DYNAMIC OVERLOADING WITH COPY SEMANTICS IN OBJECT-ORIENTED LANGUAGES: A FORMAL ACCOUNT *

LORENZO BETTINI¹, SARA CAPECCHI¹ AND BETTI VENNERI²

Abstract. Mainstream object-oriented languages often fail to provide complete powerful features altogether, such as, multiple inheritance, dynamic overloading and copy semantics of inheritance. In this paper we present a core object-oriented imperative language that integrates all these features in a formal framework. We define a static type system and a translation of the language into the meta-language $\lambda_{\text{object}}$, in order to account for semantic issues and prove type safety of our proposal.

Mathematics Subject Classification. 68N15, 68N18, 68Q55.

1. INTRODUCTION

The dynamic flexibility of object-oriented languages is essentially based on the concepts of polymorphism and dynamic binding. The former, together with subtyping and substitutivity, allows the programmer to treat different, but related, objects uniformly. The latter ensures that the right operation is performed when a message is sent to such objects, according to their actual type. The choice of combining dynamic binding with static (polymorphic) typing seems to be the suitable solution to achieve both flexibility and reliability in most popular languages, such as Java [4] and C++ [40].

Keywords and phrases. Object-oriented languages, dynamic overloading, multi-methods, copy semantics, typed lambda calculus, type systems.

* This work has been partially supported by EU Project Software Engineering for Service-Oriented Overlay Computers (SENSORIA, contract IST-3-016004-IP-09) and by MIUR project EOS DUE.

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

Article published by EDP Sciences © EDP Sciences 2009
However, static typing raises some limitations in exploiting dynamic method selection. An interesting case concerns method overloading, i.e., the ability to write methods with the same name, which associate different bodies to different parameter types. In mainstream languages, the interpretation of overloading is static, since the actual code to execute in a method call is selected by the compiler, according to static types of the parameters. Thus polymorphism is not exploited fully (e.g., [22] calls this kind of polymorphism “adhoc”). To overcome this limitation, a common programming practice is to resort to RTTI (run time type information) mechanisms and if statements, in order to explore manually the run-time type of an object, and to type downcasts, in order to force the view of an object according to its run-time representation. These techniques are discouraged by object-oriented design, since they undermine re-usability and evade the constraints of static type checking. Another alternative is to implement double dispatch [28], for example using the Visitor pattern [27]. This solution, however, requires the programmer extra effort since the dynamic dispatch mechanism must be explicitly programmed. Moreover, a cyclic dependence arises from the structure of the pattern, so undermining code re-usability.

Multi-methods [6,15,23,36] naturally support dynamic overloading: the same message name is associated to a set of different bodies (branches) and the run-time selection of the body to invoke depends not only on the actual type of the receiver (single dispatch) but also on the dynamic types of the arguments (multiple dispatch). A major advantage of multi-methods is to enable a safe covariant specialization of methods, where subclasses are allowed to redefine a method by specializing its arguments. There is general evidence that covariant code specialization is an indispensable practice in many situations [10,34]: its most expressive application appears with binary methods [7], i.e., methods that act on objects of the same type as the receiver.

In this paper our main goal is to investigate the interaction between dynamic overloading and (multiple) inheritance in a formal framework, combine them in a theoretical model, and prove type safety of the proposed approach. To this aim we define DOCS (dynamic overloading with copy semantics), a lightweight class-based language that brings basic object-oriented mechanisms into an imperative setting, while adding multiple inheritance and multi-methods. Concerning the interpretation of inheritance, DOCS adopts a copy semantics, in the sense that all the inherited overloaded methods are intended to be directly copied into the subclass. This avoids a too restrictive overloading resolution policy that leads to “strange” compilation errors or unexpected behaviors. We note that the above features, multiple inheritance, multi-methods and copy semantics, are not provided altogether in related approaches (see, e.g., [6,15,21,29,32]).

We base the foundation of our proposal on $\lambda_{\text{object}}$ [15,16], an extended typed lambda-calculus which can be used as a meta language for modeling object oriented features in a functional context. DOCS semantics is defined by translation into $\lambda_{\text{object}}$: this provides a deeper understanding of the integration of dynamic overloading with other features, such as encapsulation (any object encapsulates its methods) and copy semantics. Since $\lambda_{\text{object}}$ is type safe, by proving that every
well typed \textit{DOCS} program is translated in a well typed $\lambda_{\text{object}}$ term, we obtain an indirect proof of static type safety for \textit{DOCS}, i.e., well typed programs will not generate type errors at run time. Moreover, $\lambda_{\text{object}}$ is very close to actual object-oriented programming language implementations: this is particularly evident in that class methods are basically translated into functions that take the receiver of method invocation as a parameter (see, e.g., \cite{31}). Thus, the semantics of \textit{DOCS} into $\lambda_{\text{object}}$ gives practical hints to drive the language implementation according to the formal specification. The semantics of \textit{DOCS} is inspired by Kool \cite{15,16} translation into $\lambda_{\text{object}}$ for what concerns some basic language expressions. Instead, there is a crucial difference in multi-method interpretation, because of the copy semantics characterizing our approach. Furthermore, since translational semantics is type-driven, then differences in typing lead to significant differences between \textit{DOCS} and Kool semantics.

1.1. Design issues

In this section we discuss the main choices that characterize our approach with respect to other proposals. \textit{DOCS} implements dynamic overloading by supporting multi-methods. Indeed a standard method is seen as a particular case of a multi-method with only one branch; thus the distinction between static and dynamic overloading is made at the level of method invocation (\textit{DOCS} provides two different constructs for message sending). This choice increases the flexibility of programming and the reuse of classes: methods are declared in a uniform way and then used through method calls, by choosing the most suitable overloading policy.

Concerning multi-methods, \textit{DOCS} aims at being an integration of different design choices.

We can distinguish two kinds of multi-methods in the literature: 1) the encapsulated multi-methods that preserve the class-based encapsulation property (as in \cite{6}); 2) the view of multi-methods as global \textit{functions external} to classes (as in \cite{17,30,32,38}). In the former approach the principle that an object has a set of associated methods is preserved.

In method selection, dispatch can be asymmetric in the sense that the receiver (and possibly other parameters) has the precedence over other parameters during selection. When the receiver and other parameters participate together in determining the version of the method to be executed we have symmetric dispatch.

Usually encapsulated multi-methods are equated to asymmetric multiple dispatch \cite{7} while, in our opinion, the two aspects are orthogonal. Obviously it seems more natural to have asymmetric dispatch when methods are encapsulated: since there is a privileged receiver, we can think of it as the parameter driving method selection. Instead, there are extensions of single dispatching languages to encapsulated multi-methods that simulate symmetric dispatch \cite{8,21} and languages, such as CLOS \cite{30}, adopting asymmetric dispatch for selecting the right body of generic functions. Indeed, the choice between symmetric or asymmetric dispatch can be also driven by the need for modularity: an order on parameters during method
selection can resolve the ambiguities that arise when linking together software modules defining behaviors for the same message. For instance, asymmetric dispatch is a solution for a language like CLOS, which is dynamically typed, in order to avoid ambiguities at run time; another example is the extension of Java with parasitic methods [6] which gives the precedence to the receiver object in method selection, to obtain modular type checking.

Clearly, method selection with symmetric dispatch is a more flexible mechanism, since it can choose among all the available versions defined for a message name (it does not restrict the choice to the bodies defined in the class of the receiver). The problem is that with symmetric dispatch it is hard to obtain a good compromise between type safety [26,39] and flexibility [32].

The approach chosen for DOCS is to adopt encapsulated multi-methods and symmetric dispatch, while preserving type safety and programming flexibility.

As for the inheritance mechanism, we chose to consider multiple inheritance, since single inheritance is not expressive enough to design class hierarchies and it may lead to duplicate code since it fails in factoring out all the common features shared by classes in a complex hierarchy. Multiple inheritance is rather expressive but it complicates the issue of “ambiguities resolution” already introduced by multiple dispatch. Indeed, considering the interaction between multiple inheritance and multiple dispatch makes our approach more general; the adaptation to the single inheritance case is straightforward, while the other way round is not.

Concerning the interpretation of inheritance, [3] points out that semantics of overloading and inheritance is rather “clean” if it is interpreted through a copy semantics, where all the inherited overloaded methods are intended to be copied into the subclass (apart for those explicitly redefined by the subclass itself). In DOCS, we adopt copy semantics of inheritance as a strategy for the resolution of possible ambiguities due to overloading (not as an implementation technique). We note that the resolution policy for static overloading in Java has followed copy semantics since version 1.4. From a foundational point of view, following Meyer [35], 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) (according to 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 [11] and for specifying the semantics of classes that use traits [25].

DOCS is equipped with a type system in order to model the considered features and their interaction in a type safe way. Indeed the main goal of our approach is to guarantee type safety (other approaches do not enjoy this property [5,14,24]): typing rules ensure that no run time error will occur due to missing or ambiguous branches during dynamic multi-method invocation. The key notion of well formed multi-type (used to type multi-methods) is inspired by [15,16] but differs from it in that it takes into account that copy semantics is implemented in deducing multi-types. As a consequence, for instance, many static ambiguities that could be raised using the rules of [15,16] are automatically resolved in DOCS, in particular when
Listing 1: The gas emission example, where EURO2 is a subtype of EURO1, which is a subtype of EURO0.

single inheritance is involved. In Section 2 we present an example to show how copy semantics combines encapsulated multi-methods with symmetric dispatch.

Finally, we mention that the issue of modularity is out of the scope of the present paper, where we type-check each DOCS program as a single module containing all the classes that are used in it. Indeed modularity is a key problem when integrating multi-methods with symmetric dispatch into a language: modules developed separately, which pass type checking in isolation, may cause type errors when combined. Some works on multi-methods have resolved the problem either by relying on a global type checking [20,32] or by adopting asymmetric dispatch [6,7]. Other approaches are the result of the effort spent on modular type checking in the presence of multi-methods [19,21,33]. Concerning our proposal, the solutions presented in [19,21,33] can be exploited to obtain a type safe composition of DOCS modules.

2. Example

The integration of dynamic overloading in a language increases the complexity of ambiguities resolution during method selection since the set of applicable methods becomes larger when considering also the dynamic type of the parameters. Since type safety is a primary goal for us, the risk is that of adopting a resolution policy that is too strict. To show the advantages of copy semantics we use the example in Listing 1. The class Inspector represents inspectors checking cars gas emission according to European Commission's standards. Cars are classified according to their gas emission in different categories (EURO0, EURO1, EURO2), so the method Inspect is defined on different car types and returns a Report object including information about the inspection. Now let us suppose that, at the time when class Inspector was designed, the available categories were EURO0 and EURO1. In the year 2006 the category EURO2 is added so a subclass Inspector_2006
is defined to deal with these new rules. Finally, in the year 2007 the rules to evaluate older vehicles (EURO0 and EURO1) become stricter so we redefine in a new class Inspector\textsubscript{2007} the inspect method for parameter of type EURO0 and EURO1; instead the rules for EURO2 are still valid so we do not need to redefine the branch for EURO2 in the class Inspector\textsubscript{2007} (we simply inherit it).

Let us consider the semantics of the call Insp.Inspect(car), when Insp and car have dynamic types Inspector\textsubscript{2007} and EURO2, respectively.

Adopting symmetric dispatch (the receiver and the parameter participate together in method selection), the above call should be considered ambiguous. Indeed the version of Inspect defined in class Inspector\textsubscript{2006} is the most specific w.r.t. the argument type, while both versions defined in Inspector\textsubscript{2007} are more specific w.r.t. the receiver type. Should we detect the ambiguity rejecting the definition of method inspect or reject ambiguous method calls? In both cases the code would be rejected because unsafe resulting in a too restrictive policy.

A second alternative is to choose asymmetric dispatch. In this case the receiver has priority in method selection, so the body to execute must be chosen among the versions defined in Inspector\textsubscript{2007}. This way, we partially lose the flexibility of dynamic dispatch, since the version for EURO2 is not considered.

Instead, let us consider an interpretation based on copy semantics: all the method versions defined in the superclass are indented to be copied into the subclass (Listing 2). Thus, the body of inspect, with parameter EURO2 copied in Inspector\textsubscript{2007}, is selected as the most specialized version for both the receiver and the argument.

In summary by the use of copy semantics of inheritance we obtain a symmetric dispatch semantics with encapsulated multi-methods without renouncing static type safety and programming flexibility.
Table 1. Syntax of DOCS: classes, methods and programs.

<table>
<thead>
<tr>
<th>Program definition</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>program ::= classdef * body</td>
<td></td>
</tr>
<tr>
<td>classdef ::= class A : A_{k \in H} {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B_{i} f_{j}; i \in I</td>
</tr>
<tr>
<td></td>
<td>branchdef; j \in J</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>branchdef ::= m branches</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B'<em>{k} (B</em>{k} x_{k}){body_{k}}; k \in K</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>body ::= localdecl; stmt; return exp</td>
<td></td>
</tr>
<tr>
<td>localdecl ::= A_{1} x_{1} = exp_{1}; \ldots; A_{n} x_{n} = exp_{n}</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Syntax of DOCS: expressions and statements.

<table>
<thead>
<tr>
<th>Expression definition</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp ::= x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>this</td>
</tr>
<tr>
<td></td>
<td>this.f</td>
</tr>
<tr>
<td></td>
<td>methinvok</td>
</tr>
<tr>
<td></td>
<td>new A(exp_{1}, \ldots, exp_{n})</td>
</tr>
<tr>
<td>stmt ::= methinvok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>left = exp</td>
</tr>
<tr>
<td></td>
<td>stmt_{1}; stmt_{2}</td>
</tr>
<tr>
<td>left ::= x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>this.f</td>
</tr>
<tr>
<td>methinvok ::= receiver \leftarrow m(exp')</td>
<td></td>
</tr>
<tr>
<td></td>
<td>receiver \leftarrow m(exp')</td>
</tr>
<tr>
<td>receiver ::= exp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>super(A)</td>
</tr>
</tbody>
</table>

3. The Core Language DOCS

DOCS is proposed as a contender for a minimal core calculus, including basic imperative features of standard object-oriented languages such as C++ and Java, together with more advanced features like encapsulated multi-methods and copy semantics of inheritance.

The syntax of DOCS is defined in Table 1 and 2 starting from

- a set ClassNames of class names denoted by A, B, C, (A_{i}, B_{i}, C_{i});
- a set Fields of names for record fields denoted by f;
– a set Identifiers of identifiers including the set Var of standard variables denoted by \( x, y \), and the set MethNames of method names denoted by \( m, n, m_i, n_i \). We use \( \ell \) to denote any identifier \( x \) or \( m \). We denote with MethNames\((p)\) the set of names of the methods defined in a program \( p \).

**Notation.** Let program = classdef \( * \) body then MethNames(classdef \( * \)) will denote MethNames(program). With respect to the object-oriented core of standard mainstream languages, we adopt the following main simplifications, concerning features that are no relevant for our purpose:

– all methods accept only one parameter;
– we do not consider field access since is an orthogonal feature w.r.t. the problem a issue: thus class fields are all private, and field selection can be performed only on \( \text{this} \); let us note that the construct \( \text{this}.f \) allow us to model mutable objects;
– in order to simplify the presentation of typing, the programmer of the subclass is required to repeat all the instance variables of the superclass, as suggested in [16].

In Table 1 the structure of a program is defined as

\[
\text{classdef} \ast \text{body}
\]

where classdef \( * \) is a sequence (possibly empty) of class definitions. Each class definition can refer to any other class, including its sub-classes, in defining its fields and methods. Class definitions are followed by a body that plays the same role of main in C++ and Java. Let us note that in class definitions, the set of superclasses \( A_h \) \( h \in H \) can be empty.

Since our core language is essentially imperative, we want to treat assignments and side-effects, thus we need to deal with references. In particular, all class fields and method local variables are considered as updatable memory cells (\( i.e., \) all variables are to be considered as Java or C++ references). As in Java, we do not want to specify this behavior in the program: instead of declaring a variable as \( \text{ref } T \ x \) we simply write \( T \ x \), with the same meaning. Accordingly, the assignment between variables \( x = y \) has to be intended as \( x = \uparrow y \), where \( \uparrow y \) is the value referred to by \( y \). The distinction between a reference or location and the value stored in that location is hidden to the programmer, while it will be explicit in formal specifications of typing and semantics. This makes it significantly easier to write type-checking rules and translational semantics.

A multi-method is as a collection of overloaded methods associated to the same message (method name). We will refer to a single overloaded method of a multi-method \( m \) as a branch of \( m \). Class (overloaded) methods are written according to the \textit{branchdef} rule.

**Notation.** If \( b = m_j \text{ branches } B'_{k_j} (B_{k_j}, x_{k_j}) \{ \text{body}_{k_j} \};_{k_j \in K_j} \text{ end} \) then we use the notation \( \text{name}(b) = m_j \).

We allow method bodies to call the implementation of a method in a super-class with the construct \( \text{super}(A) \), where \( A \) is a super-class of the current class method. If we considered only single inheritance, we could simply use \text{super}().
Concerning method invocation, *dynamic binding* is employed, in any case, but when `super(A)` is explicitly used, thus always selecting the most redefined implementation. In particular, all branches of a multi-method declared in a superclass are intended to be implicitly inherited by derived classes, thus implementing a full form of copy semantics of inheritance [3]. Note that in our syntax we only consider multi-methods: standard methods are multi-methods with only one branch. We distinguish two different linguistic constructs for method invocation, denoted by ← and ⇐ that will be associated to different mechanisms for selecting the right branch of the method in DOCS semantics:

- `receiver ← m(exp)` is used for the standard static overloaded method invocation, i.e., the branch of the multi-method is selected statically according to the static type of the parameter (*static overloading*), and dynamically according to the run time type of the receiver (*single dispatch*);
- `receiver ⇐ m(exp)` is used for dynamic overloading method invocation, where the branch of the multi-method is selected dynamically according both to the run time type of the receiver and to the run time type of the parameter (*double dispatch*).

Note that the receiver of a method invocation can be either an expression `exp` or `super(A)` where `A ∈ ClassNames`.

4. Typing

The typing system is straightforward as far as basic object-oriented features are concerned. Our focus is on rules for typing multi-methods, class definitions and then programs. Indeed our primary goal is to ensure that neither “method not understood” nor “method ambiguous” errors can occur in multi-methods dispatch. The notion of multi-type to be associated to multi-methods and its well formedness are widely inspired by [15,16]. However, the adoption of copy semantics of inheritance makes our rules different from [15,16] in some crucial points (for instance the multi-type union constructor, Def. 4.8) leading to a totally different typing of a program as a whole. As a consequence, many static ambiguities that could be raised using the rules of [15,16] are automatically resolved in DOCS.

Following the approach of Featherweight Java [29], we build a class table `CT(p)` that contains type information for all the classes in a program `p`. This table is crucial to type checking multi-method declarations with branches whose parameter types are associated to classes that have not been defined yet: indeed, a class `B` can use the name `A` and vice versa without any issue about which class is defined first. This is a general feature of mainstream languages, such as Java, that is modeled in DOCS. Then we first generate the class table `CT(p)` from class definitions, which associates to each class name both the type representing its fields and its superclass relations. Using `CT(p)` the declarative part is type checked in order to produce the type environment that is needed to type the rest of the program.
Table 3. Syntax of types.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>::= $\text{ClassNames}$</td>
<td>(atomic types)</td>
</tr>
<tr>
<td>$I$</td>
<td>::= $(A \times A)$</td>
<td>(input types)</td>
</tr>
<tr>
<td>$R$</td>
<td>::= $\langle\langle f_1 : \text{ref} A_1; \ldots; f_k : \text{ref} A_k \rangle\rangle$</td>
<td>(record types)</td>
</tr>
<tr>
<td>$T$</td>
<td>::= $A</td>
<td>I</td>
</tr>
</tbody>
</table>

The section is organized as follows:
- in Sections 4.1 and 4.2 we define the syntax of types and we specify the subtyping relation respectively;
- in Section 4.3 we formalize the notion of well formed types;
- in Section 4.4 we define typing rules for $DOCS$ programs. The typing rules presented in this section are parameterized w.r.t. a type environment, which is a triple $(C, S, \Gamma)$;
- in Section 4.5 we finally formalize well typedness for $DOCS$ programs: we first define how to generate the triple $(C_p, S_p, \Gamma_p)$, from any program $p$, then we use $(C_p, S_p, \Gamma_p)$ for type checking the rest of the program (method bodies and the main body), according to the rules of Section 4.4.

4.1. Syntax

The syntax of types is defined in Table 3. Atomic types are class names. It is straightforward to include built in types as atomic types so, when convenient, we will use atomic types such as boolean, real, integers etc. We denote by $\text{AtomicTypes}$ the set of atomic types. Input types are used to represent types of the input in arrow types, namely input types for methods in our case. Record types are used to represent the types of the instance variables of objects. Let us consider the following class definition:

```latex
class A : A_1, \ldots, A_n\{ 
  B_h f_h; h \in H 
  \text{branchdef j}; j \in J 
\};
```

then the record type associated to objects of class $A$ is $\langle\langle f_1 : \text{ref} B_1; \ldots; f_k : \text{ref} B_k \rangle\rangle$ where $H = \{1, \ldots, k\}$.

$U, U_1, T, T_i$ will denote types of our system, thus including:
- atomic types;
- arrow types;
- product types;
- reference types (\text{ref} $A$);
- \text{unit} (denoting the type of assignments);
- multi-types ($\{I_1 \rightarrow T_1, \ldots, I_n \rightarrow T_n\}$).
Multi-types are sets of arrow types representing branches of multi-methods. A multi-type \( \Sigma \) is of the form
\[
\Sigma = \{ I_1 \to T_1, \ldots, I_n \to T_n \}
\]
where each input type \( I_j \) is a pair type, \((A \times B)\).

Reference types are used to represent updatable variables. In order to take into account that any variable \( x \) of type \( \text{ref} \ A \) denotes a location which can store a value of type \( A \), we introduce the following type operation on \( T \), \( \tilde{T} \), in order to distinguish between the type of \( x \) and the type of the value stored in that location.

**Definition 4.1** (ref destructor).
\[
\tilde{T} = \begin{cases} 
A & \text{if } T \equiv \text{ref} \ A \\
T & \text{otherwise.}
\end{cases}
\]

4.2. Subtyping

Now we define the notion of subtyping (\( \leq \)) on types. The subtyping relation is assumed to be given on atomic types in a type constraint environment. Then we define rules to extend subtyping to all types.

**Definition 4.2** (Type constraint environment \( C \)). A type constraint environment \( C \) is the definition of an order relation on atomic types, such that:
1. \( \forall A \in \text{AtomicTypes} (A, A) \in C; \)
2. \((A, B) \in C\) and \((B, A) \in C \Rightarrow A = B; \)
3. \((A, B) \in C\) and \((B, C) \in C \Rightarrow (A, C) \in C. \)

**Definition 4.3** (Rules for subtyping relation \( \leq \)).
\[
C \vdash A \leq B \quad C \vdash T \leq T \quad C \vdash I_1 \leq I_2 \quad C \vdash T_1 \leq T_2 \quad C \vdash T_2 \leq T_3
\]
\[
C \vdash (A \times B) \leq (A_1 \times B_1)
\]

for \((A, B) \in C\)

\[
C \vdash \langle f_1 : \text{ref} \ A_1; \ldots; f_k : \text{ref} \ A_k; f_{k+1} : \text{ref} \ A_{k+1} \rangle \leq \\
\langle f_1 : \text{ref} \ A_1; \ldots; f_k : \text{ref} \ A_k \rangle
\]
\[
C \vdash T \leq T_1 \quad C \vdash U \leq U_1
\]
\[
C \vdash T_1 \to U \leq T \to U_1
\]
\[
\forall k \in K, \exists j \in J \text{ such that } C \vdash I_k \leq I_j \text{ and } C \vdash T_k \equiv T_j
\]
\[
C \vdash \{ I_j \to T_j \} \leq \{ I_k \to T_k \} \quad \forall k \in K
\]
We will write $C \vdash A \leq B$ if $C \vdash A \leq B$ can be proved by using rules of the above Definition, by omitting $C$ when it is clear from the context. Let us observe that subtyping on atomic types is an order relation; instead, subtyping on types is a preorder, only for the case of multi-types. Namely, given two multi-types $\Sigma_1$ and $\Sigma_2$, $\Sigma_1 \leq \Sigma_2$ if and only if for every arrow type in $\Sigma_2$ there is at least one arrow type with greater input type and equal return type in $\Sigma_1$. Now, if $A' \leq A$ then $\{ A \to T \} \leq \{ A \to T, A' \to T \}$ and $\{ A \to T, A' \to T \} \leq \{ A \to T \}$, which does not imply $\{ A \to T, A' \to T \} = \{ A \to T \}$; see [16], page 117, for details on this subject.

Thus $C$ is intended as a set of assumptions on atomic types and it will be used to record subtyping relations on them. The subtyping relation induced by $C$ on atomic types is assumed to be acyclic.

4.3. Well formed types

The notion of well formedness, formally stated in Definition 4.5, introduces a suitable restriction on multi-types, depending on a subtyping relation on atomic types. From now on we will call pretypes the types that are (possibly) not well formed, while Types will denote the set of well formed types and the membership to Types is indexed by the type constraint system $C$ ($\in C$) on which well formedness relies.

We denote by RecordTypes the set of record types whose fields are typed by well formed types.

Definition 4.4 (Maximal types). Let $G$ be a set of input types such that $G \subseteq$ Types and $C$ be a type constraint environment. The set $\text{Top}_C(G)$ of maximal types of $G$ with respect to $C$ is so defined:

$$\text{Top}_C(G) = \{ I \in G \mid \forall I' \in G \ s.t. \ I' \neq I : C \vdash I \not\leq I' \}.$$ 

In the following definition, $\text{LB}_C(I_j, I_i)$ denotes the set of common lower bounds of input types $I_j$ and $I_i$.

Definition 4.5 (Well formed types).

(i) $A \in_C \text{Types unit} \in_C \text{Types}$ for any type constraint system $C$;
(ii) if $A, B, T_1, T_2 \in_C \text{Types}$ then $T_1 \to T_2 \in_C \text{Types}$, $(A \times B) \in_C \text{Types}$ and ref $A \in_C \text{Types}$;
(iii) if for all $j, i \in K$, $K = \{1, \ldots, n\}$
(a) $I_{j(i)}, T_{j(i)} \in_C \text{Types}$,
(b) if $i \neq j$ then $I_i \neq I_j$,
(c) if $C \vdash I_i \leq I_j$ then $T_i \equiv T_j$,
(d) for all maximal types $I'$ in $\text{LB}_C(I_j, I_i)$ (i.e. for all $I' \in \text{Top}_C(\text{LB}_C(I_j, I_i))$) there exists $h \in K$ such that $I_h = I'$, then $\{ I_k \to T_k \} \ k \in K \in_C \text{Types}$. 


Then atomic types are well formed types (point (i)) and also arrow and product types whose internal types are well formed (point (ii)).

Multi-types (point (iii)) are constrained by the crucial consistency conditions of the above definition, that are checked statically. Condition (b) is quite standard on overloaded definitions also in C++ and Java\(^1\). Condition (c) guarantees that a branch specialization maintains the same return type. Condition (d) is the most crucial one: it implies that for any type \(I\), such that \(I \leq I_l\) and \(I \leq I_j\), then there must be one input type in \(\{I_k \rightarrow T_k\}_{k \in K}\), say \(I_h\), such that \(I \leq I_h\), \(I_h \leq I_l\) and \(I_h \leq I_j\). Informally speaking, let us consider the following type constraint system:

\[
C = \{A_2 \leq A_1, B_2 \leq B_1, A_3 \leq A_2, B_3 \leq B_2\}
\]

and two branches in a multi-type \(\Sigma\) whose input types are \(I = (A_1 \times B_2)\) and \(I' = (A_2 \times B_1)\). We first build the set of common lower bounds of \(I\) and \(I'\) w.r.t. the class hierarchy represented by \(C\):

\[
LB_C(I, I') = \{(A_2 \times B_2), (A_2 \times B_3), (A_3 \times B_2), (A_3 \times B_3)\}.
\]

Then we build the set of maximal types of this set, that is, any maximal type of a chain of lower bounds of \(I, I'\) (see Fig. 1):

\[
Top_C(LB_C(I, I')) = \{(A_2 \times B_2)\}.
\]

To satisfy condition (iii\(\text{d}\)) \((A_2 \times B_2)\) has to belong to \(\Sigma\). This last condition will play a crucial role in catching at compile time any possible static and dynamic ambiguity in multi-method calls, since it ensures that for every potentially ambiguous pair of (receiver type \(\times\) parameter type) there is always an input type solving the ambiguity (in the above example, it is \((A_2 \times B_2)\)). Its meaning will become clear when discussing the typing of method invocation.

---

\(^1\)For simplicity, we adopt the restriction of requiring the return types to be equal, thus concentrating our attention on parameters types which play the crucial role in dynamic method selection. We note that, in our approach, no technical difficulty would arise if the above definition of well formedness was extended to covariance of return types, as in [15,16]. In most programming languages return types are not used in overloading resolution; in Java, for instance, \(T_1\) and \(T_j\) can be completely unrelated and this is sound because overloading resolution is static (namely, resolution of static overloading and covariance of return types are two unrelated features in Java and C++).
4.4. Typing DOCS constructs

In this section we define a type system for assigning types to expressions, method bodies and the main body of a program in a given type environment consisting in the triple \( \langle C, S, \Gamma \rangle \) where:

- \( C \) is a type-constraint environment (see Def. 4.2);
- \( S \) is an environment recording the internal representation of classes; namely, a set \( \{ A_1 : R_1, \ldots, A_n : R_n \} \) where \( A_i \)'s are class names and \( R_i \)'s are record types representing the types of \( A_i \)'s fields;
- \( \Gamma \) is a type environment associating types to identifiers (this, variables and multi-method names).

**Record environment** \( S \). A record environment \( S \) is a finite set (possibly empty) \( \{ A_1 : R_1, \ldots, A_n : R_n \} \) where \( R_1, \ldots, R_n \) are are record types of the shape \( \langle \langle f_1 : \text{ref} \ B_1; \ldots; f_k : \text{ref} \ B_k \rangle \rangle \). \( A_1, \ldots, A_n \) are the subjects of \( S \), as for the type environments \( C \) and \( \Gamma \). All subjects of \( S \) are pairwise disjoint and \( \text{Dom}(S) = \{ A | A : R \in S \} \). \( S \) associates to each atomic type the representation of the objects of the corresponding class.

For instance, given the following class definition:

```java
class A : A_1, \ldots, A_n {
  B_1 f_i; \ i \in I
  branchdef j; \ j \in J
};
```

then a record environment \( S \) can be updated for recording the representation of \( A \) in the following way:

\[
S \cup \{ A : \langle \langle f_1 : \text{ref} \ B_1; \ldots; f_k : \text{ref} \ B_k \rangle \rangle \}
\]

provided that \( A \notin \text{Dom}(S) \). We write \( S(A) = R \) to denote that \( A : R \in S \).

**Type environment** \( \Gamma \). The static environment \( \Gamma \) is a finite set (possibly empty) \( \{ \ell_1 : T_1, \ldots, \ell_n : T_n \} \) where:

- \( \ell_1, \ldots, \ell_n \) are called the *subjects* of \( \Gamma \);
- we denote by \( \text{Dom}(\Gamma) = \{ \ell | T \in \Gamma \} \) the domain of \( \Gamma \);
- \( \ell_i \neq \ell_j \) for each \( \ell_i, \ell_j \in \text{Dom}(\Gamma) \) (\( i \neq j \)).

Then, if \( \ell : T \in \Gamma \) we write \( \Gamma(\ell) = T \). Let us note that, according to DOCS syntax, identifiers \( \ell_i \) can only be:

- variables i.e. \( x : T \);
- the special identifier **this**, then \( \Gamma(\text{this}) = A \) for some class name \( A \);
- method names, then \( m : \Sigma \) implies that \( \Sigma \) is a multi-type; informally, \( \Sigma \) records the type of the branches of each multi-method in the form \( (A \times B) \rightarrow T \), where \( A \) is the type of the class where the branch has been defined and \( B \) is the type of the argument of the branch.

In the following we define the notion of consistency for the triple \( \langle C, S, \Gamma \rangle \).
Definition 4.6 \((\langle C, S, \Gamma \rangle \text{ consistency})\).

The triple \(\langle C, S, \Gamma \rangle\) is a consistent environment if and only if:

1. \(C\) is consistent with \(S\), that is if \(C \vdash A \leq B\) then \(C \vdash S(A) \leq S(B)\).
2. \(\Gamma\) is consistent with \(C\) and \(S\)
   - if \(\text{exp:ref} \ A \in \Gamma\) or \(\text{exp:} A \in \Gamma\) then \(S(A) = R\) for some \(R\);
   - if \(m : \Sigma \in \Gamma\) then \(\Sigma \in_{\mathcal{C}} \text{Types} \) (that is \(\Sigma\) is well formed w.r.t. the subtyping relations of \(C\)).

Notice that point (1) implies that all fields of the superclass \(A\) are repeated in the subclass \(B\).

Given a consistent environment \(\langle C, S, \Gamma \rangle\), \(\langle C, S, \Gamma[\ell \leftarrow T']\rangle\) will denote the consistent environment \(\langle C, S, \Gamma' \rangle\) where \(\Gamma' = \Gamma[\ell : T] \cup \{\ell : T'\}\). Typing rules use formulas of the shape \(C;S;\Gamma \vdash t : T\), where \(t\) is a DOCS term (i.e., a program, a body, an expression, a statement or a class definition), \(T\) is a well formed type w.r.t. \(C\) and the triple \(\langle C, S, \Gamma \rangle\) is assumed to be consistent.

Typing rules. Typing rules, parameterized on \(\langle C, S, \Gamma \rangle\), are presented in Tables 4 and 5. Let us observe that the environment \(\Gamma\) is never updated but in the case of rule \(\text{BODY}\) and \(\text{PROGRAM-CLASS}\): the context \(\Gamma\) is updated by adding reference types of variables and parameters and the type of \(\text{this}\), in order to check that the method body is well typed.

Multi-method invocation. In \(\text{DOVMETHCALL}\) and \(\text{SOVMETHCALL}\) rules, we search for the most appropriate (specialized) branch for the invocation assuming that \(\Gamma\) associates to \(m\) the corresponding well formed multi-type. Note that, thanks to the definition of well-formed type \((\text{predicate} \in_{\mathcal{C}} \text{Types})\), if such a branch exists it can be selected without ambiguities.

Let us discuss this in further details. If a multi-method invocation is well typed, with type \(\Sigma\), then for any possible input type \(I = (A \times B)\) the set \(IT_{(A \times B)}\) of invocable branches input types

\[
IT_{(A \times B)} = \{(A_i \times B_i) \mid (A \times B) \leq (A_i \times B_i)\}
\]

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

Moreover, since \(\Sigma\) is well-formed, the above set \(IT_{(A \times B)}\) contains a minimum input type, say \(I_k = (A_k \times B_k)\), i.e., \((A_k \times B_k) \leq (A_i \times B_i)\) for all \((A_i \times B_i) \in IT_{(A \times B)}\), by condition (iiid) (in Def. 4.5) of well-formedness. The branch of type \(I_k \rightarrow T_k\) is the only one that “best approximates” the input type \(I = (A \times B)\). This means that there is one and only one most specific applicable branch for any possible choice of \(I\), i.e., in a well typed program no message-ambiguous error will take place during the execution.

Let us stress that the above argument motivating the absence of message-ambiguous holds not only for any static input type \(I\), but also for any dynamic input subtype \(I' \leq I\). In this case the set of input types of matching branches, \(IT_{I'}\), could become larger \((IT_I \subseteq IT_{I'})\), and so the best approximating branch
Table 4. Typing rules: expressions, statements and bodies.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="LaTeX" /></td>
<td><img src="image" alt="LaTeX" /></td>
</tr>
</tbody>
</table>

532

L. BETTINI, S. CAPECCHI AND B. VENNERI
Table 5. Typing rule for programs.

\[ \forall k \in K, j \in J \]
\[ C; S; \Gamma[x_k \leftarrow \text{ref } B_k][\text{this } \leftarrow A] \vdash \{ \text{body}_k \} : U_k \]
\[ C \vdash U_k \leq B'_k \]
\[ C; S; \Gamma \vdash p : T \]

\[ \text{[Program-class]} \]
\[ \text{class } A : A_1, \ldots, A_n \{ \]
\[ B_i \, j_i : \in i \}
\[ m_j \text{ branches } \]
\[ B'_k \{ x_k \} \{ \text{body}_k \}; k \in K_j \] : T
\[ \text{end} \, j \in J \}
\[ ]; p \]

to be selected dynamically (minimum input type) could have a more specialized input type. The existence and the uniqueness of such branch are still ensured by condition \((iiid)\) of well-formedness, as explained above.

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

1. the static return type is unique and preserved during evaluation;
2. there will always be one and only one most specialized branch to be selected.

Formally, this property will be proved in Section 7 by providing a translational semantics into \(\lambda_{\text{object}}\) and then by lifting the type safety of \(\lambda_{\text{object}}\) to \(\text{DOCS}\).

**Typing programs.** Rule \(\text{[Program-class]}\) (Tab. 5) shows how the typing algorithm proceeds in the case of a program, given the type environment \((C, S, \Gamma)\).

This rule checks well typedness for multi-method definitions, namely that type of each branch definition of \(m\) is consistent with the well formed multi-type that is associated to \(m\) in \(\Gamma\). The body is type checked (rule \(\text{[Body]}\), Tab. 4) in the environment \(\Gamma\) updated with the types of the parameter and of \(\text{this}\). Instead, the correctness of inheritance w.r.t. fields (namely that all the fields of the superclasses are repeated in the class definition), is assured by the consistency condition of the triple \((C, S, \Gamma)\) (see Def. 4.6).

We conclude this section with a basic property that characterizes the expressions of \(\text{DOCS}\), in particular their types.

**Property 4.1.** For any expression \(\text{exp}\) and body \(\\text{body}\),

1. if \(C; S; \Gamma \vdash \text{exp} : T\), then \(T \equiv A\) or \(T \equiv \text{ref } A\) for some class name \(A\);
2. if \(C; S; \Gamma \vdash \{ \text{body} \} : T\), then \(T \equiv A\) for some class name \(A\).

**Proof.**

1. Straightforward by rules \(\text{[Taut]}\), \(\text{[New]}\), \(\text{[Real]}\), \(\text{[DOVMethCall]}\), \(\text{[SOVMethCall]}\).
2. By rule \(\text{[Body]}\) and point 1. \(\square\)
4.5. Well typedness of DOCS programs

Well typedness of DOCS programs is expressed by the formula \( p : T \) (read as “\( p \) has type \( T \)”) where \( p \) is a program and \( T \) is an atomic type (i.e., \( T \) is well formed w.r.t. any \( C \)).

The statement \( p : T \) is formalized in Definition 4.12 after the definition of the class table \( CT(p) \) and the type environment \( \Gamma_p \) that are associated to the program \( p \).

We first explain the use of \( CT(p) \) in checking the well typedness of DOCS programs. In DOCS a method \( m \) in a class \( A \) can have a parameter of type \( B \), where class \( B \) is defined after the declaration of \( A \); the same holds for types of fields. This reflects the style of Featherweight Java: we view the set of class names as being given from the beginning, so that there is no issue about which class is defined first. Thus we have to collect the type information concerning all class definitions in a program \( p \), before checking each class. Given a DOCS program \( p \), \( CT(p) \) records, for each class \( A \) defined in \( p \):

- the record type representing its fields;
- the subtyping relations (on atomic types) induced by the inheritance hierarchy;
- the method definitions of the branches defined in \( A \).

The third part of the table, recording the method definitions associated to each class, will be used only to define DOCS semantics (Sect. 6).

**Definition 4.7 (Class table \( CT \)).**

Let \( p = \text{classdef}^* \text{body} \) be a DOCS program, then

1. for any class definition in \( \text{classdef}^* \) of the shape

\[
\text{class } A : A_1, \ldots, A_n \{
\begin{align*}
& B_i \text{ }; i \in I \\
& m_j \text{ branches } \\
& B'_k (B_k \ x_k) \{\text{body}_k\}; k \in K_j \\
& \text{end } j \in J
\end{align*}
\]

we define:

- \( C_p(A) = \langle \langle f_i : \text{ref } B_i \rangle \text{ }; i \in I \rangle \);
- \( S_p(A) = \{ A \leq A_1, \ldots , A \leq A_n \} \);
- \( BT_p(A, m_j, B_k) = \{ \text{body}_k \} \);

2. \( CT(p) \) is the class table built from \( \text{classdef}^* \) in the following way:

\[
CT(p) = (C_p, S_p, BT_p).
\]

We assume that \( CT(p) \) satisfies some sanity conditions:

- any class name \( A \) is defined only once;
- if \( A_i \leq A_j \in S_p \) then \( C_p(A_i) \leq C_p(A_j) \), namely the subclass relation is consistent with subtyping on the corresponding record types;
there are no cycles in the subclass relations recorded in \( CT(p) \), that is if \( A \leq B \in S_p \) then \( B \) cannot be a subclass of \( A \).

Similarly to \( S_p, C_p \) and \( BT_p \), we also need to generate the type environment \( \Gamma_p \) which associates to each multi-method \( m \) in \( p \) its multi-type \( \Sigma \). For example, let us consider a program with the following class definitions:

```plaintext
class A{
    m branches
        T (D x){body1};
    end
}
class B{
    m branches
        T (D x){body2};
    end
}
class C : A, B{
    m branches
        T (D x){body3};
    end
}
```

Then \( C_p = \{ C \leq A, C \leq B \} \). The multi-type associated to \( m \) in class \( B \), \( \{(A \times D) \rightarrow T, (B \times D) \rightarrow T\} \), is not well formed w.r.t. \( C_p \) since it does not satisfy condition (d) in Definition 4.5. The problem is due to the fact that, when we check the branch defined in class \( B \), we have not already encountered class \( C \) which defines the disambiguating branch while \( C_p \) already includes the relations \( C \leq A \) and \( C \leq B \).

Thus we have to generate the environment \( \Gamma_p \) in a preliminary scan of \( p \), using the function \( MT(classdef^{\ast}) \) which is a finite set of associations of the form \( \{m_1 : \Sigma_1, \ldots, m_n : \Sigma_n\} \) where the subjects \( m_i \)’s are method names belonging to \( classdef^{\ast} \) and \( \Sigma_i \)’s are multi-types possibly not well formed (pretypes). If \( m : \Sigma \) belongs to \( MT(classdef^{\ast}) \) then \( MT(classdef^{\ast})(m) = \Sigma \). Since each method name \( m \) can be the subject of at most one association, we will write

\[
MT(classdef^{\ast}) \cup \{m_i : \Sigma_i^{\prime}\}
\]

to denote

\[
(MT(classdef^{\ast}) \setminus \{m_i : \Sigma_i\}) \cup \{m_i : \Sigma_i^{\prime}\}
\]

when updating \( MT \).

We introduce two operators on multi-types that will be used for the construction of \( MT \):

- \( \cup \) merges two multi-types performing copy semantics of inheritance at the typing level;
- \( \downarrow \) returns the branch types defined in a given class.

**Definition 4.8 (Multi-method type union \( \cup \)).** Given two multi-types \( \Sigma_1 \) and \( \Sigma_2 \) and a type constraint system \( C \), the new multi-type \( \Sigma_1 \cup \Sigma_2 \) is defined as follows:

\[
\Sigma_1 \cup \Sigma_2 = \Sigma_1 \cup \Sigma_2 \cup \{(A \times D) \rightarrow T \mid (A_i \times D) \rightarrow T \in \Sigma_1 \land C \vdash A \leq A_i \}
\]

Informally speaking the operator \( \cup \) copies each arrow type \( D \rightarrow T \) defined in a superclass \( A_i \) into the subclass \( A \) (thus changing the receiver type) since every branch not redefined in \( A \) is considered to be explicitly inherited.
Definition 4.9 (Restriction). If $\Sigma$ is a multi-type and $A$ a class name, then $\Sigma \downarrow A$ selects from a multi-type $\Sigma$ the branch types where the type of the receiver is $A$:

$$\Sigma \downarrow A = \{ B \rightarrow T \mid (A \times B) \rightarrow T \in \Sigma \}.$$ 

Definition 4.10 (MT).

Let $\text{classdef}^* = \text{classdef}_1, \ldots, \text{classdef}_n$ be a set of class definitions then we define $\text{MT}(\text{classdef}^*)$ by cases on the cardinality of classdef* as in the following:

1. $\text{MT}(\emptyset) = \emptyset$
2. $\text{MT}(\text{classdef}_1, \ldots, \text{classdef}_n) = \text{MT}(\text{classdef}_1, \ldots, \text{classdef}_{n-1}) \cup \{m_j : \Sigma_j \uplus\{A_{\leq A_h}\}_{h \in H} \ni \{((A \times B_{k_j}) \rightarrow T_{k_j}) \mid k_j \in K_j\} \forall j \in J\}$

if $\text{classdef}_n = \text{class } A : A_h \in H\{\ldots; \text{branchdef}_j; \; j \in J\}$;

and

$\text{branchdef}_j = m_j \; \text{branches} \; T_{k_j} (B_{k_j} \; x_{k_j})\{\text{body}_{k_j}\}; \; k_j \in K_j$$\text{end}$

where

$$\Sigma_j = \left\{ \begin{array}{ll} \text{MT}(\text{classdef}_1, \ldots, \text{classdef}_{n-1})(m_j) & \text{if } m_j \in \text{MethNames}(\text{classdef}_1, \ldots, \text{classdef}_{n-1}) \\ \emptyset & \text{otherwise} \end{array} \right.$$ 

provided that,

$$\forall j \in J \text{ and } l, l' \in H, \; \Sigma_j \downarrow A_l \cap \Sigma_j \downarrow A_{l'} \subseteq \{B_{k_j} \rightarrow T_{k_j} \mid k_j \in K_j\}.$$ 

Thus, at each recursive step, for each multi-method $m_j$ defined in the current class, the type $\Sigma_j$ of the multi-method class is updated as in the following (let us note that $\Sigma_j = \emptyset$ if $m_j$ has not already been encountered):

$$\Sigma_j \uplus\{A_{\leq A_h}\}_{h \in H} \ni \{((A \times B_{k_j}) \rightarrow T_{k_j}) \mid k_j \in K_j\}$$

where:

- all the arrow types of the branches defined in the current class are collected in the multi-type $\{((A \times B_{k_j}) \rightarrow T_{k_j}) \mid k_j \in K_j\}$;
- the union between the two multi-types is performed using the operator $\uplus$.

By using the operator $\uplus$ all the branches of a superclass are implicitly inherited by derived classes, apart from those redefined by the derived class itself. For instance, if the multi-method $m$ has type $\Sigma_1 = \{(A_1 \times B_1) \rightarrow T\}$ in the base class $A_1$ and type $\Sigma_2 = \{(A_2 \times B_2) \rightarrow T\}$ in the derived class $A_2$, then it will be given the type $\Sigma_1 \uplus\{A_2 \leq A_1\} \Sigma_2 = \{(A_1 \times B_1) \rightarrow T, (A_2 \times B_1) \rightarrow T, (A_2 \times B_2) \rightarrow T\}$. This corresponds to the intuition that branches that are not redefined in the subclass are intended to be inherited. In the context of copy semantics it means that the branch code that is inherited is copied down in the subclass: from the typing view this branch is assigned the new type where the input type is updated replacing the class where the branch has been defined with the current class.

Due to multiple inheritance, we have to check that two superclasses do not define the same method branch with the same signatures, or, in case they do, we
must ensure that the derived class provides a declaration for such branch, in order to avoid ambiguities. Note that our policy is to avoid a silent resolution of such ambiguities (as, e.g., in [38]) and to force the programmer to deal with them.

For instance, if \( m \) has type \((A \times D) \rightarrow T\) in class \( A \) and type \((B \times D) \rightarrow T\) in class \( B \) and class \( C \) inherits both from \( A \) and \( B \), then we must check that the branch \( D \rightarrow T \) is redefined in \( C \) in order to univocally associate a body to the branch of \( m \) with type \((C \times D) \rightarrow T\). This check relies on the operators of restriction (\( \downarrow \)) and intersection (\( \cap \)). The two operators are needed to collect the set of common branches definitions for a multi-method \( m \) into two different classes. Indeed, to catch ambiguities due to multiple inheritance for each pair of superclasses \( A_l, A_l' \) we perform the following checks:

1. we select the branches defined in the two superclasses with the operator \( \downarrow \): \( \Sigma_j \downarrow A_l, \Sigma_j \downarrow A_l' \);
2. we collect the set of arrow types common to the two classes: \( \Sigma_j \downarrow A_l \cap \Sigma_j \downarrow A_l' \);
3. finally we check that the arrow types common to the two superclasses are redefined in the current class: \( \Sigma_j \downarrow A_l \cap \Sigma_j \downarrow A_l' \subseteq \{ B_k j \rightarrow T_k j \} \ k_j \in K_j \).

Thus, the construction of \( \Sigma_j \) is intended to fail if a method inherited by two superclasses is not redefined.

Concluding, we stress on two distinctive features of our approach, concerning the type checking of multi-methods definitions. First, we remark that with copy semantics we obtain a less restrictive mechanism w.r.t. standard inheritance. Let us consider the two following classes:

\[
\begin{align*}
\text{class } A \{ & \quad \text{class } B : A \{ \\
& \quad m \text{ branches } T (B' x) \{\text{body}_1\}; & & m \text{ branches } T (A' x) \{\text{body}_2\}; \\
& \quad \text{end} & & \text{end}
\end{align*}
\]

where \( B' \leq A' \), and let \( \Sigma_1 = \{(A \times B') \rightarrow T\}, \Sigma_2 = \{(B \times A') \rightarrow T\} \) be the types of \( m \) in \( A \) and \( B \) respectively. The multi-type \( \Sigma_1 \cup \Sigma_2 = \{(A \times B') \rightarrow T, (B \times A') \rightarrow T\} \), obtained by standard inheritance, is not well formed because there is no most specific branch for the input type \((B \times B')\), thus condition iii(d) of well formedness on multi-types should be violated. With copy semantics of inheritance we obtain a well formed multi-type:

\[
\Sigma_1 \cup_{(B \leq A)} \Sigma_2 = \{(A \times B') \rightarrow T, (B \times A') \rightarrow T, (B \times B') \rightarrow T\}.
\]

Second, even if this mechanism is more flexible, we check inconsistencies due to multiple inheritance, so we do not solve silently ambiguities that have to be explicitly solved by the programmer.

Using \( \mathbf{MT} \) we can define the type environment \( \Gamma_p \).

**Definition 4.11** \((\Gamma_p)\).

Let \( p = \text{classdef }^* \text{ body} \) be a DOCS program. Let \( C_p \) be defined as in Definition 4.7. Then \( \Gamma_p \) will denote the following environment:

\[
\Gamma_p = \mathbf{MT}(\text{classdef }^* )
\]
if and only if for each association \( m : \Sigma \in \text{MT}(\text{classdef } \ast) \) then
\[
\Sigma \in C_p \text{ Types.}
\]

**Definition 4.12** (Well typedness of DOCS programs).

Let \( p \) be a DOCS program. We say that \( p \) is well typed, denoted by \( \mapsto p : A \),
\[
\mapsto p : A \iff C_p ; S_p ; \Gamma_p \vdash p : A \quad \text{for some } A
\]
where \( C_p, S_p \) and \( \Gamma_p \) are defined in Definitions 4.7 and 4.11.

It is trivial to verify that \( (C_p, S_p, \Gamma_p) \) is a consistent environment by construction of \( C_p, S_p \) and \( \Gamma_p \). Summarizing, the typing algorithm of a program \( p \) can be described as a procedure performing two steps.

First we analyze the declarative part of the program collecting:
- the subtype relations induced by the inheritance hierarchy \( (C_p) \);
- the record types associated to the classes \( (S_p) \);
- the multi-type associated to each multi-method \( (\Gamma_p) \).

Then, the triple \( (C_p, S_p, \Gamma_p) \), produced in the first step, is used to type expressions and bodies in the program. Namely:
- rule \([\text{Program-class}]\) checks method definitions relying on rule \([\text{Body}]\);
- rule \([\text{Body}]\) gives type to the main body of the program.

5. **The meta-language \( \lambda_{\text{object}} \)**

In this section we briefly introduce \( \lambda_{\text{object}} \), a meta-language for reasoning about properties of constructs of object-oriented languages introduced in [15]. \( \lambda_{\text{object}} \) is essentially defined as an extension of the simply typed lambda calculus to model basic object-oriented mechanisms. Since we want to represent DOCS into \( \lambda_{\text{object}} \), we use an imperative version of \( \lambda_{\text{object}} \) which is characterized by the representation of the store \( S \)(it is sketched in [16], Sect. 9.1.2). Namely, special typed identifiers \( id^T \) will denote references to (addresses of) memory locations holding values of type \( T \) (differently from variables \( x^T \) denoting values of type \( T \)).

Let us first discuss, in an informal way, two crucial ideas underlying \( \lambda_{\text{object}} \), concerning the representation of objects and overloaded functions.

**Objects.** Let us consider a simple definition of a class \( A \) in DOCS:

```plaintext
class A{
    int x = 0;
    char y = 'a';
    ....
}
```

This will correspond, in \( \lambda_{\text{object}} \), to the definition of an atomic type \( A \) that is associated to the record type of its instance variables:

\[
\langle \langle x : \text{ref int}, y : \text{ref char} \rangle \rangle.
\]
Table 6. $\lambda_{\text{object}}$ terms.

| Expressions | $M ::= x^T | \lambda x^T.M | M \cdot M | \epsilon | M \& T | M \cdot M | (M,M) |
|-------------|--------------------------------------------------|
|             | $\text{super}^A(M) | \text{in}^A(M) | \text{out}^A(M) | \pi_1(M) | \pi_2(M) |
|             | $id^T | \uparrow M | M:=M | M;M | \text{nil}$ |

| Terms | $P ::= M | \text{let } A \leq A, ..., A \text{ in } P | \text{let } A \text{ hide } T \text{ in } P$ |

Let us note that $x$ and $y$ are mutable instance variables denoting memory locations whose content can be updated by side effects. Indeed, the creation of an object of a class $A$ involves the allocation of free memory associated to fields $x$ and $y$ in an exclusive way. To this aim two fresh identifiers, $id_1\text{int}$ and $id_2\text{char}$ (of type $\text{ref int}$ and $\text{ref char}$, respectively) will be introduced at the moment of the creation of an object, that will perform the following operations:

$id_1\text{int} := 0; id_2\text{char} := 'a'; \text{in}^A((x = id_1\text{int}, y = id_2\text{char}))$

where $\text{in}^A$ is the constructor that is tagged by the type name $A$. Thus the object $\text{in}^A((x := id_1\text{int}, y := id_2\text{char}))$ is the object formed only by the values stored in $id_1$ and $id_2$ (its internal state) and by the tag $A$ denoting its type. This approach to the problem of object representation is a distinctive feature of $\lambda_{\text{object}}$ w.r.t. the record based model [13].

In order to access instance variables of objects we use the construct $\text{out}^A : A \rightarrow T$ which composed with $\text{in}$ gives the identity. To select the $j$th field we use the selection operator which returns the identifier associated to the selected field:

$(f_i = id_i^{B_i}, i \in I).f_j = id_j^{B_j}$

$\text{out}^A(\text{in}^A((f_i = id_i^{B_i}, i \in I))).f_j = id_j^{B_j}$

**Overloaded functions.** Overloaded functions are the $\lambda_{\text{object}}$ terms representing multi-methods. An overloaded function is formed by a set of ordinary functions ($\lambda$-abstractions) each one constituting a different branch. Overloaded functions are built as lists starting with an empty overloaded function denoted by $\epsilon$ and concatenating new branches by means of $\&$:

$\epsilon \& M_1 \& ... \& M_n$

where $M_i = \lambda x \cdot N_i$ and $N_i$ is a $\lambda_{\text{object}}$ expression representing the body of the $\lambda$ abstraction (see Tab. 6) and $M_1 \ldots M_n$ correspond to the branches of a multi-method. Furthermore, the $\&$ conjunctions are explicitly typed as well as all terms of $\lambda_{\text{object}}$: these types will be used to select the branch in method invocation.
and to type check multi-methods:

$$\epsilon \& T_1 \& T_2 \ldots \& T_n \ M_n.$$

The branches are concatenated so that subtypes always precede supertypes; thus, in the overloaded term $$\epsilon \& T_1 \& T_2 \ M_1 \& T_2 \ldots \& T_n \ M_n$$ either $$T_1$$ and $$T_2$$ are unrelated or $$T_1 \leq T_2$$ (see Sect. 6.2 for explanations of type ordering in overloaded functions). To perform an application $$M \bullet N$$ of the overloaded term $$M$$ to an argument, we first reduce $$N$$ to a value (tagged with its type) and $$M$$ to an overloaded term $$\epsilon \& T_1 \& T_2 \ldots \& T_n \ M_n$$. Then the branch selection is performed according to the tag of the argument, which is the actual type of the value obtained by reducing $$N$$. This way, method call implements dynamic overloading.

In the following sections we summarize formal definitions of $$\lambda_{\text{object}}$$ that are relevant for $$\text{DOCS}$$; we refer the reader to [16] for further details about $$\lambda_{\text{object}}$$.

5.1. $$\lambda_{\text{object}}$$: Syntax

First of all, $$\lambda_{\text{object}}$$ is not a language but a meta-language: this means that it must have very few constructs in order to making proofs about languages as simple as possible. For instance, records are not included in the syntax since they can be completely encoded (see encoding of records by overloaded functions in [16] Sect. 4.5.2). In the following sections we will explicitly use record and record types only to simplify the presentation of our translation from $$\text{DOCS}$$ to $$\lambda_{\text{object}}$$. It is clear that the language version using records and record types can be automatically translated into the equivalent form not including them.

Terms in $$\lambda_{\text{object}}$$ are composed by an expression preceded by a (possibly empty) suite of declarations (see Tab. 6)\(^2\). Since $$\text{coerce}^{\text{O}}(M)$$ is not interesting in our context we do not consider it in the syntax.

Some notes about terms:

- **super**\(^A\)(\(M\)): the type of \(M\) is upcast to \(A\). In $$\lambda_{\text{object}}$$ **super** does not necessarily appear in the receiver position but it is a first class value;
- \(\text{id}^T\) is the identifier of a location of type \(\text{ref} \ T\) while \(\uparrow M\) is the content of the location \(M\);
- the declaration **let** \(A\) **hide** \(T\) **in** \(P\) declares the atomic type \(A\) and associates it to the type \(T\) used for its representation. This declaration is coupled with the two constructors \(\text{in}^A : T \to A\) and \(\text{out}^A : A \to T\).

A program in $$\lambda_{\text{object}}$$ is a closed term \(P\) different from \(\epsilon\) (since \(\epsilon\) is the empty function, it makes no sense to consider it as a valid program since it could not be used in any application).

\(^2\)In [16] the syntax of tuples is \((M,M)\) while here is \((M,M)\) to distinguish it from record notation.
5.2. $\lambda_{object}$: Typing

The types in $\lambda_{object}$ are defined starting from a set of pre-types which is then restricted to well formed types,

$$D ::= A | D \times D$$

$$T ::= A | T \to T | (T \times T)$$

$$| \{D \to T, \ldots, D \to T\} | \text{ref} | \text{unit}.$$  

Let us note that the types of $\lambda_{object}$ are a superset of the types of DOCS (Sect. 4) so we will use only a subset of $\lambda_{object}$ types$^3$. The definition of the subtyping relation and of well typedness are the same as the ones defined for DOCS respectively in Sections 4.2 and 4.3.

As for the language DOCS, the rules are parametric w.r.t. a type constraint environment $C$ and a record environment $S$ (see Sect. 4 for a definition of $C$ and $S$). $\lambda_{object}$ typing rules are listed in Tables 7 and 8.

Notations. We will use $I$ as in DOCS to denote a pair of atomic types $(A \times B)$ and $\Sigma$ to denote a type of the shape $\{I_1 \to T_1, \ldots, I_n \to T_n\}$.

The following definition is used to update a multi-type.

**Definition 5.1** (Overloading composition).

$$\{I_1 \to T_1, \ldots, I_n \to T_n\} \oplus (I \to T) =$$

$$\begin{cases} 
\{I_1 \to T_1, \ldots, I_i-1 \to T_i-1, I_{i+1} \to T_{i+1}, \ldots, I_n \to T_n, I \to T\} & \text{if } I = I_i \\
\{I_1 \to T_1, \ldots, I_n \to T_n, I \to T\} & \text{otherwise.}
\end{cases}$$

**Remark 5.1.** With respect to the method selection, the order of the type indexing the & is crucial (see rule $[\ ]$INTRO and Definition 5.1 for the construction of the types indexing the &’s). For instance, the two terms

$$\epsilon \& M_1 \& (U \to V) M_2 \& (U \to V, U' \to V) M_3$$

and

$$\epsilon \& M_1 \& (U \to V) M_2 \& (U' \to V, U \to V) M_3$$

where $U' \leq U$, are both well typed but they behave differently if applied to an argument of type $U$: in the first case we execute $M_2$, while in the second case we execute $M_3$. This matter raises also in the semantics (interpretation) of DOCS multi-methods (Sect. 6.2) where we will use a permutation operator to generate the $\lambda_{object}$ multi-method branches in the correct order (Def. 6.5).

---

$^3$ Note that unit is the type denoted as () in [16]. We preferred to adopt this notation for the sake of readability.
Table 7. $\lambda_{\text{object}}$ typing rules (part I).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Taut]</td>
<td>$C;S \vdash \iota^T : T$</td>
</tr>
<tr>
<td>[Recori]</td>
<td>$C;S \vdash M_i : T_i \forall i \in I$</td>
</tr>
<tr>
<td>[Pair]</td>
<td>$C;S \vdash M : U \quad C;S \vdash N : V$</td>
</tr>
<tr>
<td>$\rightarrow$-Intro</td>
<td>$C;S \vdash M : V$</td>
</tr>
<tr>
<td>$\rightarrow$-Elim</td>
<td>$C;S \vdash \lambda^U. M : U \rightarrow V$</td>
</tr>
<tr>
<td>$\rightarrow$-Intro</td>
<td>$C;S \vdash M : U \rightarrow V \quad C;S \vdash N : T \quad T \leq U$</td>
</tr>
<tr>
<td>$\rightarrow$-Elim</td>
<td>$C;S \vdash M \cdot N : V$</td>
</tr>
<tr>
<td>[Taut]</td>
<td>$C;S \vdash \epsilon : {}$</td>
</tr>
<tr>
<td>[[]]Intro</td>
<td>$C;S \vdash M : T_1 \quad C;S \vdash T_1 \leq {T_i \rightarrow V_i} \forall i \in I \quad C;S \vdash N : T_2 \quad C;S \vdash T_2 \leq T \rightarrow V$</td>
</tr>
<tr>
<td>[[]]Elim</td>
<td>$C;S \vdash M : {U_i \rightarrow V_i} \forall i \in I \quad C;S \vdash N : U$</td>
</tr>
<tr>
<td>[NewType]</td>
<td>$C;S \vdash \text{let } A \leq A_i \text{ hide } T \text{ in } P : U$</td>
</tr>
<tr>
<td>[Constraint]</td>
<td>$C \cup (A \leq A_i)_{i=1,...,n} \vdash P : T$</td>
</tr>
</tbody>
</table>

5.3. $\lambda_{\text{object}}$: Operational Semantics

$\lambda_{\text{object}}$ operational semantics is given by the reduction $\Rightarrow$; this reduction is defined on configurations of the shape $(C,S,M)$ where:

- $C$ is a type constraint system that is built along the reduction by the declarations \textbf{let } $A \leq A_1,...,A_n$ \textbf{in } $P$ and that is used by the rules for the selection of the branch;
- $S$ is a global store associating identifiers $id^T$ to values;
- $M$ is the $\lambda_{\text{object}}$ term to reduce.
Table 8. \( \lambda_{\text{object}} \) typing rules (part II).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Antecedent</th>
<th>Consequent</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Super}</td>
<td>( C; S \vdash M : B )</td>
<td>( C \vdash B \leq A ) and ( A \in_{C,S} \text{Types} )</td>
</tr>
<tr>
<td>\textbf{In}</td>
<td>( C; S \vdash M : T )</td>
<td>( C \vdash T \leq S(A) ) and ( A \in_{C,S} \text{Types} )</td>
</tr>
<tr>
<td>\textbf{Out}</td>
<td>( C; S \vdash M : B )</td>
<td>( C \vdash B \leq A \in_{C,S} \text{Types} )</td>
</tr>
<tr>
<td>\textbf{Nil}</td>
<td>( C; S \vdash \text{nil} : \text{unit} )</td>
<td></td>
</tr>
<tr>
<td>\textbf{Ref Intro}</td>
<td>( C; S \vdash \text{id}^T : \text{ref}\ T )</td>
<td></td>
</tr>
<tr>
<td>\textbf{Ref Elim}</td>
<td>( C; S \vdash M : \text{ref}\ T )</td>
<td>( C; S \vdash \uparrow M : T )</td>
</tr>
<tr>
<td>\textbf{Assign}</td>
<td>( C; S \vdash M : \text{ref}\ T )</td>
<td>( C; S \vdash N : U \leq T )</td>
</tr>
<tr>
<td>\textbf{Seq}</td>
<td>( C; S \vdash M : U )</td>
<td>( C; S \vdash N : T )</td>
</tr>
</tbody>
</table>

Values are terms that can be considered as acceptable result of an execution and are defined as follows:

\[
G ::= \epsilon \mid x^T \mid (\lambda x^T.M) \mid (M \& T.M) \mid \text{nil} \mid \text{id}^T \\
(G, G) \mid \text{super}^A(M) \mid \text{in}^A(M).
\]

We call object values values of the shape \( \text{in}^D(M) \). A tagged value is everything an overloaded function can perform its selection on. The tag \( D \) can be either an atomic type or a product of atomic types:

**Tagged values**

\[
G^D ::= \text{in}^D(M) \mid \text{super}^D(M) \mid (G^{A_1}, \ldots, G^{A_n}).
\]

The axioms and rules of the operational semantics are listed in Tables 9 and 10, respectively. Three axioms and a rule describe the behavior of \( \text{out} \) and give it access to the internal state of an object. Three axioms and two rules describe overloaded function application, denoted by \( \bullet \). We use \( D \) to denote the \( \min_{i \in \{1, \ldots, n\}} \{D_i | C \vdash D \leq D_i\} \). In an overloaded application, we first reduce the term on the left to a \&-term, where each term is a \( \lambda \)-abstraction, taking into account that we do not reduce inside \( \lambda \)-abstractions; then the argument must be reduced to a tagged value and, lastly, the application is performed according to the index of the \&-term. Thus, the overloaded application is implemented by a call-by-value reduction. The standard \( \beta \)-rule is also formalized; in this case, the application
The operation \( \text{id} \) is used only for the selection while only the argument is passed to the \text{super} the application of overloaded functions. When the tagged value is a \text{super} it introduces the new association if there is no association for \( \text{id} \). When we dereference an identifier, \( \uparrow \text{id} \), we retrieve its associated value in the store. For the sake of simplicity, we also avoid to introduce the null references in the syntax, and then to formalize (and type) the run-time error resulting from the evaluation of a reference that points to no object (for a throughout treatment of null references we refer to [12]). In our context, the

\[
\begin{align*}
(C, S, \pi_i((G_1, \ldots, G_n))) &\Rightarrow (C, S, G_i) \\
(C, S, \text{out}^A (\text{in}^{A_2} (M))) &\Rightarrow (C, S, M) \\
(C, S, \text{out}^A (\text{super}^D (M))) &\Rightarrow (C, S, \text{out}^A (M)) \\
(C, S, (\lambda x^T . M) \cdot N) &\Rightarrow (C, S, M[x := N]) \\
(C, S, (M_1 \& (D_1 \rightarrow T_1 \ldots D_n \rightarrow T_n) M_2) \bullet G^D) &\Rightarrow (C, S, M_1 \bullet G^D) \quad \text{if } D_n \neq \top \\
(C, S, (M_1 \& (D_1 \rightarrow T_1 \ldots D_n \rightarrow T_n) M_2) \bullet G^D) &\Rightarrow (C, S, M_2 \cdot \text{ers}(G^D)) \quad \text{if } D_n = \top \\
(C, S, \text{let } A \leq A_1 \ldots A_n \text{ in } P) &\Rightarrow (C \cup (A \leq A_1) \cup \ldots \cup (A \leq A_n), S, P) \\
(C, S, \text{let } A \text{ hide } T \text{ in } P) &\Rightarrow (C, S, P) \\
(C, S, \text{id}^T := G) &\Rightarrow (C, S[\text{id}^T \leftarrow G], \text{nil}) \\
(C, S, \uparrow \text{id}^T) &\Rightarrow (C, S, S(\text{id}^T)) \\
(C, S, G; M) &\Rightarrow (C, S, M)
\end{align*}
\]
null-pointer error is represented by the attempt to evaluate \( \uparrow \text{id}^T \) when \( \text{id}^T \) has not been previously initialized by a suitable tagged value (object). In this case, \( S(\text{id}^T) \) is undefined: this is the only way in which a computation of a well typed program may get stuck, as formalized in the following theorems.

**Definition 5.2** (~\( \Rightarrow \)~). We denote by \( \Rightarrow \) the reflexive and transitive closure of \( \Rightarrow \).

**Definition 5.3** (Normal form). Given a configuration \((C, S, M)\), we say that \(M\) is in normal form with respect to \(C\) and \(S\) if \((C, S, M) \not\Rightarrow\). We say also that \((C, S, M)\) is a normal form configuration.

**Definition 5.4** (Error configuration). If \(M\) is in normal form with respect to \(C\) and \(S\), and \(M\) contains a term \(\uparrow \text{id}^T\) such that \(S(\text{id}^T)\) is undefined, then we say that \((C, S, M)\) is an error configuration.

**Theorem 5.1** (Subject reduction). Let \(P\) be a \(\lambda_{\text{object}}\) program. If \(C; S \vdash P : T\) and \((C, S, P) \Rightarrow (C', S', P')\), for some \(C', S', S\) and \(P'\); then \(C'; S \vdash P' : T'\) where \(C' \vdash T' \leq T\).

**Theorem 5.2** (Progress). Let \((C, S, M)\) be a normal form configuration such that it is not an error configuration. If \(M\) is a closed term and is typed by a (possibly unary) product of atomic types, then \(M\) is a tagged value.

The two above theorems show the standard type safety property for \(\lambda_{\text{object}}\). We point out that their proofs are fully presented in [16] (Sect. 8.2) for the basic calculus, that is \(\lambda_{\text{object}}\) without the imperative part. However, the extensions of both the proofs to our version of \(\lambda_{\text{object}}\) are straightforward; we just need to add a few cases, dealing with the imperative constructs, which do not present any significant technical difficulty.
6. Semantics of DOCS

We define the semantics of DOCS by translating DOCS into $\lambda_{\text{object}}$. The translation of DOCS expressions, statements, bodies (Sect. 6.1) and programs (Sect. 6.3) is defined by using the interpretation function $\llbracket(C, S, \Gamma, BT)\rrbracket$ which returns the $\lambda_{\text{object}}$ term associated to the argument. In particular, multi-methods are translated into $\lambda_{\text{object}}$ overloaded functions (Sect. 6.2).

We define the translation from DOCS expressions, (statements, bodies and programs) to Terms (the set of terms of $\lambda_{\text{object}}$) using the function $\llbracket(C, S, \Gamma, BT)\rrbracket$:

$\llbracket(C, S, \Gamma, BT)\rrbracket : \mathcal{L} \rightarrow \text{Envs} \rightarrow \text{Terms}$

where

- $\mathcal{L}$ is the set of DOCS expressions, statements, bodies and programs;
- $\text{Envs}$ is a tuple of environments as defined in Section 4.4 plus the body environment $BT$, mapping multi-methods’ definitions to class names;
- $\Gamma$ records the type of the identifiers and associates to self the current class, as explained later.

In the following we define the notion of consistency for the tuple $(C, S, \Gamma, BT)$.

Definition 6.1 ($(C, S, \Gamma, BT)$ consistency). The tuple $(C, S, \Gamma, BT)$ is consistent if

- $(C, S, \Gamma)$ is a consistent environment according to Definition 4.6;
- if $m \text{ branches } \ldots \text{end} \in BT$ then $m \in \text{Dom}(\Gamma)$.

Remark 6.1. We do not translate DOCS types into $\lambda_{\text{object}}$ types since, as we noted in Section 5.2, DOCS types are a subset of $\lambda_{\text{object}}$ types. The only difference between DOCS and $\lambda_{\text{object}}$ types concerns multi-types: in $\lambda_{\text{object}}$ the arrow types in a multi-type must be ordered according to the subtyping relation on input types (in particular the types that tag the &, see Sect. 6.2 for details). This ordering is relevant only for the semantics, in particular for the selection of an overloaded term, but not for the typing. Thus, we did not consider this ordering in DOCS since we only give the type system of DOCS, while, for semantics, we rely on a translation into $\lambda_{\text{object}}$; during this translation we must keep the ordering of multi-types into account.

In a tuple $(C, S, \Gamma, BT)$, $C$ and $S$ will be used only in the interpretation of static overloading and object instantiation (see Def. 6.3), while $BT$ will be used only for the interpretation of multi-methods (see Def. 6.7). In particular, during the interpretation of a DOCS term, $C$, S and BT will never change, while $\Gamma$ will change (e.g., during the interpretation of a method body). Thus, from now on, for simplicity, when $C$, $S$ and $BT$ are not necessary, we will write only $\Gamma$ instead of $(C, S, \Gamma, BT)$, and $\llbracket\Gamma\rrbracket$ should always be intended as $\llbracket(C, S, \Gamma, BT)\rrbracket$. 

While defining the translation function $\llbracket \langle C,S,\Gamma,BT \rangle \rrbracket$ we will assume that $\langle C,S,\Gamma,BT \rangle$ contains all the information needed to interpret DOCS programs. Indeed, in Section 6.3 we define the translation of well typed DOCS programs into $\lambda_{\text{object}}$, thus $\langle C,S,\Gamma,BT \rangle$ will contain all the informations collected while checking the declarative part of programs.

### 6.1. Interpretation of expressions, statements and bodies

We first introduce an auxiliary function, $\text{deref}$, that deals with references.

**Definition 6.2** (Definition of $\text{deref}$ function). Given a DOCS expression, $\text{exp}$, the function $\text{deref}(\llbracket \text{exp} \rrbracket)$ is defined on the syntax of $\text{exp}$:

1. $\llbracket x \rrbracket_\Gamma = x^{\Gamma(x)}$.
2. $\llbracket m \rrbracket_\Gamma = m^{\Gamma(m)}$.
3. $\llbracket \text{this} \rrbracket_\Gamma = \text{self}^{\Gamma(\text{this})}$.
4. $\llbracket \text{this}.f \rrbracket_\Gamma = (\text{out}^{\Gamma(\text{this})}(\text{self}^{\Gamma(\text{this})})).f$.
5. $\llbracket \text{receiver} \leftarrow m(\text{exp}) \rrbracket_{\langle C,S,\Gamma,BT \rangle} = [m]_\Gamma \bullet (\text{deref}(\llbracket \text{receiver} \rrbracket_\Gamma), \text{super}^A(\text{deref}(\llbracket \text{exp} \rrbracket_\Gamma)))$ for some $A$ such that $C;S;\Gamma \vdash \text{exp} : T$ and $A = \tilde{T}$.
6. $\llbracket \text{receiver} \leftarrow m(\text{exp}) \rrbracket_\Gamma = [m]_\Gamma \bullet (\text{deref}(\llbracket \text{receiver} \rrbracket_\Gamma), \text{deref}(\llbracket \text{exp} \rrbracket_\Gamma)))$.
7. $\llbracket \text{stmnt} \rrbracket_\Gamma = \text{id}_{\Gamma_1} = \text{exp}_1; \ldots; \text{id}_{\Gamma_n} = \text{exp}_n; \text{stmnt}; \text{return} \ \text{exp} \rrbracket_\Gamma = \text{id}_{\Gamma_1} = \text{deref}(\llbracket \text{exp}_1 \rrbracket_\Gamma); \ldots; \text{id}_{\Gamma_n} = \text{deref}(\llbracket \text{exp}_n \rrbracket_\Gamma); \lambda(x_1^\text{ref} A_1, \ldots, x_n^\text{ref} A_n).\llbracket \text{stmnt} \rrbracket_\Gamma; \llbracket \text{return} \ \text{exp} \rrbracket_\Gamma \cdot (\text{id}_{\Gamma_1} A_1, \ldots, \text{id}_{\Gamma_n} A_n)$ where $\Gamma = \Gamma_1[x_1 \leftarrow \text{ref} A_1] \ldots [x_n \leftarrow \text{ref} A_n]$ and $\text{id}_1, \ldots, \text{id}_n$ are fresh identifiers.
8. $\llbracket \text{new} A(\text{exp}_i^\text{ref} I) \rrbracket_{\langle C,S,\Gamma,BT \rangle} = \text{id}^B_i := \text{deref}(\llbracket \text{exp}_i \rrbracket_\Gamma)^{I_i}; \text{in}^A(\langle \text{id}_i^B_i \rangle^{I_i})$.
9. $\llbracket \text{super}(A) \rrbracket_\Gamma = \text{super}^A(\llbracket \text{this} \rrbracket_\Gamma)$.
10. $\llbracket \text{left} = \text{exp} \rrbracket_\Gamma = \llbracket \text{left} \rrbracket_\Gamma := \text{deref}(\llbracket \text{exp} \rrbracket_\Gamma)$.
11. $\llbracket \text{stmnt} \rrbracket_\Gamma = [\text{stmnt}_1]_\Gamma; [\text{stmnt}_2]_\Gamma$.
12. $\llbracket \text{return} \ \text{exp} \rrbracket_\Gamma = \text{deref}(\llbracket \text{exp} \rrbracket_\Gamma)$.

In some sense, the function $\text{deref}$ is the semantics counter- and complementary part of the $\text{ref} \ \text{destructor}$ operator in the type system (Def. 4.1) and returns the content of mutable instance variables.

In the following we define the translation (interpretation) of expressions, statements and bodies into $\lambda_{\text{object}}$. 

**Definition 6.3** (Interpretation of DOCS expression, statements and bodies). We define the translation of expression, statements and bodies as in the following:

1. $\llbracket x \rrbracket_\Gamma = x^{\Gamma(x)}$.
2. $\llbracket m \rrbracket_\Gamma = m^{\Gamma(m)}$.
3. $\llbracket \text{this} \rrbracket_\Gamma = \text{self}^{\Gamma(\text{this})}$.
4. $\llbracket \text{this}.f \rrbracket_\Gamma = (\text{out}^{\Gamma(\text{this})}(\text{self}^{\Gamma(\text{this})})).f$.
5. $\llbracket \text{receiver} \leftarrow m(\text{exp}) \rrbracket_{\langle C,S,\Gamma,BT \rangle} = [m]_\Gamma \bullet (\text{deref}(\llbracket \text{receiver} \rrbracket_\Gamma), \text{super}^A(\text{deref}(\llbracket \text{exp} \rrbracket_\Gamma)))$ for some $A$ such that $C;S;\Gamma \vdash \text{exp} : T$ and $A = \tilde{T}$.
6. $\llbracket \text{receiver} \leftarrow m(\text{exp}) \rrbracket_\Gamma = [m]_\Gamma \bullet (\text{deref}(\llbracket \text{receiver} \rrbracket_\Gamma), \text{deref}(\llbracket \text{exp} \rrbracket_\Gamma)))$.
7. $\llbracket \text{stmnt} \rrbracket_\Gamma = \text{id}_{\Gamma_1} = \text{exp}_1; \ldots; \text{id}_{\Gamma_n} = \text{exp}_n; \text{stmnt}; \text{return} \ \text{exp} \rrbracket_\Gamma = \text{id}_{\Gamma_1} = \text{deref}(\llbracket \text{exp}_1 \rrbracket_\Gamma); \ldots; \text{id}_{\Gamma_n} = \text{deref}(\llbracket \text{exp}_n \rrbracket_\Gamma); \lambda(x_1^\text{ref} A_1, \ldots, x_n^\text{ref} A_n).\llbracket \text{stmnt} \rrbracket_\Gamma; \llbracket \text{return} \ \text{exp} \rrbracket_\Gamma \cdot (\text{id}_{\Gamma_1} A_1, \ldots, \text{id}_{\Gamma_n} A_n)$ where $\Gamma = \Gamma_1[x_1 \leftarrow \text{ref} A_1] \ldots [x_n \leftarrow \text{ref} A_n]$ and $\text{id}_1, \ldots, \text{id}_n$ are fresh identifiers.
8. $\llbracket \text{new} A(\text{exp}_i^\text{ref} I) \rrbracket_{\langle C,S,\Gamma,BT \rangle} = \text{id}^B_i := \text{deref}(\llbracket \text{exp}_i \rrbracket_\Gamma)^{I_i}; \text{in}^A(\langle \text{id}_i^B_i \rangle^{I_i})$.
9. $\llbracket \text{super}(A) \rrbracket_\Gamma = \text{super}^A(\llbracket \text{this} \rrbracket_\Gamma)$.
10. $\llbracket \text{left} = \text{exp} \rrbracket_\Gamma = \llbracket \text{left} \rrbracket_\Gamma := \text{deref}(\llbracket \text{exp} \rrbracket_\Gamma)$.
11. $\llbracket \text{stmnt} \rrbracket_\Gamma = [\text{stmnt}_1]_\Gamma; [\text{stmnt}_2]_\Gamma$.
12. $\llbracket \text{return} \ \text{exp} \rrbracket_\Gamma = \text{deref}(\llbracket \text{exp} \rrbracket_\Gamma)$. 

We note that:

– in field selection (4) `out` unpacks `self` into a record and it selects its `f` component;

– a method invocation ((5), (6)) is translated in an application of an overloaded function: thus, in the operational semantics of the translated term, the selection of the right branch is driven by the dynamic type of both the `receiver` and the static (5) or dynamic (6) parameter `exp` (see Tab. 9). Note that, in the case of static overloading, we force the argument to its static type; in order to do this we need the type of the argument as inferred by the type system. In this case we need to refer to the type system (`C`, `S` and `Γ`). This corresponds to the fact that static overloading method selection is made at compile time;

– a `DOCS` body (7) is composed by a (possibly empty) sequence of local variable declarations and initializations \( x_1 = \text{exp}_1; \ldots; x_n = \text{exp}_n \), a (possibly empty) sequence of statements and a `return`. The corresponding \( λ_{\text{object}} \) term is a \( λ \)-abstraction

\[
λ(\text{x}_1^{\text{ref} \ A_1} \ldots \text{x}_n^{\text{ref} \ A_n}) \cdot [\text{body}]_{Γ'}
\]

applied to the fresh identifiers `id_1^{A_1} \ldots id_n^{A_n}` initialized with the evaluation of the corresponding expressions: thus, `id_1^{A_1} \ldots id_n^{A_n}` will replace the occurrences of the `DOCS` variables `x_1, \ldots, x_n` in the first step of the evaluation of the `λ_{object}` term obtained by translation. As a consequence, during evaluation, we will manage only identifiers `id` instead of `DOCS` variables and the store `S` will associate to `id` values of type `A` according to rule \( (C, S, \text{id}^{A} := G) \Rightarrow (C, S[\text{id}^{A} ← G], \text{nil}) \);

– the creation of an object (8) is translated in:

\[
\cdot \text{a sequence of assignments of the shape } \text{id}_i^{B_i} := \text{deref}([\text{exp}_i]) \text{ to assign to each identifier } \text{id}_i^{B_i} \text{ the evaluation of the } i^{\text{th}} \text{ argument expression;}
\]

\[
\cdot \text{the construction of an object of type } A \text{ using the } \text{in} \text{ operator with a record which associates to each field } f_i \text{ the corresponding identifier } \text{id}_i.
\]

Thus, as in point (7), we will manage identifiers `id^{B}` instead of field labels, during the evaluation of the translated terms. As in the previous case all the identifiers are fresh, thus each object will have its own state.

\begin{itemize}
  \item \textbf{DOCS} multi-methods are translated into \( λ_{\text{object}} \) overloaded functions. We briefly discuss the informal idea and some crucial points, before giving the formal definitions.
\end{itemize}
Let us consider, for example, the following class definition:

```plaintext
class A{
  m branches
  T (B x){body_1};
  T (C x){body_2};
}
```

Then, the multi-method \( m \) will be translated into the following \( \lambda \) object overloaded function:

\[
\epsilon \& \{ (A \times B) \rightarrow T \lambda (self^A, x^B) \cdot M_1 \\
\& \{ (A \times B) \rightarrow T, (A \times C) \rightarrow T \lambda (self^A, x^C) \cdot M_2 \\
\& \{ (A \times D) \rightarrow T, (B \times D) \rightarrow T \lambda (self^B, x^D) \cdot M_2 \\
\& \{ (A \times D) \rightarrow T, (B \times D) \rightarrow T, (C \times D) \rightarrow T \lambda (self^C, x^D) \cdot M_3
\] 

where each branch of the multi-method is translated in a \( \lambda \) abstraction (\( M_1 \) and \( M_2 \) are the \( \lambda \) object terms obtained by translating the definition of the bodies of the related branches). The \( \lambda \) abstraction has parameters \( self^A \) (which represents the receiver of the message, \( \text{this} \) is translated into \( self^A \)) and \( x^C \) (which represents the parameter of type \( C \) of the method) and \( A \) denotes the current class.

All the \( \lambda \) abstractions are then concatenated through the conjunctions \& which are tagged with the type of the corresponding branch (the one on the right of \&). When tagging the conjunctions with the multi-type a problem arises. Suppose we have three classes \( A, B \) and \( C \) with \( C \) inheriting both from \( A \) and \( B \). Suppose also that a branch with type parameter \( D \) for multi-method \( m \) is defined both in \( A \) and \( B \); then by conditions of well formedness on multi-types we need to redefine the branch also in \( C \) (see condition (iiid) in Def. 4.5). In DOCS the simplest solution is to define the branches in \( A \) and \( B \) and then to add the branch to \( C \) at the moment of \( C \) definition:

```plaintext
class A{
  m branches
  T (D x){body_1};
}
```

```plaintext
class B{
  m branches
  T (D x){body_2};
}
```

```plaintext
class C : A, B{
  m branches
  T (D x){body_3};
}
```

Hence, the corresponding translation of \( m \) in \( \lambda \) object would be:

\[
\epsilon \& \{ (A \times D) \rightarrow T \lambda (self^A, x^D) \cdot M_1 \\
\& \{ (A \times D) \rightarrow T, (B \times D) \rightarrow T \lambda (self^B, x^D) \cdot M_2 \\
\& \{ (A \times D) \rightarrow T, (B \times D) \rightarrow T, (C \times D) \rightarrow T \lambda (self^C, x^D) \cdot M_3
\]
where $M_1$, $M_2$ and $M_3$ are the $\lambda_{\text{object}}$ terms obtained by translating the bodies of the branches of method $m$.

However, in the above overloaded term the second type index $\langle (A \times D) \rightarrow T, (B \times D) \rightarrow T \rangle$ is not well formed since it does not satisfy condition (d) in Definition 4.5: if we consider the input type $(C \times D)$ there is not a most specific branch applying. The problem is due to the fact that, when we translate the branch defined in class $B$, we have not already encountered class $C$: for this reason the branch written to solve ambiguities due to multiple inheritance has to precede at least one of the branches of its direct ancestors$^4$. Indeed, the following overloaded term is well formed:

$$\epsilon \& \langle (C \times D) \rightarrow T \rangle \lambda (\text{self}^C, x^D) \cdot M_3$$
$$\& \langle (C \times D) \rightarrow T, (B \times D) \rightarrow T \rangle \lambda (\text{self}^B, x^D) \cdot M_2$$
$$\& \langle (C \times D) \rightarrow T, (B \times D) \rightarrow T, (A \times D) \rightarrow T \rangle \lambda (\text{self}^A, x^D) \cdot M_1$$

where the $\lambda$-abstractions are concatenated in an ascending order w.r.t. the subtyping relation on the pair type of the $\lambda$-abstraction. Concluding, in order to build well typed overloaded functions (namely overloaded functions where all the &'s are indexed with well formed multi-types) we need to concatenate the branches in a suitable order.

Thus, to translate $DOCS$ multi-methods we have to scan the program from the bottom to the top to build well typed overloaded terms using the following functions:

- $\mathcal{M}[p]_{(C,S,F,BT)}(m)$ returns the overloaded term associated to the message $m$ by the definition in $p$;
- $T[p](m)$ returns the (pre)type that indexes the & concatenating the branches of the translation of $m$. Obviously, if $p$ is well typed we expect that $\mathcal{M}[p]_{(C,S,F,BT)}(m) : T[p](m)$ (see Lem. 7.5);

Summarizing, every method is translated in a $\lambda$-abstraction (always parameterized over $\text{self}^A$, where $A$ is the class of the receiver), and method names, i.e., messages, become identifiers of these functions. So, in $\lambda_{\text{object}}$ each method actually becomes a function parameterized also over the receiver object$^5$.

$^4$The technical reason of this problem concerns the features of the $\lambda\mu$ calculus (on which $\lambda_{\text{object}}$ is based): $\lambda\mu$ totally lacks of the notion of time. Atomic types are given all at once and there is no perception of the temporal dependence of type definitions. Further comments on this topic can be found in [18].

$^5$This makes $\lambda_{\text{object}}$ pretty close to implementations of mainstream object-oriented languages where methods are basically translated into functions where the implicit object, $\text{this}$, is implicitly passed as the first parameter, see, e.g., [31].
Now, let us consider the case of an invocation of the multi-method $m$ and its translation (Def. 6.3):

$$\new{\cdot} m \cdot (\text{deref}(\new{\cdot} C), \text{deref}(\new{\cdot} D)) =$$

$$\new{\cdot} m \cdot (\text{in}^{C}(), \text{in}^{D}()) =$$

$$\epsilon_{\&}(C \times D) \rightarrow T \lambda(self^{C}, x^{D}).M_{3}$$

$$\&\{(C \times D) \rightarrow T, (B \times D) \rightarrow T\} \lambda(self^{B}, x^{D}).M_{2}$$

$$\&\{(C \times D) \rightarrow T, (B \times D) \rightarrow T, (A \times D) \rightarrow T\} \lambda(self^{A}, x^{D}).M_{1} \cdot (\text{in}^{C}(), \text{in}^{D}())$$

According to the semantic rule for overloaded function application (see Tab. 9), the branches of the function are scanned until the input of the last element indexing the $\&$ coincides with the type of the input (condition $D = D_{n}$ in the semantic rule); then, in the example, the selected branch is $\lambda(self^{C}, x^{D}).M_{3}$ since the input type is $(C \times D)$.

This will gives to the invocation of $m$ the expected behavior. Indeed, in a direct operational semantics of DOCS the rule for multi-method invocation would be very simple (see, for instance, [33]):

1. extract the set of the associated implementations of $m$ matching the type of the receiver and the parameters;
2. choose the most specific one in the above set;
3. evaluate the chosen implementation in the suitable context.

Our interpretation into $\lambda_{\text{object}}$, mapping the multi-method in an overloaded function and its invocation in an application term, formalizes and implements the above operational rule in a typed $\lambda$-calculus.

Finally, we remark that:

- DOCS typing rules guarantee that there always exist one and only one most specialized branch to be selected (see Sect. 4.4) w.r.t an input type $I$;
- the translation into $\lambda_{\text{object}}$ guarantees that the body of this most specialized version is associated to the branch of the overloaded function in which the last arrow type (indexing the $\&$) has input type $I$ (see Defs. 6.6 and 6.7).

Now we formally define the interpretation function $\mathcal{M}$ after some auxiliary definitions.

**Auxiliary definitions.** The functions $T$ and $\mathcal{M}$ use the auxiliary function $F$ to implement copy semantics of inheritance. Given a class name $A$ and a method name $m$, $F(A, m)$ returns a set of branch types $(A_{i} \times B_{k}) \rightarrow T_{k}$ where $A_{i}$ is either $A$ or a superclass of $A$ in the hierarchy and $B_{k} \rightarrow T_{k}$ is the type of a branch definition of $m$ in $A_{i}$. So, the branch types collected in this set are both the ones inherited and the ones defined in $A$, abstracting away from branches that
are specifically introduced by copy semantics inheritance. Indeed \( F(A, m) \) scans the branches of \( m \) defined in \( A \) and recursively all the superclasses of \( A \), thus the program \( p \) containing the definition of \( A \) (and its superclasses) is an implicit parameter of \( F \). However, we shall omit such argument since it will be always clear from the context. Note that \( F(A, m) \) is simply a set of types (which may not be well formed) and not a multi-type. For this reason we denote with \textbf{MultiTypes} the codomain of \( F \), instead of \textbf{Types}.

**Definition 6.4** (Definition of \( F \) and \( L \)).

Let

\[
p = \text{class } A : A_h \stackrel{h \in H}{\prod} C_i \ {i \in I} \ {m_j} \ \text{branches} \ T_{k_j} (B_{k_j}, x_{k_j}) \ {k_j \in K_j} \ \text{end} \ {j \in J} \end{equation}

then the function

\( F : (\text{ClassNames} \times \text{MethodNames}) \to \text{MultiTypes} \)

is defined as follows:

\[
F(A, m_j) = \{(A \times B_{k_j}) \to T_{k_j} \ {k_j \in K_j}\} \cup \\
\{(A_h \times B_l) \to T_l \mid (A_h \times B_l) \to T_l \in F(A_h, m_j), h \in H, l \notin K_j\}
\]

where \( l \notin K_j \) assures that the branch \( B_l \to T_l \) is not defined in class \( A \).

Moreover,

\[
L(F(A, m)) = \{l \mid (A_i \times B_l) \to T_l \in F(A, m)\}
\]

\( L(F(A, m)) \) permits accessing the types of the signatures of branch definitions of \( m \) in the hierarchy of \( A \). By definition of \( F \), we have the following property:

**Property 6.1.** Let \( A \) be a class name and \( m \) a method name, then for each \( (A_i \times B_j) \to T_j \in F(A, m) \), we have:

- either \( A_i \equiv A \) and there is a branch of \( m \) in \( A \) with type \( B_j \to T_j \) or;
- \( A_i \leq A_k \) and \( (A_k \times B_j) \to T_j \in F(A_k, m) \) and for each \( (A_k \times B_l) \to T_l \in F(A, m) \) if \( A_i \leq A_k \) and \( i \neq k \) then \( l \neq j \).

This property ensures that we collect all the types of the branches defined in a class and all the most redefined versions inherited from superclasses (that are not redefined by the class itself).

The following definition of permutation orders the types within multi-types in ascending order according to the subtype relation, \textit{i.e.}, leftmost arrow types are those with smallest input types. This is a crucial issue to guarantee that multi-method selection will work correctly (see Rem. 5.1).

**Definition 6.5** (Permutation). Given an overloaded type \( \{I_h \to T_h\} \ {h \in H} \), we denote by \( \{I_{\sigma(h)} \to T_{\sigma(h)}\} \ {h \in H} \) the type obtained applying to the indexes the permutation \( \sigma \) that orders the \( I_h \)'s in such a way that \( I_h \leq I_k \Rightarrow \sigma(h) \leq \sigma(k) \).
We now define $T$, $M$ to build multi-types. We assume that the type environment $\Gamma$ associates to each multi-method $m$ in $p$ its equivalent multi-type $T[p](m)$.

**Definition 6.6** (Definition of $T$).

Let

\[
p = \text{class } A : A_h, h \in H \{ \\
C_i, f_i; \ i \in I \\
m_j \text{ branches } T_{k_j} (B_{k_j}, x_{k_j}) \{ \text{body}_{k_j} \}; \ k_j \in K_j \end{cases} \end{align*}
\]

then we define

\[
T[p](m) = \begin{cases} 
T[p’](m_j) \oplus (A \times B_{\sigma(1)}) \rightarrow T_{\sigma(1)} \oplus \cdots \oplus (A \times B_{\sigma(n)}) \rightarrow T_{\sigma(n)} \\
T[p’](m) \text{ otherwise}
\end{cases}
\]

where $L(F(A, m_j))$ is defined in Definition 6.4 and $\oplus$ in Definition 5.1.

$T[p](m)$ is the function that returns $\{\}$ in all the other cases.

Summarizing, for each multi-method declaration, in each class definition, $T$ performs the following steps:

1. it lists the arrow types of the branches defined in the class and in its super classes for a multi-method $m_j$ using $L(F(A, m_j))$;
2. for each arrow type collected in point (1) it replaces, in the input type, the first element of the pair with the type of the current class: this substitution implements copy semantics of inheritance at typing level;
3. it orders the arrow types so collected according to an ascending order on the parameter types w.r.t. subtyping relation: this is done by applying the permutation $\sigma$ on the indexes of the second elements of the input type pairs;
4. finally, it concatenates the resulting multi-type to the multi-type obtained by translating the rest of the program $(T[p’](m))$, namely to remaining class definitions. The concatenation is performed through the operator $\oplus$.

Let us note that the multi-type obtained by the function $T[p](m)$ is the same multi-type that $\Gamma_p$ associates to $m$ in $DOCS$ typing. The only difference is that the arrow types in the multi-type $T[p](m)$ are ordered while in $DOCS$ typing the order of arrow types in a multi-type is not relevant (Rem. 6.1). For instance, if we have three branches for the multi-method $m$ in class $A$ with parameters $B_1$, $B_2$ and $B_3$ (and return type $B’$), where $B_3 \leq B_2 \leq B_1$, since $\sigma(3) \leq \sigma(2) \leq \sigma(1)$, then the types indexing the & will be generated in this order: $(A \times B_3) \rightarrow B’$, $(A \times B_2) \rightarrow B’$, $(A \times B_1) \rightarrow B’$. Since conditions of well formedness on multi-types do not rely on the order of arrow types in multi-types (indeed multi-types are sets), it is easy to verify that if (Def. 4.5) $\Gamma_p(m)$ is well formed also $T[p](m)$ is well formed. Thus, from the typing point of view, $\Gamma_p(m)$ and $T[p](m)$ can be considered equivalent.

**Property 6.2.** Let $p$ be a well typed $DOCS$ program, then $\forall m \in \text{MethNames}(p)$, $T[p](m)$ is well formed.
Proof. By contradiction, it is easy to show that if $T[p](m)$ is not well formed, then $p$ cannot be well typed. \qed

In the following we define the function $\mathcal{M}$ that is devoted to translate multi-methods of a program into overloaded functions of $\lambda$-object.

**Definition 6.7** (Definition of $\mathcal{M}$). Let

$$p = \text{class } A : A_h \{ C_i \text{, } f_i \in \ell \}$$

$m_j$ branches $T_{k_j}(B_{k_j} \{ \text{body}_{k_j} \}; \ k_j \in K_j \text{ end } j \in J \}$

be a DOCS program. We define $\mathcal{M}[p](C,S,T,B,T)(m)$ as follows:

1. if $m = m_j$ for some $j \in J$, then let

   $L(\mathcal{F}(A,m_j)) = \{1, \ldots, n\}$ and

   $\mathcal{F}(A,m_j) = \{(D_{B_{1_i}} \times B_{1_i}) \rightarrow T_{1_i}, \ldots, (D_{B_{n_i}} \times B_{n_i}) \rightarrow T_{n_i}\}$

   where each $D_{B_{i_j}}$ is either $A$ or one of its superclasses. Then,

   $$\mathcal{M}[p](C,S,T,B,T)(m) =$$

   $$(\ldots(\mathcal{M}[p](C,S,T,B,T)(m) \& \mathcal{T}[p'](m) \oplus \{(A \times B_{\sigma(1)}) \rightarrow T_{\sigma(1)}\}) \ldots)$$

   $$(\lambda(\text{self}^A \cdot x^{B_{\sigma(1)}} \cdot [BT(m_{j_i}, \sigma_{B_{\sigma(1)}}) \{ \text{this} \rightarrow A \cdot x^{B_{\sigma(1)}} \rightarrow B_{\sigma(1)} \}] \ldots \& \mathcal{T}[p'](m) \oplus \{(A \times B_{\sigma(n_i)}) \rightarrow T_{\sigma(n_i)}\}))$$

2. $\mathcal{M}[p](C,S,T,B,T)(m) = \mathcal{M}[p'](m)$ otherwise

3. finally, $\mathcal{M}[\subseteq](C,S,T,B,T)(m)$ is the function that returns $\epsilon$ in all the other cases.

$\mathcal{M}$ builds the overloaded term associated to each multi-method defined in a DOCS program. Each branch declaration in a class $A$ is translated in a $\lambda$-abstraction. The body of the $\lambda$-abstraction is obtained using the body table $BT$ (see Def. 4.7). Note that $\mathcal{M}$ searches for the definition of the branch body using the class where the method has been originally defined. In this way, if the method is inherited from a superclass, the body is “copied” in the current class. Finally, the $\lambda$-abstractions are concatenated by $\&$ (using the permutation $\sigma$).

In the definition of $\mathcal{M}$ we did not consider (mutually) recursive methods. The definition of $\mathcal{M}$ can be easily extended to cope with recursive methods, using the fix-point technique as suggested in [15], but it is not particularly relevant in our context so we skip this technicality that would only make proofs more complex. Note, however that the definition of $T$ do not need to be changed.

### 6.3. Semantics of DOCS programs

We define the interpretation of any program $p$, $\Omega(p)$, as a composition of the interpretation of its main body and multi-methods, in the specific environment $\langle C_p, S_p, \Gamma_p, BT_p \rangle$ induced by the declarations of $p$. 
**Definition 6.8** (Type environment $\Gamma_T[p]$). Let $p$ be a DOCS program. Then we define the type environment $\Gamma_T[p]$ as follows:

$$\Gamma_T[p] = \{m : T[p](m) \mid m \in \text{MethNames}(p)\}.$$ 

For simplicity, since $\Gamma_T[p]$ is equivalent to the environment $\Gamma_p$ used to type check DOCS programs (see Def. 4.11), modulo the order of arrow types in multi-types (namely $\Gamma_T[p](m) \approx \Gamma_p(m) \forall m \in \text{MethNames}(p)$), from now on we will use $\Gamma_p$ to denote $\Gamma_T[p]$. This abuse of notation will simplify the proof of main properties.

Now, we can define the interpretation of programs.

**Definition 6.9** (Interpretation of DOCS programs). Let $p = \text{classdef}^* \text{ body}$ be a well-typed DOCS program, and

$$\text{classdef}^* = \text{ class } A_h : A_1, \ldots, A_n \{$$

$$C_i f_i; \ i \in I_h$$

$$m_j \text{ branches}$$

$$T_k (B_k x_k) \{\text{body}_{k}; \ k \in K_j$$

$$\} \}; h \in H$$

$$\}$$

Then we define

$$\Omega(p) \overset{\text{def}}{=} [\text{body}_1](C_p, S_p, T_p, \Gamma_p)[m_j]^{\Gamma_p(m_j)} := M[p](C_p, S_p, T_p, \Gamma_p)(m_j)$$

for each $j \in J_h$ and $h \in H$.

To clarify the use of the type environment $\Gamma_p$, we remark a key aspect of the translation of DOCS multi-methods into $\lambda_{\text{object}}$ overloaded functions. Let us consider the following code:

```plaintext
class A{
    m branches
    T (B x){body_1};
    end
};

class B{
    n branches
    T (B x){body_2};
    end
};
```

The method $m$ defined in class $A$ has a branch with parameter of type $B$. When translating programs in $\lambda_{\text{object}}$, methods definitions are put “inline” with methods declarations (see Def. 6.7). The problem is that the body of the branch $A::m(B x)$ uses the method $n$ defined in class $B$, which has not already been translated. Thus, $n \notin \text{Dom}(\Gamma)$ and cannot be interpreted into $\lambda_{\text{object}}$ (indeed
\([n]_t = n^{\Gamma(n)}\). We adopt the following solution:
- we first build the multi-types associated to each multi-method defined in
  a program \(p\) with the function \(T[p](m_i)\) (see Def. 6.6);
- the multi-types \(T[p](m_i)\) are collected in the environment \(\Gamma_p\) which is
  used by the translation function \(\Omega\).

From an algorithmic point of view, the translation of a \textit{DOCS} program \(p\) into \(\Omega(p)\)
can be seen as a procedure consisting of the following steps:
1. the prescan of the declarative part of \(p\) (\textit{classdef } *) constructs
   \(\langle C_p, S_p, \Gamma_p, BT_p \rangle\), and associates to each method name \(m\) in \(p\) its (ordered)
   multi-type \(T[p](m)\);
2. the interpretation of the program translates the main body of \(p\) using
   \(\langle C_p, S_p, \Gamma_p, BT_p \rangle\);
3. the interpretation of multi-methods into overloaded functions associates to
   each multi-method name \(m^{\Gamma_p(m)}\) the \(\lambda\)-\textit{object} term \(M[p]\(\langle C_p, S_p, \Gamma_p, BT_p \rangle(m)\)
   (this step scans again the program \(p\) from the beginning since it works on
   the declarative part). Finally, each method identifier \(m^{\Gamma_p(m)}\) is replaced by
   \(M[p]\(\langle C_p, S_p, \Gamma_p, BT_p \rangle(m)\).

The semantics of \(p\) is intended to be defined as the operational semantics of the
\(\lambda\)-\textit{object} term \(\Omega(p)\), according to the rules of Tables 9 and 10 (Sect. 5.3).

\textbf{Definition 6.10 (Semantics of \textit{DOCS} programs).} Let \(p\) be a well-typed \textit{DOCS}
program. The semantics of \(p\) is the evaluation of the \(\lambda\)-\textit{object} term \(\Omega(p)\), starting
from the type constraints \(C_p\) and an empty store, \textit{i.e.},
\[(C_p, \emptyset, \Omega(p)).\]

Let us note that in [15] the type constraint system \(C\) used in the evaluation of a
program is built along the reduction by the declarations \textbf{let } \(A \leq A_1, \ldots, A_n\ \textbf{in } p\)
(indeed, the evaluation of a program starts from an empty \(C\)). In our case, instead,
since we need \(C_p\) to translate the program into a \(\lambda\)-\textit{object} term, we avoid the
generation of all the \textbf{let}'s: these would only generate a type constraint system
equal to \(C_p\).

\section{DOCS IS TYPE SAFE}

In this section we prove that any well typed \textit{DOCS} program is translated into
a well-typed \(\lambda\)-\textit{object} program, namely, types are preserved upon translation (\textit{type preservatiom}). Thus, \textit{type safety} of \(\lambda\)-\textit{object} results in an indirect proof of type
safety for the language \textit{DOCS} (Theorem 7.1). In particular this proves the cor-
rectness of \textit{DOCS} typing rules.

Type safety for \textit{DOCS} programs, formalized in Theorem 7.2, relies on:
- type preservation under translation for expressions and statements
  (Lem. 7.2);
- type preservation under translation for bodies (Lem. 7.3);
- type preservation under translation for programs (Thm. 7.1);
- type correctness of the substitutions

\[ [m_j \Gamma_p (m_j) := M[p]_{(s_p, \Gamma_p, BT_p)}(m_j) \; j \in J] \]

used in the interpretation (Lem. 7.5).

**Lemma 7.1** (Type of `deref`). Let `exp` be a DOCS expression. Then for every consistent environment \( \langle C, S, \Gamma, BT \rangle \)

\[
\text{if } C; S; \Gamma \vdash \text{exp} : T \; \text{then} \; C; S \vdash \text{deref}([\text{exp}]_{(C, S, \Gamma, BT)}) : \tilde{T}.
\]

**Proof.** By structural induction on `exp`. The only interesting case is `exp \equiv \text{methinvok}`:

\[
\text{deref}([\text{methinvok}]_{\Gamma}) = [\text{methinvok}]_{\Gamma} \Rightarrow^* [\text{return} \; \text{exp}]_{\Gamma} = \text{deref}([\text{exp}]_{\Gamma});
\]

then the result follows from the induction hypothesis. \( \square \)

**Lemma 7.2** (Type preservation for expression and statements). Let `t` be a well typed DOCS expression or statement. Then for every consistent environment \( \langle C, S, \Gamma, BT \rangle \)

\[
\text{if } C; S; \Gamma \vdash t : T \; \text{then} \; C; S \vdash [t]_{(C, S, \Gamma, BT)} : T.
\]

**Proof.** By structural induction on `t`. Let us consider the case `t \equiv \text{receiver} \leftarrow m(\text{exp}')`.

By rule `[DOVMETHCALL]` we have:
- \( C; S; \Gamma \vdash m : \{ I_k \rightarrow U_k \} \; k \in K \);
- \( C; S; \Gamma \vdash \text{receiver} : T \);
- \( C; S; \Gamma \vdash \text{exp}' : T' \);
- \( I_h = \min_{k \in K} \{ I_k | C \vdash (\tilde{T} \times \tilde{T'}) \leq I_k \} \)

\[
[\text{receiver} \leftarrow m(\text{exp}')]_{\Gamma} = [m]_{\Gamma} \cdot (\text{deref}([\text{receiver}]_{\Gamma}), \text{deref}([\text{exp}']_{\Gamma}))
\]

Then we have:
- \( C; S \vdash [m]_{\Gamma} : \{ I_k \rightarrow U_k \} \; k \in K \);
- \( C; S \vdash [\text{receiver}]_{\Gamma} : T \) by induction hypothesis;
- \( I_h = \min_{k \in K} \{ I_k | C \vdash (\tilde{T} \times \tilde{T'}) \leq I_k \} \).

Then, the thesis follows from the \( \lambda_{\text{object}} \) rule `[\lambdaELIM]`.

The case `t \equiv \text{receiver} \leftarrow m(\text{exp}')` is similar. The other cases are trivial. \( \square \)

**Lemma 7.3** (Type preservation for bodies). Let

\[
\{ A_1 \; x_1 = \text{exp}_1; \ldots; A_n \; x_n = \text{exp}_n; \text{stmt} ; \text{return} \; \text{exp} \}
\]

be a DOCS body. Then for every consistent environment \( \langle C, S, \Gamma, BT \rangle \) if:

\[
C; S; \Gamma \vdash \{ A_1 \; x_1 = \text{exp}_1; \ldots; A_n \; x_n = \text{exp}_n; \text{stmt} ; \text{return} \; \text{exp} \} : T
\]

then

\[
C; S \vdash [(A_1 \; x_1 = \text{exp}_1; \ldots; A_n \; x_n = \text{exp}_n; \text{stmt} ; \text{return} \; \text{exp})]_{(C, S, \Gamma, BT)} : T
\]
Proof. \[\{A_1, x_1 = \exp; \ldots; A_n, x_n = \exp; \text{return}; \exp\}_\Gamma = \lambda(x_1^{ref} A_1, \ldots, x_n^{ref} A_n).[\text{return} \exp]_\Gamma; [\text{return} \exp]_\Gamma;(id_1 A_1 \ldots id_n A_n).\]

By DOCS rule \[\text{Body}\] and Lemma 7.2 we have \(C; S \vdash [\text{return}\ exp]_\Gamma; [\text{return}\ exp]_\Gamma: T\).

The thesis comes from \(\lambda\_\text{object}\) rules \(\rightarrow\text{INTRO}\) and \(\rightarrow\text{ELIM}\). \(\square\)

The following lemma is a technical step for proving Lemma 7.5: it states that the types of the branches of the overloaded function \(M[p]_{(C,S,\Gamma,BT)}(m)\) have the same types labeling the conjunctions \(\land\).

**Lemma 7.4 (Well typedness of \(M\) branches).** Let \(p\) be a well typed DOCS program. Let \(M[p]_{(C,S,\Gamma,BT)}(m)\)

\[
\lambda(self^A, x^{B_\sigma(1)}).[BT(m_j, D_{B_\sigma(1)}), B_\sigma(1)]_\Gamma[\text{this}^A x^{B_\sigma(1)} e B_\sigma(1)]
\]

be the interpretation in \(\lambda\_\text{object}\) of the methods defined in class \(A\) according to Definition 6.7, where

\[L(F(A, m_j)) = \{1, \ldots, n\} \quad \text{and} \quad F(A, m_j) = \{(D_{B_1} \times B_1) \to T_1, \ldots, (D_{B_n} \times B_n) \to T_n\}\]

where each \(D_{B_i}\) is either \(A\) or one of its superclasses.

Then:

\(C; S \vdash \lambda(self^A, x^{B_\sigma(1)}).[BT(m_j, D_{B_\sigma(1)}), B_\sigma(1)]_\Gamma[\text{this}^A x^{B_\sigma(1)} e B_\sigma(1)] : (A \times B_i) \to T_i\)

for each \(l \in \{\sigma(1), \ldots, \sigma(n)\}\).

**Proof.** The body of each lambda abstraction is built by

\[BT(m_j, D_{B_\sigma(1)}), B_\sigma(1)]_\Gamma[\text{this}^A x^{B_\sigma(1)} e B_\sigma(1)] = [\text{body}]_\Gamma[\text{this}^A x^{B_\sigma(1)} e B_\sigma(1)].\]

Since \(p\) is well typed we have \(C; S; \Gamma \vdash \text{body} : U\) with \(U \leq T_i\) (from DOCS rule \[\text{Program-class}\]). By Lemma 7.2. \(C; S \vdash [\text{body}]_\Gamma[\text{this}^A x^{B_\sigma(1)} e B_\sigma(1)] : U\). The result comes from \(\lambda\_\text{object}\) rule \(\rightarrow\text{INTRO}\). \(\square\)

Now, in order to guarantee that the substitutions

\(m_j^{\Gamma_p(m_j)} := M[p]_{(C_p, S_p, \Gamma_p, BT_p)}(m_j) \quad \forall j\)

preserve well typedness we have to
prove that $\mathcal{M}[p]_{(C_p, S_p, \Gamma_p, \mu p)}(m_j)$ has type $\mathcal{T}[p](m_j)$ for all $j \in \text{MethNames}(p)$, since $\Gamma_p(m_j) = \mathcal{T}[p](m_j)$.

**Lemma 7.5** (Type correctness of method translation). Given a well typed DOCS program $p$

$$\iff p : T \ (i.e., C_p; S_p; \Gamma_p \vdash p : T)$$

then for each $m \in \text{MethNames}(p)$

$$C_p; S_p \vdash \mathcal{M}[p]_{(C_p, S_p, \Gamma_p, \mu p)}(m) : \mathcal{T}[p](m).$$

**Proof.** By induction on the cardinality $N$ of classdef $^*$. 

- $(N = 1)$, then $p$ is defined as in the following

$$p \equiv \text{class } A : \left\{ \begin{array}{l}
C_i f_i; \ \forall i \\
m_j \text{ branches } T_k (B_k x_k)\{\text{body}_k\}; \ k \in K_j \end{array} \right\} : \text{body}$$

The thesis is proved by induction on the number $n$ of the branches of $\mathcal{M}[p]_{(C_p, S_p, \Gamma_p, \mu p)}(m_j)$ which coincides with the cardinality of the set $L(\mathcal{F}(A, m_j))$.

- The base case ($n = 0$) is trivial.

- Let us assume $\forall i \leq n \mathcal{M}[p]_{(C_p, S_p, \Gamma_p, \mu p)}(m_j) : \mathcal{T}[p](m_j)$; then we want to prove that

$$\mathcal{M}[p]_{(C_p, S_p, \Gamma_p, \mu p)}(m_j) : \mathcal{T}[p](m_j)$$

where $m_j$ has $n + 1$ branches. The result comes from $\lambda_{\text{object}}$ rule $[\bot] \text{INTRO}$, Lemma 7.4, the induction hypothesis and by using the definition of $\mathcal{M}[p]_{(C, S, \Gamma, \mu BT)}(m_j)$ and $\mathcal{T}[p](m_j)$.

- $(N > 1)$

$$p \equiv \text{class } A : A_h \left(\begin{array}{l}
C_i f_i; \ \forall i \\
m_j \text{ branches } T_k (B_k x_k)\{\text{body}_k\}; \ k \in K_j \end{array} \right\} : p'$$

We have to prove that, $\forall j \in J$:

$$\mathcal{M}[p]_{(C, S, \Gamma, \mu BT)}(m_j) =$$

$$\ldots (\mathcal{M}[p]_{(C, S, \Gamma, \mu BT)}(m_j) \& \mathcal{T}[p](m) \& \ldots \& (A \times \mathcal{B}_{\sigma(1)}) \vdash \mathcal{T}_{\sigma(1)})$$

$$\lambda (self^A, x^{\mathcal{B}_{\sigma(1)})} \cdot [BT(m_j, D_{\mathcal{B}_{\sigma(1)}}, B_{\mathcal{B}_{\sigma(1)})}] \Gamma[\text{this} \vdash A, x^{\mathcal{B}_{\sigma(1)} \vdash B_{\mathcal{B}_{\sigma(1)}}}]$$

$$:$$

$$\& \mathcal{T}[p](m) \& \ldots \& (A \times \mathcal{B}_{\sigma(n-1)} \vdash \mathcal{T}_{\sigma(n-1)}) \& (A \times \mathcal{B}_{\sigma(n)}) \vdash \mathcal{T}_{\sigma(n)})$$

$$\lambda (self^A, x^{\mathcal{B}_{\sigma(n)})} \cdot [BT(m_j, D_{\mathcal{B}_{\sigma(n)}}, B_{\mathcal{B}_{\sigma(n)})}] \Gamma[\text{this} \vdash A, x^{\mathcal{B}_{\sigma(n)} \vdash B_{\mathcal{B}_{\sigma(n)}}}]$$

is of type $\mathcal{T}[p](m_j)$, where
L(\mathcal{F}(A, m_j)) = \{1, \ldots, n\} and
\mathcal{F}(A, m_j) = \{(D_{B_1} \times B_1) \rightarrow T_1, \ldots, (D_{B_n} \times B_n) \rightarrow T_n\}
where each \(D_{B_i}\) is either \(A\) or one of its superclasses.

By induction hypothesis \(C; S \vdash M[p^!][C_p, S_p, Γ, BT_p](m_j) : T[p^!][m_j]\).
Then the proof proceeds as in the previous case.

**Theorem 7.1** (Type preservation under semantic translation). Let \(p\) be a well typed closed DOCS program. Then

\(C_p; S_p; Γ_p \vdash p : A\) implies \(C_p; S_p \vdash Ω(p) : A\)

**Proof.** Let \(p = \text{classdef}^{*} \text{body}\). Then:

\(\vdash p : A \equiv C_p; S_p; Γ_p \vdash p : A\) by definition

\(⇒ C_p; S_p; Γ_p \vdash \text{body} : A\) by iterate application of DOCS rule [\textsc{Program-class}].

\(⇒ C_p; S_p \vdash [\text{body}]_{(C_p, S_p, Γ_p, BT_p)} : A\) by Lemma 7.3.

\(⇒ C_p; S_p \vdash [\text{body}]_{(C_p, S_p, Γ_p, BT_p)}[m_j]_{Γ_p(m_j)} := M[p][C_p, S_p, Γ, BT_p](m_j) \in J] : A\) by Lemma 7.5 and by the substitution property of \(λ_{object}\), [15]

that is, \(C_p; S_p \vdash Ω(p) : A\) by Definition 6.10.

If \(Ω(p)\) is well typed, then the execution of \(Ω(p)\) cannot result in a run time type error, because \(λ_{object}\) is type safe. We recall that, in a well typed program \(p\), all local variables are initialized and, when creating an object, suitable expressions are provided to initialize all the fields of that objects. As a consequence, during the translation of \(p\) into \(λ_{object}\), any creation of a fresh identifier \(id^T\) will be associated to an assignment for \(id^T\). Thus, taking into account the semantics of the assignment operation, we can conclude that the computation starting from \(Ω(p)\) cannot produce an error configuration, and then the execution of \(p\) never gets stuck.

**Theorem 7.2** (DOCS is type safe). The execution of a well typed (closed) DOCS program \(p\) cannot result in a type error.

**Proof.** If \(p\) is well typed (\(\vdash p : T\)) then \(Ω(p)\) is well typed \((C_p; S_p \vdash Ω(p) : T)\) by Theorem 7.1. The execution of \(Ω(p)\), according to the operational semantics of \(λ_{object}\), cannot produce run time type errors by type safety of \(λ_{object}\).
8. Related works

Various languages have been proposed to study multi-methods with dynamic overloading. CLOS [9] 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 [38] 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. Fortress [2] is an object-oriented language supporting methods within traits [37] 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 [6] are a variant of the encapsulated multi-methods of [7,16] 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; 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. MultiJava [21] is an extension of Java that supports multi-methods by using open classes (classes to which new methods can be added without editing the class directly) and multi-methods.

Concerning core languages and calculi, we can compare DOCS to FJ [29]: DOCS models a set of features that are in between a minimal core language as FJ and a concrete language. Indeed, while FJ abstracts the functional core of Java, DOCS abstracts the imperative part by integrating in this core advanced features that language extension experiments can rely on. Note that, passing to a version of DOCS with less powerful features is straightforward.

KOOL [16] integrates multi-methods and overloaded functions (external to classes) in a functional kernel object-oriented language in a type safe way. KOOL type system and its semantics by translation into \( \lambda \) object were the starting point to build DOCS; indeed the notion of well formedness for DOCS types is widely inspired by the one defined in [16]. Both KOOL and DOCS aim at being light and compact in presenting kernel object-oriented features, however, they are quite different since they have two different goals. KOOL includes both encapsulated multi-methods and overloaded functions external to classes (not included in DOCS) thus unifying in a single language two different styles of programming. As it is explained in [16] (Sect. 3.5) KOOL aims at defining a formalism in which to study the features of object-oriented programming so the choice in the design was “to keep the language small simple and easily comprehensible but keep it from being a realistic
"programming language". For this reason KOOL omits imperative features in its presentation (although Chap. 9 of [16] introduced imperative features that we extended). On the other hand DOCS is designed to study the interaction of complex mechanisms in a context as close as possible to real languages so we include imperative features; we also allow forward declarations in methods class definitions, namely, we let the programmer write a method with a parameter of type A in a class no matter if the class A is defined before or after the current class. This feature (included in mainstream languages such as Java and C++) is supported by KOOL, influenced the typing of DOCS: first the declarative part is checked producing a class table which is used to type check method declarations and definitions. Finally, KOOL does not interpret overloading by copy semantics, so many multi-method definitions which are well typed in DOCS are discarded in KOOL. Thus DOCS allows a less restrictive overloading resolution policy still maintaining the primary goal of type safety.

[19] presents a polynomial-time static type checking algorithm that checks the conformance, completeness, and consistency of a group of method implementations with respect to declared message signatures. This result can be used also in our approach to address the efficiency issue of the typing procedure.

Tuple [32] is a simple object-oriented language designed to integrate multiple dispatch in single dispatching languages by adding tuples as primitive expressions. Tuple is statically typed and it is proved to be type safe. In Tuple methods can be defined either inside classes or in tuple classes (external to standard classes). Message look-up 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 DOCS. First, Tuple supports dynamic overloading only for methods external to classes thus adopting a mechanism which is closer to multiple dispatching languages. As a consequence in Tuple the distinction between static and dynamic overloading is made at method definition while in DOCS it is deferred at the time of method calls thus increasing flexibility.

Dubious [33] 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. Dubious and DOCS share some basic goals such as the symmetry of dispatch, a clear policy for ambiguities resolution (avoiding linearization of parameters or of superclasses) and static type safety. On the other hand the two languages are totally different since DOCS is based on classes and does not address the issue of separate type checking.
9. Conclusions

In this paper we presented DOCS (Dynamic Overloading with Copy Semantics), an object-oriented imperative kernel language, including basic features of many mainstream programming languages, combined with advanced features: multiple inheritance, dynamic overloading and copy semantics. Indeed, DOCS gives a type-based foundation to dynamic overloading with copy semantics. We defined a translational semantics of DOCS into the meta-language $\lambda_{\text{object}}$ [15,16].

Therefore, DOCS can be thought of as a formal framework to design and develop language extensions and to prove theoretical properties of such extensions. For instance, we exploited this framework to study and implement an extension of C++ with dynamic overloading in [8], where dynamic overloading is implemented by using only dynamic binding and static overloading. This extended C++ has been implemented and the correctness of this implementation has been proven using the semantics of DOCS (the implementation is freely available at http://doublecpp.sf.net/).

Acknowledgements. We thank Franco Barbanera and Mariangiola Dezani for many helpful remarks and suggestions. We also thank the anonymous referees for useful suggestions on how to improve the paper.

References


[25] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts and A.P. Black, Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems 28 (2006) 331–388.


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


Communicated by G. Longo.
Received November 9, 2007. Accepted January 5, 2009.