mathsf{c}, \mathsf{d}) = \mathit{MSel}_\mathsf{p}(\mathsf{c}, \mathsf{d}')$.

Then, the more specific relation $\preceq$ *is*

- sound *w.r.t.* $\mathcal{L}$ *iff* $\forall \mathsf{p} \in \mathsf{P}, \mathsf{d}, \mathsf{d}' \in \mathit{MaxMs}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$:
  $\mathsf{d} \equiv^{\preceq}_\mathsf{p} \mathsf{d}' \implies \mathsf{d} \approx^{\mathsf{t}_0}_\mathsf{p} \mathsf{d}'$,
- complete *w.r.t.* $\mathcal{L}$ *iff* $\forall \mathsf{p} \in \mathsf{P}, \mathsf{d}, \mathsf{d}' \in \mathit{MaxMs}^{\preceq}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$:
  $\mathsf{d} \approx^{\mathsf{t}_0}_\mathsf{p} \mathsf{d}' \implies \mathsf{d} \equiv^{\preceq}_\mathsf{p} \mathsf{d}'$.

Completeness is a strong language design criterion. It requires any compile-time ambiguous call to be sensitive to the choice of descriptor from the set of maximally specific descriptors. Java overloading as in the 1996 version is not complete, as illustrated in example 2, and formally argued in Prop. 6.

Lemma 2 *$\preceq$ is sound w.r.t. $\mathcal{L}$ if it is a partial order*[9] *on $MaxMs^{\preceq}_{\mathsf{p}}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$, for any $\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}$.*

A more specific relation is less flexible than another more specific relation if each unambiguous call *w.r.t.* the first is unambiguous *w.r.t.* the latter, and gives the same result:

Definition 6 *Let $\langle\mathcal{L}, \preceq^1\rangle$ and $\langle\mathcal{L}, \preceq^2\rangle$ be overloading frameworks s.t. $\preceq^1$ and $\preceq^2$ are sound w.r.t. $\mathcal{L}$. We say that $\preceq^1$ is* less flexible *than $\preceq^2$ w.r.t. $\mathcal{L}$ iff, for all $\mathsf{p}{\in}\mathsf{P}$, and for all $\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}$:*

$UnAmb^{\preceq^1}_{\mathsf{p}}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}) \implies$

- $UnAmb^{\preceq^2}_{\mathsf{p}}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$
- $\forall \mathsf{d}{\in}MaxMs^{\preceq^1}_{\mathsf{p}}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}), \mathsf{d}'{\in}MaxMs^{\preceq^2}_{\mathsf{p}}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$: $\mathsf{d} \approx^{\mathsf{t}_0}_{\mathsf{p}} \mathsf{d}'$.

Note that, by virtue of the soundness hypothesis, in the second consequence universal quantifiers can be equivalently replaced by existential quantifiers.

4 Examples of overloading frameworks

We now consider the issues illustrated in section 2 in terms of the formalization introduced in the previous section. We first develop a language framework $\mathcal{L}^s$ which corresponds to a Java-like language with single inheritance and non-abstract classes only. On top of $\mathcal{L}^s$ we define three overloading frameworks, $\mathcal{O}^{s,j}$, $\mathcal{O}^{s,a1}$ and $\mathcal{O}^{s,a2}$ which correspond to the Java rule and the two alternative rules informally introduced in section 2. We show that $\mathcal{O}^{s,j}$, $\mathcal{O}^{s,a1}$ and $\mathcal{O}^{s,a2}$ are sound and complete, but $\mathcal{O}^{s,j}$ is strictly less flexible than $\mathcal{O}^{s,a1}$ and $\mathcal{O}^{s,a2}$.

Then, we enrich $\mathcal{L}^s$ with abstract classes and interfaces, and obtain $\mathcal{L}^a$. We show that the first alternative is still complete, while the Java rule and the second alternative rule are not, and show how to make them complete.

Finally, we enrich $\mathcal{L}^a$ allowing classes with many parents, and obtain $\mathcal{L}^m$ corresponding to a language with both interfaces and multiple inheritance. We propose a sound and complete overloading framework for $\mathcal{L}^a$ and briefly compare $\mathcal{L}^a$ with overloading resolution in C++.

4.1 Single inheritance

Underlying language framework The framework $\mathcal{L}^s = (\mathsf{P}^s, \mathsf{M}^s, \mathsf{C}^s, \mathsf{AC}^s, \mathsf{I}^s, \sqsubseteq^s, \mathsf{Dscr}^s, CndMs^s, \mathsf{B}^s, MSel^s, :^s)$ defined below corresponds to a Java-like language with only non-abstract classes.

Assume that p is a given well-formed program. Then:

1. $\mathsf{M}^s_{\mathsf{p}}$ are all method names in p.
2. $\mathsf{C}^s_{\mathsf{p}}$ are all the classes declared in p, and $\mathsf{AC}^s_{\mathsf{p}} = \mathsf{I}^s_{\mathsf{p}} = \emptyset$.
3. The subtyping relation $\sqsubseteq^s_{\mathsf{p}}$ is the reflexive and transitive closure of the `extends` relation. Each class extends either one or no parent (for simplicity, we do not consider predefined classes like Object), and $\sqsubseteq^s_{\mathsf{p}}$ is acyclic (and hence a partial order).
4. $\mathsf{Dscr}^s_{\mathsf{p}} = \mathsf{T}^s_{\mathsf{p}} \times (\mathsf{T}^s_{\mathsf{p}})^* \times \mathsf{T}^s_{\mathsf{p}}$. The three components are the type containing the method, the argument and the result type, respectively.
5. $\mathsf{B}^s_{\mathsf{p}}$ is the set of (compiled) method bodies in p.[10]
6. If class c defines a body with name m, argument types $\bar{\mathsf{t}}$, return type t and body b, then $\mathsf{b}{:}^s_{\mathsf{p}}\langle\mathsf{c}', \langle\mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle\rangle$ for each $\mathsf{t}_0{\in}\mathsf{T}^s_{\mathsf{p}}$ and $\mathsf{c}' \sqsubseteq^s_{\mathsf{p}} \mathsf{c}$.
7. $CndMs^s_{\mathsf{p}}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$ correspond to methods of t_0 (either directly declared or inherited[11]) with name m and whose argument types are supertypes of $\bar{\mathsf{t}}$.
8. $MSel^s_{\mathsf{p}}(\mathsf{c}, \mathsf{m}, \langle\mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle)$=
   - b, if c declares a method m with argument types $\bar{\mathsf{t}}$, and body b (unique, by assumption (9) below),
   - $MSel^s_{\mathsf{p}}(\mathsf{c}', \mathsf{m}, \langle\mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle))$ otherwise, if c' is the direct superclass of c,
   - none if c has no superclass.
9. No class has two methods (directly declared or inherited) with same name and argument types but different result type.

From now on, we use the following abbreviations: for each subtyping relation $\sqsubseteq$, $\mathsf{t} \sqsubset_{\mathsf{p}} \mathsf{t}'$ iff $\mathsf{t} \sqsubseteq_{\mathsf{p}} \mathsf{t}'$ and $\mathsf{t} \neq \mathsf{t}'$; $\mathsf{t}_1 ... \mathsf{t}_n \sqsubseteq_{\mathsf{p}} \mathsf{t}'_1 ... \mathsf{t}'_n$ iff $\mathsf{t}_i \sqsubseteq_{\mathsf{p}} \mathsf{t}'_i$ for all $1 \leq i \leq n$; $\bar{\mathsf{t}} \sqsubset \bar{\mathsf{t}}'$ iff $\bar{\mathsf{t}} \sqsubseteq_{\mathsf{p}} \bar{\mathsf{t}}'$ and $\exists i$ *s.t.* $\mathsf{t}_i \neq \mathsf{t}'_i$.

Proposition 1 *$\mathcal{L}^s$ is type safe.*

Proof 1 *Take a program $\mathsf{p} \in \mathsf{P}^s$, $\mathsf{c} \in \mathsf{C}^s_{\mathsf{p}}$, $\langle\mathsf{t}^{\mathsf{r}}_0, \mathsf{m}, \bar{\mathsf{t}}^{\mathsf{r}}\rangle \in \mathsf{T}^s_{\mathsf{p}} \times \mathsf{M}^s_{\mathsf{p}} \times (\mathsf{T}^s_{\mathsf{p}})^*$, $\mathsf{c} \sqsubseteq^s_{\mathsf{p}} \mathsf{t}^{\mathsf{r}}_0$.*
Assume that $\mathsf{d}{\in}CndMs^s_{\mathsf{p}}(\mathsf{t}^{\mathsf{r}}_0, \mathsf{m}, \bar{\mathsf{t}}^{\mathsf{r}})$. Let d be $\langle\mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle$. From the definition of $CndMs^s_{\mathsf{p}}$ in (7), t_0 declares a method with name m, argument types $\bar{\mathsf{t}}$ and result type t, and $\mathsf{t}^{\mathsf{r}}_0 \sqsubseteq^s_{\mathsf{p}} \mathsf{t}_0$. Hence $\mathsf{c} \sqsubseteq^s_{\mathsf{p}} \mathsf{t}_0$. Then, from the definition of $\sqsubseteq^s_{\mathsf{p}}$ in (3), c inherits from some c' s.t. $\mathsf{c} \sqsubseteq^s_{\mathsf{p}} \mathsf{c}' \sqsubseteq^s_{\mathsf{p}} \mathsf{t}_0$ a method with name m and argument types $\bar{\mathsf{t}}$; from (9), this method has return type t. Let b be the body of this method. Then, $MSel^s_{\mathsf{p}}(\mathsf{c}, \mathsf{m}, \mathsf{d}) = \mathsf{b}$ and, from (6), $\mathsf{b}{:}^s_{\mathsf{p}}\langle\mathsf{c}, \mathsf{d}\rangle$. □

[9]This means that $\preceq$ is also anti-symmetric, hence the induced equivalence is equality.

[10]"Compiled method bodies" means that method calls are enriched with descriptors, and field accesses are enriched with class information; this is necessary for assumption (6), and also in any proof of type soundness of the language.

[11]As already noticed, in Java inherited means not overridden. However, unless explicitly indicated, in our formal treatment it makes no difference if overridden methods are included.

The proof above, and the others in the sequel, are not, of course, complete proofs of the safety of the type system of the corresponding language; indeed, the notion of language framework introduced in this paper abstracts from many features only focusing on overloading resolution and run-time selection. However, the result of type safety of the language framework can be considered as the kernel of a proof of type soundness for the full language, as those provided, *e.g.*, for Java, in [9, 16, 18]. In particular, the type system should guarantee that $\mathsf{b}:^s_{\mathsf{p}}\langle \mathsf{c}, \langle \mathsf{t}_0, \mathsf{t}_1...\mathsf{t}_n, \mathsf{t}\rangle\rangle$ holds if b is a method body with n arguments, and replacing the receiver by an object of class c, and the arguments with values of subtypes of $\mathsf{t}_1...\mathsf{t}_n$, gives a value of a subtype of t.

We show a simple condition, sufficient for soundness and completeness of more specific relations in $\mathcal{L}^s$.

Proposition 2 *The semantic equivalence $\approx^{\mathsf{t}_0}_{\mathsf{p}}$ on $\mathsf{Dscr}^s_{\mathsf{p}}$ coincides with equality of argument types.*

Proof 2 *From the definition of $MSel^s$, which only depends on argument types.* $\square$

Overloading frameworks We define three overloading frameworks with $\mathcal{L}^s$ as the underlying language framework, which correspond to the three rules informally introduced in section 2.2.

- $\mathcal{O}^{s,j} = \langle \mathcal{L}^s, \preceq^j\rangle$ corresponds to the Java rule,
- $\mathcal{O}^{s,a1} = \langle \mathcal{L}^s, \preceq^{a1}\rangle$ describes the first alternative,
- $\mathcal{O}^{s,a2} = \langle \mathcal{L}^s, \preceq^{a2}\rangle$ describes the second alternative.

Definition 7 *For any* $\mathsf{p}\in\mathsf{P}$, $\langle \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle$, $\langle \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle\in\mathsf{Dscr}^s_{\mathsf{p}}$,

- $\langle \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle \preceq^j_{\mathsf{p}} \langle \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$ *iff* $\bar{\mathsf{t}} \sqsubseteq^s_{\mathsf{p}} \bar{\mathsf{t}}'$ *and* $\mathsf{t}_0 \sqsubseteq^s_{\mathsf{p}} \mathsf{t}'_0$,
- $\langle \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle \preceq^{a1}_{\mathsf{p}} \langle \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$ *iff* $\bar{\mathsf{t}} \sqsubseteq^s_{\mathsf{p}} \bar{\mathsf{t}}'$,
- $\langle \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle \preceq^{a2}_{\mathsf{p}} \langle \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$ *iff* $\bar{\mathsf{t}} \sqsubseteq^s_{\mathsf{p}} \bar{\mathsf{t}}'$ *and* $\mathsf{t}_0 = \mathsf{t}'_0$,
  or $\mathsf{t}_0 \sqsubset^s_{\mathsf{p}} \mathsf{t}'_0$.

The three more specific relations, $\preceq^j_{\mathsf{p}}$, $\preceq^{a1}_{\mathsf{p}}$ and $\preceq^{a1}_{\mathsf{p}}$, are well-defined, since they are pre-orders on the whole set of descriptors Dscr^s. Note that, if overridden methods are not considered, then the three pre-orders correspond to different ways of combining the subtype relations on the first (declaring class) and second (argument types) component of Dscr^s, that is, componentwise, from right to left (which only considers the argument types), and from left to right.

The relations $\preceq^j_{\mathsf{p}}$, $\preceq^{a1}_{\mathsf{p}}$ and $\preceq^{a1}_{\mathsf{p}}$ are defined on the same set of descriptors Dscr^s. This allows a better comparison. However, the first component in Dscr^s is not required for $\mathcal{L}^s$, nor for the first alternative, $\preceq^{a1}$, but *is* required for $\preceq^j$ and $\preceq^{a2}$. This reflects the fact that $\preceq^{a1}$ has the simplest copy semantics.

We show that the three more specific relations are sound and complete.

Lemma 3 *For any* $\mathsf{p}\in\mathsf{P}$, *if* $\mathsf{d} = \langle \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle, \mathsf{d}' = \langle \mathsf{t}'_0, \bar{\mathsf{t}}, \mathsf{t}'\rangle \in \mathsf{Dscr}^s_{\mathsf{p}}$ *are two candidates for some call in* p, *then* $\mathsf{t} = \mathsf{t}'$ *and either* $\mathsf{t}_0 \sqsubseteq^s_{\mathsf{p}} \mathsf{t}'_0$ *or conversely.*

Proof 3 *By the definition of candidates in (7), the fact that each class has at most one parent class (3) and the assumption in (9).* $\square$

Proposition 3 $\preceq^j$, $\preceq^{a1}$ *and* $\preceq^{a2}$ *are sound and complete w.r.t.*$\mathcal{L}^s$.

Proof 4 $\preceq^j$: *It is easy to see that, for any* $\mathsf{p} \in \mathsf{P}^s$, *the equivalence* $\equiv^{\preceq^j}_{\mathsf{p}}$ *on maximally specific candidates is equality, since by assumptions (3) and (9),* $\preceq^j_{\mathsf{p}}$ *is a partial order over candidates for a given call. Hence soundness follows from lemma 2. For completeness, assume that* $\mathsf{d} = \langle \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle$ *and* $\mathsf{d}' = \langle \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$ *are two semantically equivalent maximally specific candidates. From Prop.2,* $\bar{\mathsf{t}} = \bar{\mathsf{t}}'$ *and from Lemma 3,* $\mathsf{t} = \mathsf{t}'$ *and either* $\mathsf{t}_0 \sqsubseteq^s_{\mathsf{p}} \mathsf{t}'_0$, *or conversely. Hence* $\mathsf{d} \preceq^j_{\mathsf{p}} \mathsf{d}'$ *or conversely hence, from Lemma 1, we can conclude that* $\mathsf{d} \equiv^{\preceq^j}_{\mathsf{p}} \mathsf{d}'$. $\square$

$\preceq^{a1}$: *Equivalence* $\equiv^{\preceq^{a1}}_{\mathsf{p}}$ *is (by assumption (3)) equality of argument types. The rest follows from Prop.2.* $\square$

$\preceq^{a2}$: *The same proof as for* $\preceq^j$ *applies.* $\square$

We show now that the alternative rules are strictly more flexible than the Java rule.

Lemma 4 *If* $\preceq$ *is a pre-order over a set* Ds, *and there exists* $x\in\mathsf{Ds}$, *s.t.* $x \preceq y$ $\forall y\in\mathsf{Ds}$, *then* x *is a minimal element and any other minimal element is equivalent to* x.

Proposition 4 $\preceq^{a1}$ *and* $\preceq^{a2}$ *are strictly more flexible than* $\preceq^j$ *w.r.t.*$\mathcal{L}^s$.

Proof 5 $\preceq^{a1}$: *First we show that* $\preceq^{a1}$ *is more flexible than* $\preceq^j$. *Consider an unambiguous call w.r.t.* $\preceq^j_{\mathsf{p}}$ *in a program* p. *Since* $\preceq^j_{\mathsf{p}}$ *is a partial order, there exists exactly one descriptor, say* d, *which is more specific than all the candidates w.r.t.* $\preceq^j_{\mathsf{p}}$. *Since* $\preceq^j_{\mathsf{p}}$ *implies* $\preceq^{a1}_{\mathsf{p}}$, d *is more specific than all the candidates w.r.t.* $\preceq^{a1}_{\mathsf{p}}$ *as well, By Lemma 4* d *is maximally specific for* $\preceq^{a1}_{\mathsf{p}}$ *(hence the second condition in definition 6 is satisfied) and the call is unambiguous w.r.t.* $\preceq^{a1}_{\mathsf{p}}$ *(hence the first condition in definition 6 is satisfied). To show that there are calls which are ambiguous w.r.t.* $\preceq^j$ *and unambiguous w.r.t.* $\preceq^{a1}$, *consider the example 1 in section 2.2, where the candidates for the call* $\mathsf{c.m(b')}$ *are* $\langle \mathsf{A}, \mathsf{B}', \mathsf{int}\rangle$ *and* $\langle \mathsf{B}, \mathsf{A}', \mathsf{int}\rangle$. $\square$

$\preceq^{a2}$: *To show that* $\preceq^{a2}$ *is more flexible than* $\preceq^j$, *since they are partial orders, it is enough to note that* $\preceq^j_{\mathsf{p}}$ *implies* $\preceq^{a2}_{\mathsf{p}}$. *To show that there are calls which are ambiguous w.r.t.* $\preceq^j$ *and unambiguous w.r.t.* $\preceq^{a1}$, *consider again example 1.* $\square$

Finally, the alternatives $\preceq^{a1}$ and $\preceq^{a2}$ are not comparable: There exist calls which are ambiguous *w.r.t.* to $\preceq^{a1}$ and unambiguous *w.r.t.* $\preceq^{a2}$, and conversely, *c.f.* App. D and E.

4.2 Single inheritance with abstract methods

We enrich $\mathcal{L}^s$ with abstract classes and interfaces, obtaining framework $\mathcal{L}^a = (\mathsf{P}^a, \mathsf{M}^a, \mathsf{C}^a, \mathsf{AC}^a, \mathsf{I}^a, \sqsubseteq^a, \mathsf{Dscr}^a, \mathit{CndMs}^a, \mathsf{B}^a, \mathit{MSel}^a, :^a)$. We only list the components and assumptions which are different from $\mathcal{L}^s$.

2. AC^a_p and I^a_p are abstract classes and interfaces declared in p, respectively.

3. The subtyping relation $\sqsubseteq^a_\mathsf{p}$ is the reflexive and transitive closure of the union of the `extends` and `implements` relations. Same assumptios as for $\sqsubseteq^s_\mathsf{p}$.

4. $\mathsf{Dscr}^a_\mathsf{p} = \mathsf{Kind}^a \times \mathsf{T}^a_\mathsf{p} \times (\mathsf{T}^a_\mathsf{p})^* \times \mathsf{T}^a_\mathsf{p}$, with $\mathsf{Kind}^a = \{\mathsf{a}, \neg\mathsf{a}\}$; a, $\neg\mathsf{a}$ indicate abstract and non-abstract methods.

10. For each $\mathsf{c} \in (\mathsf{C}^s_\mathsf{p} \setminus \mathsf{AC}^a_\mathsf{p})$, if $\mathsf{c} \sqsubseteq^a_\mathsf{p} \mathsf{t}$ and t declares an abstract method, then c directly declares or inherits a non-abstract method with the same name and argument types (which has the same return type by assumption (9)).

Proposition 5 *$\mathcal{L}^a$ is type safe.*

Proof 6 *The proof is analogous to that for Prop.1. Here we apply (10) which ensures that, since $\mathsf{c} \sqsubseteq^a_\mathsf{p} \mathsf{t}_0$, c declares or inherits one non-abstract method with name m and argument types $\bar{\mathsf{t}}$.* □

We extend the three more specific relation from section 4.1 to $\mathsf{Dscr}^a_\mathsf{p}$, in the trivial way, by disregarding the kind component. Then, the first alternative, $\preceq^j$, is still sound and complete, but $\preceq^j$ and $\preceq^{a2}$ are sound but in-complete.

Proposition 6 *1. $\preceq^{a1}$ is sound and complete w.r.t. $\mathcal{L}^a$.*

2. $\preceq^j$ and $\preceq^{a2}$ are sound, but not complete w.r.t. $\mathcal{L}^a$.

Proof 7 *1. Proof is same as for $\mathcal{L}^s$, in Prop.3.* □
2. The proof of soundness is the same as for $\mathcal{L}^s$, in Prop.3. To show that completeness does not hold, assume a program p as in the counterexample 2 in section 2.3, containing I, $\mathsf{I1}$, $\mathsf{I2}$ and some classes implementing I. Then $\langle \mathsf{a}, \mathsf{I1}, (), \mathsf{int}\rangle \not\preceq^j_\mathsf{p} \langle \mathsf{a}, \mathsf{I2}, (), \mathsf{int}\rangle$ and $\langle \mathsf{a}, \mathsf{I2}, (), \mathsf{int}\rangle \not\preceq^j_\mathsf{p} \langle \mathsf{a}, \mathsf{I1}, (), \mathsf{int}\rangle$; the same for $\preceq^{a2}_\mathsf{p}$. However, for all c implementing I, $\mathit{MSel}^a_\mathsf{p}(\mathsf{c}, \mathsf{m}, \langle \mathsf{a}, \mathsf{I1}, (), \mathsf{int}\rangle) = \mathit{MSel}^a_\mathsf{p}(\mathsf{c}, \mathsf{m}, \langle \mathsf{a}, \mathsf{I2}, (), \mathsf{int}\rangle)$. □

To achieve completeness, $\preceq^j$ and $\preceq^{a2}$ need to be modified, so that no ambiguity arises from different abstract methods with same argument types. For this, kind information will be used, although $\preceq^{a1}$ considers only the argument types.

Definition 8 *For* $\mathsf{p} \in \mathsf{P}^a$, $\langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle, \langle \mathsf{k}', \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle \in \mathsf{Dscr}^a_\mathsf{p}$:

- $\langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle \preceq^{j+}_\mathsf{p} \langle \mathsf{k}', \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$ *iff*
  (1) $\mathsf{t}_0 \sqsubseteq^a_\mathsf{p} \mathsf{t}'_0$ *and* $\bar{\mathsf{t}} \sqsubseteq^s_\mathsf{p} \bar{\mathsf{t}}'$ *or*
  (2) $\bar{\mathsf{t}} = \bar{\mathsf{t}}'$ *and* $\mathsf{t}'_0 \not\sqsubseteq^a_\mathsf{p} \mathsf{t}_0$ *and* $\mathsf{k}' = \mathsf{a}$
- $\langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle \preceq^{a2+}_\mathsf{p} \langle \mathsf{k}', \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$ *iff*
  (1) $\mathsf{t}_0 \sqsubseteq^a_\mathsf{p} \mathsf{t}'_0$ *or* ($\mathsf{t}_0 = \mathsf{t}'_0$ *and* $\bar{\mathsf{t}} \sqsubseteq^a_\mathsf{p} \bar{\mathsf{t}}'$) *or*
  (2) $\bar{\mathsf{t}} = \bar{\mathsf{t}}'$ *and* $\mathsf{t}'_0 \not\sqsubseteq^a_\mathsf{p} \mathsf{t}_0$ *and* $\mathsf{k}' = \mathsf{a}$.

Def.8 extends Def.7; we added case (2), which makes a method more specific than another, when they have the same argument types and the latter is abstract (unless declared in a heir class). The definition of $\preceq^{j+}$ corresponds to overloading resolution as in the revised Java specification [4], in 15.12.2.2. We prove now that the more specific relations defined above are well-defined, sound and complete.

Lemma 5 *For any $\mathsf{p} \in \mathsf{P}^a$, if $\mathsf{d} = \langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle, \mathsf{d}' = \langle \mathsf{k}', \mathsf{t}'_0, \bar{\mathsf{t}}, \mathsf{t}'\rangle \in \mathsf{Dscr}^a_\mathsf{p}$ are two candidates for some call in p, then $\mathsf{t} = \mathsf{t}'$ and either $\mathsf{t}_0 \sqsubseteq^a_\mathsf{p} \mathsf{t}'_0$ or conversely, or at least one of $\{\mathsf{k}, \mathsf{k}'\}$ is a.*

Proof 8 *By the definition of candidate methods in (7), the fact that each class has at most one parent class (3) and assumption (9).* □

Proposition 7 *$\preceq^{j+}$ and $\preceq^{a2+}$ are well-defined, sound and complete w.r.t. $\mathcal{L}^a$.*

Proof 9 *We give the proof for $\preceq^{j+}$, the other case is analogous.*
Take any $\mathsf{d}, \mathsf{d}', \mathsf{d}'' \in \mathit{MaxMs}^{\preceq^{j+}}_\mathsf{p}(\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}})$ for some $\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}$. Assume that $\mathsf{d} = \langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle, \mathsf{d}' = \langle \mathsf{k}', \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$, and $\mathsf{d}'' = \langle \mathsf{k}'', \mathsf{t}''_0, \bar{\mathsf{t}}'', \mathsf{t}''\rangle$.

We first prove that $\preceq^{j+}$ is a pre-order on the set of maximally specific candidates, i.e., that it is transitive. Assume that $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}' \preceq^{j+}_\mathsf{p} \mathsf{d}''$. If $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}'$ holds by (1), then it must be $\mathsf{d} = \mathsf{d}'$ (since $\mathsf{d} \equiv^{\preceq^{j+}}_\mathsf{p} \mathsf{d}'$ holds by Lemma 1), hence $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}''$. Analogously, if $\mathsf{d}' \preceq^{j+}_\mathsf{p} \mathsf{d}''$ holds by (1), then it must be $\mathsf{d}' = \mathsf{d}''$, hence $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}''$. Finally, if both $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}'$ and $\mathsf{d}' \preceq^{j+}_\mathsf{p} \mathsf{d}''$ hold by (2), then the thesis holds since either $\mathsf{t}''_0 \not\sqsubseteq^a_\mathsf{p} \mathsf{t}_0$ (hence $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}''$ by (2)), or $\mathsf{d}'' \preceq^{j+}_\mathsf{p} \mathsf{d}$ hence $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}''$ too must hold by Lemma 1.
It is easy to see that $\equiv^{\preceq^{j+}}_\mathsf{p}$ can be characterized as follows:
  $\mathsf{d} \equiv^{\preceq^{j+}}_\mathsf{p} \mathsf{d}'$ *iff*
  $\bar{\mathsf{t}} = \bar{\mathsf{t}}'$ *and* ($\mathsf{t}_0 = \mathsf{t}'_0$ *or* $\mathsf{t}_0 \not\sqsubseteq^a_\mathsf{p} \mathsf{t}'_0, \mathsf{t}'_0 \not\sqsubseteq^a_\mathsf{p} \mathsf{t}_0, \mathsf{k} = \mathsf{k}' = \mathsf{a}$)
Then, soundness follows from Prop.2. For completeness, assume that d and d' are two semantically equivalent maximally specific candidates, hence, from Prop.2, $\bar{\mathsf{t}} = \bar{\mathsf{t}}'$ and, from lemma 5, $\mathsf{t} = \mathsf{t}'$ and either $\mathsf{t}_0 \sqsubseteq^a_\mathsf{p} \mathsf{t}'_0$ or conversely, or at least one of $\{\mathsf{k}, \mathsf{k}'\}$ is a. In the first case, $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}'$ or conversely, hence we can conclude by Lemma 1; in the second case, if, e.g., $\mathsf{k}' = \mathsf{a}$, then $\mathsf{d} \preceq^{j+}_\mathsf{p} \mathsf{d}'$ and the thesis follows analogously. □

4.3 Multiple inheritance

We define an overloading framework $\mathcal{O}^m = \langle \mathcal{L}^m, \preceq^m \rangle$ for a paradigmatic language supporting both multiple inheritance and abstract methods, as shown in section 2.4. This framework is an extension of $\mathcal{L}^a$, with which we explore the new issues introduced when dropping the assumption that a class extends at most one parent. We also introduce the distinction between virtual and non-virtual methods which was not significant in the preceding frameworks. Method selection, consequently, is dynamic only for abstract or virtual methods, while for non-virtual methods the selected body corresponds to the descriptor determined by overloading resolution.

$\mathcal{L}^m$ differs from $\mathcal{L}^a$ in that:

3. A class can have many parent classes.

4. $\mathsf{Dscr}^m_\mathsf{p} = \mathsf{Kind}^m \times \mathsf{T}^m_\mathsf{p} \times (\mathsf{T}^m_\mathsf{p})^* \times \mathsf{T}^m_\mathsf{p}$, $\mathsf{Kind}^m = \{\mathsf{a}, \mathsf{v}, \neg\mathsf{v}\}$.

8. $\mathit{MSel}^m_\mathsf{p}(\mathsf{c}, \mathsf{m}, \langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle)$=

   - if $\mathsf{k} = \neg\mathsf{v}$, then
     - b if t_0 declares a method m with argument types $\bar{\mathsf{t}}$ and body b,
     - none otherwise;
   - otherwise,
     - b, if c declares a virtual method m with argument types $\bar{\mathsf{t}}$ and body b (unique by assumption (9)),
     - b, if $\mathit{MSel}^m_\mathsf{p}(\mathsf{c}', \mathsf{m}, \langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle)) = \mathsf{b}$ for some c' direct superclass of c, and this definition does not depend on the choice of c'[12]
     - none otherwise.

10. For each $\mathsf{c} \in (\mathsf{C}^s_\mathsf{p} \setminus \mathsf{AC}^a_\mathsf{p})$, if $\mathsf{c} \sqsubseteq^a_\mathsf{p} \mathsf{t}$ and t declares an either virtual or abstract method, then c directly declares or inherits *exactly one* non-abstract method with the same name and argument types (which has the same return type by assumption (9)).

Proposition 8 *$\mathcal{L}^m$ is type safe.*

Proof 10 *The proof is analogous to those provided for $\mathcal{L}^s$ (Prop.1) and $\mathcal{L}^a$ (Prop.5); the only difference is that, in the case $\mathsf{k} \in \{\mathsf{a}, \mathsf{v}\}$, we have to apply the new version of (10). In the case $\mathsf{k} = \neg\mathsf{v}$, method selection directly gives the method body corresponding to d, and the thesis holds trivially.* $\square$

We define sound and complete more specific relation $\preceq^m$:

Definition 9 *For* $\mathsf{p} \in \mathsf{P}^m$, $\langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle$, $\langle \mathsf{k}', \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle \in \mathsf{Dscr}^m_\mathsf{p}$, $\langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle \preceq^m_\mathsf{p} \langle \mathsf{k}', \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$ *iff*

$$
\begin{array}{ll}
(1)\ \bar{\mathsf{t}} \sqsubset^m_\mathsf{p} \bar{\mathsf{t}}' & \textit{or} \\
(2)\ \bar{\mathsf{t}} = \bar{\mathsf{t}}'\ \textit{and} & \\
\quad (\mathsf{t}_0 \sqsubseteq^m_\mathsf{p} \mathsf{t}'_0\ \textit{or}\ (\mathsf{t}'_0 \not\sqsubseteq^m_\mathsf{p} \mathsf{t}_0, \mathsf{k}' = \mathsf{a},\ \mathsf{k} \neq \neg\mathsf{v})). &
\end{array}
$$

[12] Thus, method selection would return none, if two superclasses which were not subclasses of each other contained a method body with appropriate argument types.

Informally, a method is more specific than another iff either (1) it has (strictly) more specific argument types or (2) they have the same argument types and either the former overrides the latter, or the latter is abstract (and does not override the former) and the former is either abstract or virtual. We can prove that the more specific relation defined above is well-defined, sound and complete.

We first give a sufficient condition which guarantees soundness and completeness for overloading resolution in $\mathcal{L}^m$, analogous to that provided for $\mathcal{L}^s$ in Prop.2. Since we have introduced non-virtual methods, two descriptors are now semantically equivalent if either they are both abstract or virtual and they have the same argument types (since in this case method selection only depends on argument types), or they are both non-virtual and they have the same argument types and are declared in the same class (that is, are the same method) .

Proposition 9 *For any* $\mathsf{p} \in \mathsf{P}^m$, *and any* $\mathsf{t}_0, \mathsf{m}, \bar{\mathsf{t}}$: $\langle \mathsf{k}, \mathsf{t}_0, \bar{\mathsf{t}}, \mathsf{t}\rangle \approx^{\mathsf{t}_0}_\mathsf{p} \langle \mathsf{k}', \mathsf{t}'_0, \bar{\mathsf{t}}', \mathsf{t}'\rangle$ *iff* $\bar{\mathsf{t}} = \bar{\mathsf{t}}'$ *and* $(\mathsf{k}, \mathsf{k}' \neq \neg\mathsf{v}$ *or* $\mathsf{t}_0 = \mathsf{t}'_0)$.

Proof 11 *From the definition of MSel^s.* $\square$

Proposition 10 *$\preceq^m$ is well-defined, sound and complete w.r.t.$\mathcal{O}^m$.*

Proof 12 *The proof that $\preceq^m$ is a pre-order on the set of maximally specific candidates can be done by case analysis, analogously to that provided for $\preceq^{j+}$ in Prop.7. It is easy to see that $\equiv^{\preceq^m}_\mathsf{p}$ can be characterized as follows:*
$\mathsf{d} \equiv^{\preceq^m}_\mathsf{p} \mathsf{d}'$ *iff*
$\bar{\mathsf{t}} = \bar{\mathsf{t}}'$ *and* $((\mathsf{t}_0 = \mathsf{t}'_0)$ *or* $(\mathsf{t}_0 \not\preceq^m_\mathsf{p} \mathsf{t}'_0, \mathsf{t}'_0 \not\preceq^m_\mathsf{p} \mathsf{t}_0, \mathsf{k} = \mathsf{k}' = \mathsf{a}))$.
This condition implies $\bar{\mathsf{t}} = \bar{\mathsf{t}}'$ and $(\mathsf{k}, \mathsf{k}' \neq \neg\mathsf{v}$ or $\mathsf{t}_0 = \mathsf{t}'_0)$, hence soundness follows by Prop.9. For completeness, we have to show the converse. Indeed, if d, d' are semantically equivalent maximally specific candidates, then either $\mathsf{t}_0 = \mathsf{t}'_0$, hence $\mathsf{d} \equiv^{\preceq^m}_\mathsf{p} \mathsf{d}'$, or $\mathsf{k}, \mathsf{k}' \neq \neg\mathsf{v}$. In this case: if $\mathsf{t}_0 \sqsubseteq^m_\mathsf{p} \mathsf{t}'_0$ then $\mathsf{d} \preceq^m_\mathsf{p} \mathsf{d}'$, hence $\mathsf{d} \equiv^{\preceq^m}_\mathsf{p} \mathsf{d}'$ by Lemma 1, and analogously if $\mathsf{t}'_0 \sqsubseteq^m_\mathsf{p} \mathsf{t}_0$; if $\mathsf{d} \not\preceq^m_\mathsf{p} \mathsf{d}', \mathsf{d}' \not\preceq^m_\mathsf{p} \mathsf{d}$, then either $\mathsf{k} = \mathsf{k}' = \mathsf{a}$, hence $\mathsf{d} \equiv^{\preceq^m}_\mathsf{p} \mathsf{d}'$, or, e.g., $\mathsf{k} = \mathsf{v}$, but in this case it must be $\mathsf{k}' = \mathsf{a}$ for assumption (10), hence $\mathsf{d} \preceq^m_\mathsf{p} \mathsf{d}'$, hence $\mathsf{d} \equiv^{\preceq^m}_\mathsf{p} \mathsf{d}'$ by Lemma 1. $\square$

Comparison with C++ The motivation for $\mathcal{O}^m$ came from multiple inheritance as in C++. However, because we had $\mathcal{L}^a$, we developed $\mathcal{O}^m$ as an extension of $\mathcal{L}^a$. Therefore, $\mathcal{O}^m$ and C++ are similar in some aspects, and different in others:

Both $\mathcal{O}^m$ and C++ adopt the arguments-only rule and assumptions analogous to those in (10). However, C++ and $\mathcal{O}^m$ differ in the following aspects: C++ candidate

sets do not contain methods from a superclass if the current class contains a method definition with the same name (see App.B). Also, $\preceq^m$ in $\mathcal{O}^m$ is complete, whereas in C++ the conflict between several inherited abstract methods makes a call ambiguous *c.f.* example 2. Last, $\mathcal{O}^m$ does not model many C++ specific features *e.g.*, using-clauses, private members, non-virtual inheritance.
We plan extensions of the model to deal with these issues.

5 Applicability of the frameworks

In our frameworks static resolution is modelled by the function $CndMs$, which defines candidates for method calls, and by the relation $\preceq$ which selects maximally specific candidates. Dynamic resolution is modeled by the function $MSel$.

In this paper we focused on different more specific relations, and therefore kept the definitions of $CndMs$ and $MSel$ fixed. However, the framework can be adapted to represent many more language features:
• Private members are not inherited (but may be redefined) in subclasses. For that, Kind should also account for access modifiers. The domain of $CndMs$ would be extended to: $\mathsf{C_p} \times \mathsf{T_p} \times \mathsf{M_p} \times \mathsf{T_p^*}$ and the first argument, representing the class containing the method call, would be used to filter out private methods. $MSel$ would start looking-up in the first superclass of the dynamic receiver's class which inherits the method from the descriptor – *c.f.* [2] for details.
• Contravariant overriding, whereby a method overrides a parent's method with more specific argument and less specific return types, can be expressed by suitable definitions of $CndMs$, and of $MSel$.
• Multimethods would require the $MSel$ function to take into account the dynamic class of all, or some, arguments, as well as that of the receiver.

In further work we plan to analyze these possibilities, through different extensions of our framework.

6 Conclusions

We have discussed cases where programming languages treat the interaction between overloading resolution and inheritance in counterintuitive and restrictive ways.

In order to clarify the issues involved in this interaction, we have developed a formal notion of *overloading framework* which describes static and dynamic function selection, abstracting from other language features. We have defined the following properties of an overloading framework:

- *type safety*: guarantees that all candidate method descriptors for a given call lead at run-time to a correct body,
- *soundness*: guarantees that all method descriptors which have distinct dynamic semantics are distinguished by overloading resolution,
- *completeness*: guarantees that all method descriptors which are distinguished by overloading resolution have distinct dynamic semantics,
- *flexibility*: allows as many calls as sensible to be unambiguous.

We have demonstrated that Java overloading resolution in the first 1996 version [11] is type safe, sound but not complete, while in the 2000 version [4] it is complete but not as flexible as could be. We have suggested two alternative approaches, which are type safe, sound, complete and more flexible than those of Java. We have also defined a sound and complete framework for overloading and inheritance in languages with multiple inheritance.

The main outcome of our work is the definition of a framework in which to study in a formal way the interaction between static overloading resolution (*overloading* in the terminology of this paper) and dynamic overloading resolution (*overriding*), abstracting from other language features. With such frameworks one can express the "overall idea" of the languages design, which gets lost in the many details of complete systems as usually described through operational semantics, type systems, and subject reduction proofs.

In this paper we have focused on different ways of defining the more specific method, but through modification of other components of the framework ($CndMs$, $MSel$, Dscr) we shall be able to characterize other features, such as private methods, multimethods, non-virtual inheritance *etc.*

Finally, in the paper we have informally shown that various overloading policies correspond to different copy semantics for inheritance. We have also seen that the systems with best properties are those with the simplest copy semantics. It is an interesting open question in how far the copy semantics could be automatically derived from the overloading and method selection definition.

Acknowledgments

We are grateful to Giovanni Lagorio for useful discussions, and to the anonymous FOOL'8 referees for their suggestions. This work was partially supported by Murst - TOSCA Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi, APPlied SEMantics - Esprit Working Group 26142 and by the EPSRC (Grant Ref: GR/L 76709).

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag, New York, 1996.

[2] D. Ancona and E. Zucca. True modules for Java classes. Technical Report DISI-TR-00-12, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, August 2000. Submitted for publication.

[3] D. Ancona, E. Zucca, and S. Drossopoulou. Overloading and inheritance in Java (extended abstract). In *2th Workshop on Formal Techniques for Java Programs*, June 2000.

[4] G. Bracha, J. Gosling, B. Joy, and G. Steele. *The Java^{TM} Language Specification, Second Edition*. Addison-Wesley, 2000.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

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

[7] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, number 615 in Lecture Notes in Computer Science, Berlin, 1992. Springer Verlag.

[8] W. Cook. A proposal for making Eiffel type-safe. In S. Cook, editor, *ECOOP '87*, pages 57–70, Nottingham, July 1989. Cambridge University Press.

[9] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer Verlag, Berlin, 1999.

[10] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. In *Nordic Journal of Computing 1(1)*, pages 3–37, 1994.

[11] J. Gosling, B. Joy, and G. Steele. *The Java^{TM} Language Specification*. Addison-Wesley, 1996.

[12] S.C. Keene. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, 1989.

[13] Bertrand Meyer. Static typing and other mysteries of life. Electronic note available at `http://http://www.eiffel.com`.

[14] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachussetts, 1990.

[15] B. Stroustrup. *The C++ Programming Language*. Reading. Addison-Wesley, MA, USA, special edition, 2000.

[16] D. Syme. Proving Java type sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 83–118. Springer Verlag, 1999.

[17] US Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815 A.

[18] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 119–156. Springer Verlag, 1999.

A A program in an overloading framework

The following sets, functions and predicates come out of the application of the $\mathcal{O}^{s,j}$ overloading framework to program p_0, which stands for the first example in section 2.1:

- $\mathsf{M}_{\mathsf{p}_0}^{s} = \{\mathsf{m}\}$,
- $\mathsf{C}_{\mathsf{p}_0}^{s} = \{\mathsf{A}, \mathsf{B}, \mathsf{A}', \mathsf{B}', \mathsf{C}'\}$, $\mathsf{I}_{\mathsf{p}_0}^{s} = \emptyset = \mathsf{AC}_{\mathsf{p}_0}^{s}$,
- $\mathsf{B} \sqsubseteq_{\mathsf{p}_0}^{s} \mathsf{A}$, $\mathsf{B}' \sqsubseteq_{\mathsf{p}_0}^{s} \mathsf{A}'$, $\mathsf{C}' \sqsubseteq_{\mathsf{p}_0}^{s} \mathsf{B}'$,
- $\mathsf{Dscr}_{\mathsf{p}_0}^{s} = \{\langle \mathsf{A}, \mathsf{A}', \mathsf{int}\rangle, \langle \mathsf{B}, \mathsf{B}', \mathsf{int}\rangle\}$
- $\mathit{CndMs}_{\mathsf{p}_0}^{s}(\langle \mathsf{A}, \mathsf{m}, \mathsf{A}\rangle) = \emptyset$,
  $\mathit{CndMs}_{\mathsf{p}_0}^{s}(\langle \mathsf{A}, \mathsf{m}, \mathsf{A}'\rangle) = \{\langle \mathsf{A}, \mathsf{A}', \mathsf{int}\rangle\}$,
  $\mathit{CndMs}_{\mathsf{p}_0}^{s}(\langle \mathsf{B}, \mathsf{m}, \mathsf{B}'\rangle) = \{\langle \mathsf{A}, \mathsf{A}', \mathsf{int}\rangle, \langle \mathsf{B}, \mathsf{B}', \mathsf{int}\rangle\}$,
  etc.
- $:_{\mathsf{p}}^{s} = \{ \{ \text{ return } 1\}, \langle \mathsf{A}, \mathsf{A}', \mathsf{int}\rangle, \{ \text{ return } 2\}, \langle \mathsf{B}, \mathsf{B}', \mathsf{int}\rangle\}$.
- $\langle \mathsf{B}, \mathsf{B}', \mathsf{int}\rangle \preceq_{\mathsf{p}_0}^{j} \langle \mathsf{A}, \mathsf{A}', \mathsf{int}\rangle$
- $\mathit{MSel}_{\mathsf{p0}}^{s}(\mathsf{A}, \langle \mathsf{A}, \mathsf{A}', \mathsf{int}\rangle) = \{ \text{ return } 1\}$
  $\mathit{MSel}_{\mathsf{p0}}^{s}(\mathsf{B}, \langle \mathsf{A}, \mathsf{A}', \mathsf{int}\rangle) = \{ \text{ return } 1\}$
  $\mathit{MSel}_{(}^{\mathsf{p0}}\mathsf{A}, \langle \mathsf{B}, \mathsf{B}', \mathsf{int}\rangle) = \text{none}$
  $\mathit{MSel}_{\mathsf{p0}}^{s}(\mathsf{B}, \langle \mathsf{B}, \mathsf{B}', \mathsf{int}\rangle) = \{ \text{ return } 2\}$

B Overloading resolution in C++

Here we demonstrate in terms of some examples how candidate methods are selected in C++. Candidate methods are all the methods declared (or imported via *using* clauses) in the *first* parent class of c which has *some* method with the same name as in the call. Consider the following example.

Example 4

```
class A {               int m(B' x) return 1; }
class B : public A {    int m(A' x) return 2; }
class C : public B {    }
```

In the call $\mathsf{c.m(b')}$ the method declared in B is selected. This resembles the second alternative, but note that the method in B would be selected even in

```
class A {               int m(B' x) return 1; }
class B : public A {    int m(C' x) return 2; }
class C : public B {    }
```

with $\mathsf{C'}$ arbitrary class; in this case the call would be ill-formed, while the second alternative selects the method declared in A. On the contrary, in

Example 5

```
class A {               int m(B' x) return 1; }
class B : public A {    using A :: m;
                        int m(A' x)  { return 2; }   }
class C : public B {    }
```

in the method call $\mathsf{c.m(b')}$ the method declared in A is selected. This demonstrates that a method is considered more specific than another based on argument types only. Thus, the first alternative is adopted, but on a different set of candidates.

Copy semantics Following the C++ rule, the example 4 would be transformed as follows.

```
class A {               int m(B' x) return 1; }
class B : public A {    int m(A' x) return 2; }
class C : public B {    int m(A' x) return 2; }
```

The example 5 would be transformed as follows.

```
class A {               int m(B' x) return 1; }
class B : public A {    int m(B' x) return 1; }
                        int m(A' x) return 2; } }
class C : public B {    int m(B' x) return 1; }
                        int m(A' x) return 2; } }
```

C The pre-order $\preceq^{j+}_{\mathsf{p}}$

The following example illustrates why $\preceq^{j+}_{\mathsf{p}}$ is not a pre-order on $\mathsf{Dscr_p}$. In fact, $\preceq^{j+}_{\mathsf{p}}$ is not a pre-order on the candidate methods either!

```
abstract class P {    abstract void m(B' x)
                      void m(A' x) { }      }
interface I {         void m(B' x)    }
class H extends P implements I {  }
```

The candidate methods for $\mathsf{h.m(b')}$, with h of type H (remember that $\mathsf{B'}$ is a subclass of $\mathsf{A'}$), are the two methods from P and that from I, *i.e.,* the candidate method descriptors are

$$\{ \langle \mathsf{a}, \mathsf{P}, \mathsf{B'}, \mathsf{void}\rangle,\ \langle \neg\mathsf{a}, \mathsf{P}, \mathsf{A'}, \mathsf{void}\rangle,\ \langle \mathsf{a}, \mathsf{I}, \mathsf{B'}, \mathsf{void}\rangle \}.$$

With $\mathsf{p0}$ standing for the above program, the Java relation $\preceq^{j+}$ from Sect. 4.2 gives:

$$\langle \mathsf{a}, \mathsf{P}, \mathsf{B'}, \mathsf{void}\rangle, \preceq^{j+}_{\mathsf{p0}} \langle \mathsf{a}, \mathsf{I}, \mathsf{B'}, \mathsf{void}\rangle, \quad \text{and}$$
$$\langle \mathsf{a}, \mathsf{I}, \mathsf{B'}, \mathsf{void}\rangle, \preceq^{j+}_{\mathsf{p0}} \langle \mathsf{a}, \mathsf{P}, \mathsf{B'}, \mathsf{void}\rangle, \quad \text{and}$$
$$\langle \mathsf{a}, \mathsf{I}, \mathsf{B'}, \mathsf{void}\rangle, \preceq^{j+}_{\mathsf{p0}} \langle \neg\mathsf{a}, \mathsf{P}, \mathsf{A'}, \mathsf{void}\rangle, \quad \text{but}$$
$$\langle \mathsf{a}, \mathsf{I}, \mathsf{B'}, \mathsf{void}\rangle \not\preceq^{j+}_{\mathsf{p0}} \langle \neg\mathsf{a}, \mathsf{P}, \mathsf{A'}, \mathsf{void}\rangle,$$

therefore $\preceq^{j+}_{\mathsf{p0}}$ is *not* and ordering relation on $\mathsf{Dscr_{p_0}}$.

The set of maximally specific method descriptors however, is only

$$\{ \langle \mathsf{a}, \mathsf{P}, \mathsf{B'}, \mathsf{void}\rangle,\ \langle \mathsf{a}, \mathsf{I}, \mathsf{B'}, \mathsf{void}\rangle \},$$

on which set $\preceq^{j+}_{\mathsf{p0}}$ *is* a pre-order.

D $\preceq^{a1}$ is *not* more flexible than $\preceq^{a2}$

The following example demonstrates that $\preceq^{a1}$ is not more flexible than $\preceq^{a2}$: The call $\mathsf{b.m(b', b')}$ is ambiguous *w.r.t.* $\preceq^{a1}$, but un-ambiguous *w.r.t.* $\preceq^{a2}$ – the method from B would be selected:

```
class A {               void m (A' x, B' y) {  }    }
class B extends A {     void m (B' x, A' y) {  }    }
```

E $\preceq^{a2}$ is *not* more flexible than $\preceq^{a1}$

The following example demonstrates that $\preceq^{a2}$ is not more flexible than $\preceq^{a1}$: The call $\mathsf{b.m(b', b')}$ is ambiguous *w.r.t.* $\preceq^{a2}$, but un-ambiguous *w.r.t.* $\preceq^{a1}$ – the method from A would be selected:

```
class A {               void m (B' x, B' y) {  }    }
class B extends A {     void m (B' x, A' y) {  }
                        void m (A' x, B' y) {  }    }
```