# A meta-language for typed object-oriented languages

Giuseppe Castagna*

LIENS(CNRS)-DMI
45 rue d'Ulm, 75005 Paris. FRANCE
e-mail: castagna@dmi.ens.fr

**Abstract.** In [3] we defined the $\lambda\&$-calculus, a simple extension of the typed $\lambda$-calculus to model typed object-oriented languages. To develop a formal study of type systems for object-oriented languages we define, in this paper, a meta-language based on $\lambda\&$ and we show by a practical example how to use it to prove properties of a language. To this purpose we define a toy object-oriented language and its type-checking algorithm; then we translate this toy language into our meta-language. The translation gives the semantics of the toy language and a theorem on the translation of well-typed programs proves the correction of the type-checker of the toy language.
As an aside we also illustrate the expressivity of the $\lambda\&$-based model by showing how to translate existing features like multiple inheritance and multiple dispatch, but also by integrating in the toy language new features directly suggested by the model, such as first-class messages, a generalization of the use of super and the use of explicit coercions.

## 1 Introduction

In [3] we introduced the $\lambda\&$-calculus. It is a simple extension of the typed lambda calculus to deal with overloaded functions, subtyping and dynamic binding. The main motivation of its definition was to give a kernel calculus possessing the key properties of object-oriented programming, in the line of some ideas of [6]. In the same paper we showed how this calculus could be intuitively used to model some features of object-oriented programming. It resulted that such a calculus yields a model orthogonal to the ones proposed in the literature so far. Thus we returned to object-oriented programming and we reviewed it in the light of the model arising from the $\lambda\&$-calculus. The experiment was surprising since we were able to deal with some features (such as multiple dispatch or the extension of the set of methods of a certain class) and introduce new ones (as first class messages or a generalization of "`super`") the usual models could not.

However, $\lambda\&$ is inadequate for a formal study of the properties of real object-oriented languages, and it was not meant for this: it is a calculus not a meta-language; thus, even if it possesses the key mechanisms to model object-oriented features, it cannot be used to "reason about" (i.e. to prove properties of) an object-oriented language.

For this reason in this paper we define a meta-language (i.e. a language to reason about —object-oriented— languages)[2] that we call $\lambda$_object. This

[1] In *13th Conference on Foundations of Software Technology and Theoretical Computer Science*. Bombay, December 1993. LNCS to appear

* Supported by grant no. 203.01.56 of the Consiglio Nazionale delle Ricerche, Comitato Nazionale delle Scienze Matematiche, Italy, to work at LIENS

[2] In this case the prefix "meta" is used w.r.t. the object-oriented languages

language is still based on the key mechanisms of $\lambda\&$ (essentially, overloading and dynamic binding) but it is enriched by those features (like commands to define new types, to work on their representations, to handle the subtyping hierarchy, to change the type of a term or to modify the discipline of dispatching etc.) that are necessary to reproduce the constructs of a programming language and that $\lambda\&$ lacks for.

We also show, by a practical example, how to use $\lambda$_object to prove properties of an object-oriented language. To this purpose we define a simple toy object-oriented language (a mix of Objective-C and CLOS constructs) and an algorithm to type-check its programs. We then translate the programs of the toy object-oriented language into $\lambda$_object. We prove that every well typed program of the former is translated into a well typed program of the latter; since the latter enjoys the subject-reduction property, it implies that the reduction of the translated program never goes wrong on a type error; in particular this proves the correction of the type-checker for the toy language.

The paper is organized as follows: section 2 gives an informal description of the toy language and of its type discipline. In section 3 we briefly summarize the $\lambda\&$-calculus. In section 4 we describe $\lambda$_object: we give its operational semantics, a type-checker and prove the subject reduction theorem. In section 5 we hint the translation and we prove the correction of the type discipline for the toy language. For space reasons we cannot give a detailed description of all the systems. All the details and the precise connection between $\lambda\&$ and $\lambda$_object will be included in the author's PhD. thesis.

## 2 The toy language

### 2.1 Message passing

There exist many syntaxes for messages; in our toy language *message-expressions* are enclosed in square brackets: [*receiver message*]. There are two ways to model message passing. One is to consider an object as a record of methods and message passing as dot selection (e.g. in Eiffel; see [8]). The other is to consider message passing as functional application where the message is the function and the receiver is the argument (as in CLOS; see [7]). In this paper we choose this second solution. Though the fact that a method belongs to a specific object (more precisely to a specific *class* of objects) implies that *message passing* is a mechanism different from the usual functional call (i.e. $\beta$-reduction). In our approach the main characteristics that distinguish messages from functions are:

*Overloading:* Two objects may respond differently to the same message. For instance, the code executed when sending a message `inverse` to an object representing a matrix will be different from the one executed when the same message is sent to an object representing a real number. But the same message behaves uniformly on objects of the same kind (e.g. on all objects of *class* `matrix`). This feature is known as *overloading* since we overload the same operator (in this case `inverse`) by different operations; the actual operation depends on the type of the operands. Thus *messages are identifiers of overloaded functions* and in message passing the *receiver* is the first argument of an overloaded function, i.e. the one on whose type is based the selection of the code to be executed. Each method constitutes a *branch* (i.e. a code or operation) of the overloaded function referred by the message it is associated to.

*Dynamic binding:* The second crucial difference between function application and message passing is that a function is bound to its meaning at compile time

while the meaning of a method can be decided only at run-time when the receiving object is known (fully evaluated). This feature is called *dynamic binding*.

Therefore in our model overloading and dynamic binding are the basic mechanisms. [3]

### 2.2 Classes and programs

The name of a class is used as the type of its objects and constitutes an "atomic type" of our type system. We restrict our attention to a functional case of OOP; thus the instance variables of an object are modified by an operation `update` which returns a new object of the same type of the current object. We show the syntax of class definition in our toy language by an example:

```
class 2DPoint
  {
    x:Int = 0;
    y:Int = 0
  }
    norm = sqrt(self.x^2 + self.y^2);
    erase = (update{x = 0});
    move = fn(dx:Int,dy:Int) => (update{x=self.x+dx; y=self.y+dy})
  [[
    norm: Real,
    erase: 2DPoint,
    move: (Int x Int) -> 2DPoint
  ]]
```

Instances of a class are created by means of the command `new`. Since the name of a class is used for the type of its instances then `new(2DPoint):2DPoint`.

A *program* in our toy language is a sequence of declarations of classes followed by an expression (the *body* of the program) where objects of these classes are created and interact by exchanging messages.

### 2.3 Refinement

It is possible to define new classes by *refining* existing ones. The refinement induces on the atomic types two different hierarchies generated by two distinct mechanisms: inheritance, which is the mechanism that allows to reuse code written for other classes and which concerns the definition of the objects; subtyping, which is the mechanism that allows to use one object instead of another of a different class and which concerns the computation of the objects. It is well-known that these hierarchies are distinct (see [5]). In our toy language we take a simpler approach, including in it only subtyping. Thus it is not possible to have "pure" inheritance (i.e. code reuse without the substitutivety given by subtyping). We use the keyword `is` in the class definition to define the *subtype* relation among classes. A typical example of its use is:

[3] The use of dynamic binding automatically introduces a further distinction between ordinary functional application and message passing: while the former can be dealt with by either call-by-value or call-by-name, the latter can be performed only when the run-time type of the argument is known, i.e. when the argument is fully evaluated (closed and in normal form). In view of our analogy "messages as overloaded functions" this (nearly) corresponds to say that message passing (i.e. overloaded application) acts by call-by-value: see proposition 4.2 and corollary 5.

```
class 2DColorPoint is 2DPoint
  { x:Int = 0 ; y:Int = 0 ; c:String = "black"}
   isWhite = (self.c == "white")
   move = fn(dx:Int,dy:Int)=>(update{x=self.x+dx; y=self.y+dy; c="white"})
  [[ isWhite: Bool , move: (Int x Int) -> 2DColorPoint]]
```

The keyword `is` says that `2DColorPoint` is a subtype of `2DPoint` (denoted by `2DColorPoint` $\leq$ `2DPoint`). It is possible to specify more then one superclass after `is`, by separating the ancestors by commas (multiple inheritance).

To substitute values of some type by those of another type some requirements must be satisfied. If the type at issue is a class then the following conditions must hold:

1. *state coherence*: The set of the instance variables of a class must contain those of all its superclasses. Moreover common variables must appear with the same type.
2. *covariance*: A method that overrides another method must specialize it, in the sense that the type returned by the new method must be a subtype of the type returned by the old method.
3. *multiple inheritance*: When a class is defined by multiple refinement, the methods that are in common to more than one unrelated supertype must be explicitly redefined

We have chosen not to use a class precedence list (as in CLOS) but rather the explicit redefinition of common methods (as in Eiffel) which is less syntax dependent and thus mathematically cleaner.

### 2.4 Extending classes

Refinement is not the only way to specialize classes. It is also possible to add new methods to existing classes or to redefine the old ones (see for example Objective-C or `add-method` in CLOS). In our toy language this can be done by the following expression:

`extend` *classname*
*methodDefinitions*
*interface*
`in` *exp*

the newly defined methods are available inside the expression *exp*. The extension of a class affects all its subtypes, in the sense that when you extend a class with a method then that method is available to the objects of every subtype of that class.

### 2.5 Super, self and the use of coercions

The use of the reserved keyword `self` is well-known: it denotes in a method the receiver of the message that invoked the method. Though, in view of our analogy of messages as identifiers of overloaded functions, `self` assumes also another meaning. Indeed recall that the receiver of a message is the argument of the overloaded function denoted by that message. Thus in the definition of a method, `self` is the formal parameter of the overloaded function in which that method appears as a branch.

Also the use of `super` is well-known: when we send a message to `super`, the effect is the same as sending it to `self` but with the difference that the *selection is performed as if the receiver were* an instance of a super-class. Here we generalize this usual meaning of super in two ways: the selection does not

assume that the receiver is `self`, but takes as receiver the parameter of `super`; and `super` does not necessary appears in the receiver position, but it is a first-class value (i.e. it can appear in any context its type allows to). Finally, since we use multiple inheritance without class precedence lists, we are obliged to specify in the expression the supertype from which to start the search of the method[4]. Thus the general syntax of super is `super[`$A$`](`*exp*`)`. When a message is sent to this expression then *exp* is considered the receiver but the search of the method is started from the class $A$ (which then must be a supertype of the class of *exp*).

Very close to the use of `super` is the use of the coercions. By a coercion we change the class of an object to a supertype. The difference between them is that `super` changes the class of an object only in the first message passing, while a coercion changes it for the whole life of the object. The syntax is the same as that of `super`: thus we write `coerce[`$A$`](`$M$`)` to change to $A$ the type of the object $M$. In conclusion, `coerce` changes the class of its argument and `super` changes the rule of selection of the method in message passing (it is a coercion that is used only once and then disappears).[5]

### 2.6 Multiple dispatch

In this toy language it is possible to base the choice of the methods not only on the class of the receiver of a message but also on the class of possible parameters of the message. This feature is called *multiple dispatch* and the method at issue is usually referred as a *multi-method* (see e.g. [7]). An example of multi-method in our toy language is:

```
extend 2DPoint
  compare = & fn(p:2DPoint) => ([self norm] == [p norm])
            & fn(p:2DColorPoint) => [p isWhite];
  [[ compare:#{2DPoint -> Bool; 2DColorPoint ->Bool} ]]
in ...
```

If the parameter of `compare` is a `2DPoint` then the first line is executed; the second one if it is (a subtype of) a `2DColorPoint`. Note that the type of a multi-method appears in the interface as the set of the types of the possible choices (the reason why we prefixed the type by `#` is explained in the next session).

The number of parameter on which the dispatch is performed may be different in every branch. For this reason, when a message denoting a multi-method is sent, we must single out those parameters the dispatching is performed on. This is done by including them *inside* the brackets of the message-passing, after the message. Thus the general syntax of message passing is: `[`*receiver message parameter*`, ... ,` *parameter*`]`. For example, consider a class $C$ with the following interface: `[[ msg:#{Int -> (Int -> Bool), Int x Int -> Bool}]]`; if $M$ is of class $C$ then the expression `[` $M$ `msg 3 ] 4` selects the first branch while `[` $M$ `msg 3,4]` selects the second one. We have to impose a restriction in our system: `super` cannot work with multiple dispatching; when super selects a multi-method, it works as `coerce`

### 2.7 Messages as first class values: adding overloading

Messages are identifiers of overloaded functions. But up to now overloaded functions can be defined only through class definitions. Thus the next step is to

[4] This is what is done in Fibronacci, developed at the University of Pisa

[5] It is interesting that with our generalization of `super` it is possible to predetermine the life of a coercion: for example `super[`$A$`](super[`$A$`](`$M$`))` coerces $M$ to $A$ only for the first two message passing.

introduce explicit definitions for overloaded functions and to render them (and thus messages) first class values. The gain is evident: for example we can have functions accepting or calculating messages (indeed overloaded functions) and to write message passing of the form `[`*receiver* $f(x)$`]` (see [2] for an example).

We use the syntax of message passing for overloaded application; thus in `[`$exp_0$ *exp* $exp_1, \ldots,$ $exp_n$`]` we have that *exp* is the overloaded function and $exp_0, exp_1, \ldots, exp_n$ are the arguments. We use the syntax of multi-methods to define overloaded functions. Therefore we build an overloaded function by concatenating the various branches by `&`; the argument of each branch must have an atomic type. The type of an overloaded function is the set of the types of its branches. For example an overloaded "plus" working both on integers and reals can be defined in the following way:

```
let plus = (& (fn(x:Real,y:Real) => x real_plus y)
              & (fn(x:Int,y:Int) => x int_plus y))
```

which has type $\{Real \times Real \to Real, Int \times Int \to Int\}$. Thus the sum of two numbers, $x$ and $y$, using `plus` is written `[ x plus y ]`.

Finally note that the use `of #` in the interfaces is necessary to distinguish multi-methods from ordinary methods returning an overloaded function. [6]

### 2.8 Type checking of the toy language

In this section we describe the type system of our toy language. We define here only the rules for the object-oriented part of the language, since the typing of the functional part is quite standard.

**Types**

The types that can be used in a program of our toy-language are: Class-names which are user-defined atomic types. Product types $(T \times T')$, for pairs. Arrow types $T \to T'$, for ordinary functions. Sets of arrow types $\{A_1 \to T_1, \ldots, A_n \to T_n\}$ called *overloaded types* and used for overloaded functions where we call $A_1 \ldots A_n$ and $T_1 \ldots T_n$ *input* and *output* types respectively. In an overloaded type there cannot be two different arrow types with the same input type (*input type uniqueness*).

$$R ::= \langle\!\langle \ell_1 : T_1; \ldots; \ell_n : T_n \rangle\!\rangle \qquad \text{(record types)}$$

$$T ::= A \mid T \to T \mid (T \times \ldots \times T) \mid \{(A_1 \mathtt{x} \ldots \mathtt{x} A_{m_1}) \to T_1, \ldots, (A'_1 \times \ldots \times A'_{m_n}) \to T_n\} \quad (m_i \geq 1) \qquad \text{(raw types)}$$

$$V ::= T \mid \#\{(A_1 \times \ldots \times A_{m_1}) \to T_1, \ldots, (A'_1 \times \ldots \times A'_{m_n}) \to T_n\} \qquad \text{(interface types)}$$

In the following we use the meta-variables $T, U$ and $W$ to range over raw types. If $T$ denotes the type $\{U_i \to T_i\}_{i=1..n-1}$ then the notation $T \cup \{U_n \to T_n\}$ denotes the type $\{U_i \to T_i\}_{i=1..n}$ if $U_n \to T_n$ is different from all the arrow types in $T$, and it denotes $T$ itself otherwise. In other terms $\cup$ denotes the usual set-theoretic union.

[6] Note that the use of the syntax of message passing also for overloaded application, while providing a conceptual uniformity, has a major drawback: when the overloaded function has more than one argument then the arguments have to be "split" around the overloaded function. In case of binary infix overloaded operators, like the case of `plus`, this turns out to be very readable. But, apart from these special cases, it remains a problem and it may suggest us to consider a different syntax for message passing where the message is the left argument, as done in CLOS (see [7]).

**Rules for Subtyping**
The subtyping relation is predefined by the system on the built-in atomic types; the programmer defines it on the atomic types (i.e. the classes) he introduces, by means of the construct `is`. This relation is automatically extended to arrow types and product types by the usual rules (pairwise ordering for products and contravariance in the left argument for the arrow constructor). To define the subtyping relation on overloaded types, note that an overloaded function can substitute another overloaded function iff for every branch of the latter there is at least one in the former that can substitute it. Thus an overloaded type is smaller than another if for every arrow type in the latter there is at least one smaller arrow type in the former. Formally the subtyping relation on the atomic types is stored in a type constraint system:

**Definition 1.** $\emptyset$ is a type-constraint system. If $C$ is a type-constraint system and $A_1, A_2$ are atomic types then $C \cup (A_1 \leq A_2)$ is a type-constraint system.

And the subtyping rule for overloaded types is:

$$\frac{\text{for all } i \in I, \text{ there exists } j \in J \text{ such that } C \vdash D_i'' \leq D_j' \text{ and } C \vdash U_j' \leq U_i''}{C \vdash \{D_j' \to U_j'\}_{j \in J} \leq \{D_i'' \to U_i''\}_{i \in I}}$$

Using this subtyping relation we select among the raw types those which satisfy the conditions of the good formation of class in section 2.3. In particular the last two conditions, reformulated in terms of overloading, become[7]:

1. *covariance*: In an overloaded type, if an input type is a subtype of another input type then their corresponding output types must be in the same relation
2. *multiple inheritance*: In an overloaded type if two unrelated input types have a common subtype then for every maximal type of the set of their common subtypes there must be one branch whose input type is that maximal type.

**Rules for Terms**
1. The type of an object is the name of its class.
2. The type of a coercion and of a super is the class specified in it, provided that it is a supertype of the type of the argument.
3. The type of `self` is the name of the class whose definition `self` appears in.
4. The type of an overloaded function is the set of the types of its branches
5. The type of an overloaded application is the output type of the branch whose input type "best approximates" the type of the argument. This branch is selected among all the branches whose input type is a supertype of the type of the argument and it is the one with the least input type.

These are all the typing rules we need to type the object-oriented part of the toy language, since we said that messages are nothing but overloaded functions and message passing reduces to overloading application. However to fully understand message passing we must specify which overloaded function a message denotes. Suppose that you are defining a class $C$ and remember that inside the body of a method, the receiver is denoted by `self`. Then there are two cases:

[7] The formal definition of the well formed types is the same as the one for $\lambda$_object in appendix A.1 but without the "$S$" indexes

1. The method `msg=`*exp* is not a multi-method and returns (according to the interface) the type $T$. This corresponds to add to the overloaded function denoted by `msg` the branch `fn(self:`$C$`).`*exp* whose type is $C \to T$.
2. We have the multi-method
`msg = & fn(`$\mathtt{x}_1$`:`$A_1$`,...,`$\mathtt{x}_i$`:`$A_i$`) =>` $exp_1$`...& fn(`$\mathtt{y}_1$`:`$B_1$`,...,`$\mathtt{y}_j$`:`$B_j$`) =>` $exp_n$ which returns the type `#{`$(A_1 \times \ldots \times A_i) \to T_1, \ldots, (B_1 \times \ldots \times B_j) \to T_n$`}`. This corresponds to add to the overloaded function denoted by `msg` the $n$ branches `fn(self:`$C$`, `$\mathtt{x}_1$`:`$A_1$`,...,`$\mathtt{x}_i$`:`$A_i$`) =>` $expr_1$ ... `fn(self:`$C$`, `$\mathtt{y}_1$`:`$B_1$`, ..., `$\mathtt{y}_j$`:`$B_i$`) =>` $expr_n$ of types $(C \times A_1 \times \ldots \times A_i) \to T_1, \ldots, (C \times B_1 \times \ldots \times B_j) \to T_n$

The selection of the branch corresponds to the search of the least supertype of the class of the receiver (a class is a supertype of itself) in which a method has been defined for the message (this is the usual method look-up).

Formally, we define the relation $C; S; \Gamma \vdash p{:}T$, where $C$ is a type-constraint system, $p$ a program, $T$ a well-formed type and $\Gamma$ and $S$ are partial functions between the following sets: $\Gamma{:}\,(Vars \cup \{\mathtt{self}\}) \to \mathbf{Types}$ and $S{:}\,\mathbf{AtomicTypes} \to \mathbf{RecordTypes}$. $\Gamma$ records the types of the various identifiers. The function $S$ records the type of the internal states of the previously defined classes. In particular $\Gamma(\mathtt{self})$ is the *current class* and the domain of $S$ (i.e. the values for which $S$ is defined) is the set containing the names of all the classes that have been defined up to that point. We give here just the most significant type-checking rules followed by a short comment:

[New] $$C; S; \Gamma \vdash \mathtt{new}(A){:}\,A \qquad A \in dom(S)$$
The type of a new object is the name of its class. $A \in dom(S)$ checks that the class has been previously defined.

[Read] $$C; S; \Gamma \vdash \mathtt{self}.\ell{:}\,T \qquad S(\Gamma(\mathtt{self})) = \langle\!\langle \ldots \ell{:}\,T \ldots \rangle\!\rangle$$
The expression $\mathtt{self}.\ell$ reads the value of an instance variable of an object. Thus it must be contained inside the body of a method; then $S(\Gamma(\mathtt{self}))$ is the type of the internal state of the current class.

[Write] $$\frac{C; S; \Gamma \vdash r{:}\,R}{C; S; \Gamma \vdash \mathtt{(update}\ r\mathtt{)} : \Gamma(\mathtt{self})} \qquad C \vdash S(\Gamma(\mathtt{self})) \leq R$$
Also this expression must be contained in a method. $S(\Gamma(\mathtt{self})) \leq R$ check that the fields specified in it are instances variables of the current class[8].

[OvAbst] $$\frac{C; S; \Gamma \vdash exp_1{:}\,T_1 \ldots C; S; \Gamma \vdash exp_n{:}\,T_n}{C; S; \Gamma \vdash\ \mathtt{\&}exp_1\mathtt{\&} \ldots \mathtt{\&}\ exp_n : \{T_1, \ldots, T_n\}} \quad \{T_1, \ldots, T_n\} \in_C \mathbf{Types}$$
The type of an overloaded function is the set of the types of its branches.

[OvAppl] $$\frac{C; S; \Gamma \vdash exp{:}\,\{D_i \to T_i\}_{i \in I} \qquad C; S; \Gamma \vdash exp_j{:}\,A_j \quad (j=0..n)}{C; S; \Gamma \vdash \mathtt{[}exp_0\ exp\ \ exp_1, \ldots, exp_n\ \mathtt{]} : T_h}$$
$$D_h = \min_{i \in I}\{D_i \mid C \vdash A_0 \times A_1 \times \ldots \times A_n \leq D_i\}$$
When we pass a message or, more generally, we perform an overloaded application we look at the type of the function, *exp*, and we select the branch whose input type best approximates the type of the argument. The argument is $(exp_0, exp_1, \ldots, exp_n)$ and the selected branch is the branch $h$ such that $D_h = \min_{i \in I}\{D_i \mid C \vdash A_0 \times A_1 \times \ldots \times A_n \leq D_i\}$.

[Coerce] $$\frac{C; S; \Gamma \vdash exp{:}\,A}{C; S; \Gamma \vdash \mathtt{coerce[}A'\mathtt{](}exp\mathtt{)}{:}\,A'} \qquad C \vdash A \leq A'$$
The construct `coerce[`$A'$`](`*exp*`)` says to consider *exp* (whose type is $A$) as if it were of type $A'$. This is a type safe operation if and only if $A \leq A'$. A similar rule can be used for `super`, too.

[8] We consider only *field extension* for record subtyping.

Finally let us consider the typing of a class definition. We have to open a short parenthesis. A class definition is always of the form:

$\texttt{class}\ A\ \texttt{is}\ A_1,\ldots,A_n\ \ r\colon R\ \ \mathtt{m}_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}\mathtt{m}_m\texttt{=}exp_m\ \ \texttt{[[}\ \mathtt{m}_1\colon V_1,\ldots,\mathtt{m}_m\colon V_m\ \texttt{]]}\ \texttt{in}\ \mathrm{p}$

where we use the notation $r\colon R$ to denote that the instance variables have type $R$ and initial values given by $r$. The whole program is well-typed if the class definition is well-typed and the program $p$ is well-typed under an environment including the new definitions introduced by this class. To obtain this environment we have to update the type of the messages by adding the types of the the new branches defined in the class. We have to distinguish the case of a simple method from that of a multi-method. For every message $\mathtt{m}_i$ in the interface such that $V_i$ is a raw type we must update its current type $\Gamma(\mathtt{m}_i)$ in the following way: $\Gamma(\mathtt{m}_i) := \Gamma(\mathtt{m}_i) \cup \{A \to V_i\}$ (where we use the convention that $\Gamma(\mathtt{m}_i) = \{\}$ if $\mathtt{m}_i \notin dom(\Gamma)$ ). If the type of a message in the interface is preceded by a `#`, then the associated method is a multi-method; recall that the type of its argument is the cartesian product of the type of the current class with the types the dispatch is performed on (see the rule [OvAppl]). Thus for example if in the interface $\mathtt{m}_i\texttt{:\#}\{D \to U, D' \to T\}$ then we have the following updating: $\Gamma(\mathtt{m}_i) := \Gamma(\mathtt{m}_i) \cup \{(A \times D) \to U, (A \times D') \to T\}$. More generally we define

$$A \rightsquigarrow V = \begin{cases} \{(A \times D_i) \to U_i\}_{i \in I} & \text{if } V \equiv \texttt{\#}\{D_i \to U_i\}_{i \in I} \\ \{A \to V\} & \text{otherwise} \end{cases}$$

thus the updating of $\Gamma$ gets: $\Gamma(\mathtt{m}_i) := \Gamma(\mathtt{m}_i) \cup \{A \rightsquigarrow V_i\}$, (where the same convention as before applies).
We are now able to write the rule [Class]. In order to shorten it we use the following abbreviations:

- $S' \equiv S[A \leftarrow R]$ the function $S$ where to the class $A$ is associated the type of its internal state $R$.
- $C' \equiv C \cup (\bigcup_{i=1..n} A \le A_i)$ the set $C$ extended by the type constraints generated by the definition
- $I \equiv\ \texttt{[[}\ \mathtt{m}_1 : V_1,\ldots,\mathtt{m}_m : V_m\ \texttt{]]}$ the interface of the class
- $\Gamma' \equiv \Gamma[\mathtt{m}_i \leftarrow \Gamma(\mathtt{m}_i) \cup \{A \rightsquigarrow V_i\}]_{i=1..m}$ the environment $\Gamma$ where the (overloaded) type of the messages is updated with the type of the new methods (branches) added by the class-definition

$$\frac{C; S; \Gamma \vdash r\colon R \qquad C'; S'; \Gamma'[\texttt{self} \leftarrow A] \vdash exp_j : V_j \ \ (j=1..m) \qquad C'; S'; \Gamma' \vdash p : T}{C; S; \Gamma \vdash \texttt{class}\ A\ \texttt{is}\ A_1,\ldots,A_n\ r\colon R\ \ \mathtt{m}_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}\mathtt{m}_m\texttt{=}exp_m\ \ I\ \ \texttt{in}\ p : T}$$

$$A \notin dom(S),\ C \vdash R \le S(A_j)\ \ \Gamma(\mathtt{m}_i) \cup \{A \rightsquigarrow V_i\} \in_{C'} \textbf{Types}\ (j = 1..n, i = 1..m)$$

Let us examine the single parts of this rule more in detail: first we control that a class with this name does not already exist ($A \notin dom(S)$), we check the type of the initial values of the instance variables ($C; S; \Gamma \vdash r\colon R$) and we verify that the type of the internal state of the class is compatible (i.e. it is an extension) with the states of its ancestors ($C \vdash R \le S(A_i)$ for $i = 1..n$). Then we check that the defined messages possess well-formed overloaded types ($\Gamma(\mathtt{m}_i) \cup \{A \rightsquigarrow V_i\} \in_{C'}$ **Types**), i.e. that they satisfy the conditions of covariance, multiple inheritance and input type uniqueness; we also check that the methods have the type declared in the interface ($exp_j : V_j$), and this check is performed in an environment where we have recorded in $C'$ the newly introduced type-constraints, in $S'$ the type of the internal state of the current class and in $\Gamma'$ the types of the new methods (since they can be mutually recursive). Finally we type the rest of the program; in order to implement the protection mechanisms we restore in the environment the old value for `self`.

The rule [EXTEND] can be seen as a special case of the rule [CLASS] where there are no type constraint and no instance variable to check; we just have to verify that the class in the extend expression has already been defined (i.e. $A \in dom(S)$ ):

$$\frac{C;S;\Gamma'[\texttt{self} \leftarrow A] \vdash exp_j : V_j \quad (j=1..m) \qquad C;S;\Gamma' \vdash exp : T}{C;S;\Gamma \vdash \texttt{extend A m}_1\texttt{=}exp_1\texttt{;...;m}_m\texttt{=}exp_m \ \texttt{[[m}_1 : V_1\texttt{,...,m}_m : V_m\texttt{]]} \ \texttt{in} \ exp : T}$$

$A \in dom(S)$ and for $i = 1..m$ $\Gamma(\texttt{m}_i) \cup \{A \rightsquigarrow V_i\} \in_C$ **Types**

## 3 The $\lambda$&-calculus

In this section we briefly recall the main definitions of the $\lambda$&-calculus, which has been defined in [3] to model overloading and dynamic binding. For a detailed discussion of its characteristics the reader may refer to the paper above and to [4].

An overloaded function is formed by a set of ordinary functions (i.e. lambda-abstractions), each one constituting a different branch. Overloaded functions are built as it is customary with lists, starting by an *empty* overloaded function denoted by $\varepsilon$, and concatenating new branches by means of &; therefore an overloaded function with $n$ branches $M_i$ is written as $(...((\varepsilon\&M_1)\&M_2)...)\&M_n)$. The type of an overloaded function is the set of the types of its branches. Thus if $M_i : U_i \rightarrow T_i$ then the overloaded function above has type $\{U_1 \rightarrow T_1, U_2 \rightarrow T_2, \ldots, U_n \rightarrow T_n\}$. The application of an overloaded function (i.e. the message passing) is denoted by "$\bullet$". If we apply the function above to an argument $N$ of type $U$ then we select the branch whose $U_i$ "best approximates" the type of the argument; i.e. we select the branch $j$ s.t. $U_j = min\{U_i | U \leq U_i\}$. And thus

$$(\varepsilon\&M_1\& \ldots \&M_n)\bullet N \ \triangleright^* \ M_j \cdot N \qquad (*)$$

where $\triangleright^*$ means "reduces in zero or more steps to".

Also, a set of arrow types is an overloaded type iff it satisfies these two conditions:

$$\text{if } \ U_i \leq U_j \ \text{ then } \ T_i \leq T_j \qquad (1)$$

$$\text{if } \ U_i \Downarrow U_j \ \text{ then there exists a unique } h \in I \text{ such that } U_h = \inf\{U_i, U_j\} \qquad (2)$$

where $U_i \Downarrow U_j$ means that $U_i$ and $U_j$ are downward compatible (have a common lower bound).

These are (a stronger version of) the conditions in section 2.8; i.e. we select those pretypes that satisfy the conditions of covariance, multiple inheritance and input type uniqueness.

This models overloading: it remains to include *dynamic-binding*. This can simply be done by requiring that a reduction as (*) can be performed only if $N$ is a closed normal form.

The formal description of the calculus is given by the following definitions:

**PreTypes** $\qquad T ::= A \mid T \rightarrow T \mid \{T'_1 \rightarrow T''_1, \ldots, T'_n \rightarrow T''_n\}$

**Subtyping**
We define a partial order on the pretypes. We start by a partial lattice[9] of

[9] A partial lattice is a (partially) ordered set such that for every pair of elements $a$ and $b$ if $a \Downarrow b$ then $\exists \inf\{a, b\}$ and if $a \Uparrow b$ then $\exists \sup\{a, b\}$

atomic types and we extend this order to higher pretypes in the following way:

$$\frac{U_2 \leq U_1 \quad T_1 \leq T_2}{U_1 \to T_1 \leq U_2 \to T_2} \qquad \frac{\forall i \in I, \exists j \in J \quad U_i'' \leq U_j' \text{ and } T_j' \leq T_i''}{\{U_j' \to T_j'\}_{j \in J} \leq \{U_i'' \to T_i''\}_{i \in I}}$$

The subtyping relation on **Pretypes** is given by the reflexive closure of the rules above (in [3] it is proved that transitivity is not necessary).

**Types**
A pretype is also a type if all the overloaded types that occur in it satisfy the conditions (1) and (2). We denote by **Types** the set of types. Types are equal modulo the ordering of the arrows in the overloaded types.

**Terms** $\qquad M ::= x^T \mid \lambda x^T M \mid MM \mid \varepsilon \mid M \&^T M \mid M \bullet M$
The type indexing the & is used for the selection of the branch in overloaded application.

**Type-checking Rules**
The type checking rules are very close to those for the toy object-oriented language. Indeed they are more general since any type can appear as input type of and overloaded function. We do not need any type context $\Gamma$ since the variables are indexed by their type.

$$[\text{Taut}] \quad x^T : T \qquad [\text{Taut}_\varepsilon] \quad \varepsilon : \{\}$$

$$[\to\text{Intro}] \quad \frac{M : T}{\lambda x^U . M : U \to T} \qquad [\{\}\text{Intro}] \quad \frac{M : W_1 \leq \{U_i \to T_i\}_{i \leq (n-1)} \quad N : W_2 \leq U_n \to T_n}{(M \&^{\{U_i \to T_i\}_{i \leq n}} N) : \{U_i \to T_i\}_{i \leq n}}$$

$$[\to\text{Elim}_\leq] \quad \frac{M : U \to T \quad N : W \leq U}{MN : T} \qquad [\{\}\text{Elim}] \quad \frac{M : \{U_i \to T_i\}_{i \in I} \quad N : U \quad U_j = \min_{i \in I}\{U_i | U \leq U_i\}}{M \bullet N : T_j}$$

**Reduction**
The reduction $\rhd$ is the *compatible closure* of the following *notion of reduction* (for definitions see [1]):

$\beta$) $(\lambda x^T . M) N \rhd M[x^T := N]$
$\beta_\&$) If $N : U$ is closed and in normal form, and $U_j = min_{i=1..n}\{U_i | U \leq U_i\}$ then

$$(M_1 \&^{\{U_i \to T_i\}_{i=1..n}} M_2) \bullet N \rhd \begin{cases} M_1 \bullet N \text{ for } j < n \\ M_2 \cdot N \text{ for } j = n \end{cases}$$

For the $\lambda\&$-calculus we proved in [3] some fundamental theorems like the Church-Rosser property, the theorem of subject reduction and the strong normalization of some relevant sub-calculi.

## 4 $\lambda$_object

In the previous section we recalled the definition $\lambda\&$-calculus. It constitutes the paradigmatic calculus from which we draw our model. Now we enter the core of this paper by defining the meta-language $\lambda$_object. We pass from a calculus, which possesses an equational presentation, to a language, which thus is associated to a reduction strategy and a set of values. It is like if we had the $\lambda$-calculus and we wanted to define the SECD machine. The analogy is quite

suggestive since, as in the case of the SECD machine, we do not want an exact correspondence with the $\lambda$-calculus (e.g. as the one between the SECD machine and the $\lambda_V$: see [9]); rather we aim to define a language that implements the "general" behavior of the $\lambda\&$-calculus, and that constitutes a meta-language for object-oriented languages. A meta-language is conceived to "speak about", to describe a language. Thus it must possess the syntactic structures to reproduce the constructs of that language, structures that are not generally present in a calculus. Thus to reproduce object-oriented languages we provide $\lambda$_object with constructs to define new atomic types, to define a subtyping hierarchy on them, to work on the implementation of a value of atomic type, to define recursive terms, to change the type of a term and to deal with `super`. We give an operational semantics for the untyped terms, we define a notion of run-time type error and a type-checking algorithm. Finally we prove the subject reduction theorem (thus the correction of the type-checker) which plays a key role, being $\lambda$_object envisaged for typed object-oriented languages.

The main decision in the definition of $\lambda$_object is how to represent objects. This decision will drive the rest of the definition of the language. Running languages usually implement objects by records formed by three kinds of fields: fields containing the values of the instance variables, fields used by the system (for example for garbage collection) and a special field containing a reference to the class of the object. Obviously in this theoretical account we are not interested in the fields for the system, hence an object in $\lambda$_object will be formed only by the values of its instance variables (the so-called *internal state*) and by a *tag* indicating the class of the object. The tag of an object must univocally determine the type of the object, for in our approach the selection of a method is based on the type of the object. There are two reasonable ways to do it, and in both of them the name of the class is considered an atomic type:

(a) An object is a record whose fields are the instance variables plus a special empty field whose type is the name of the class
(b) An object is a record whose fields are the instance variables and which is given a tag, say $A$, by applying it to a special constructor $in^A$. In other terms, $in^{tag}$ is the constructor for the values of (atomic) type *tag* whose internal representation is given by the record of the instance variables.

For $\lambda$_object we choose to use the solution (b) for, even if it needs the introduction of new operations and new typing rules, it has the advantage that, as in our toy language, the type of an object is its class. Thus types will be conserved during the translation from the toy language to $\lambda$_object. Furthermore the operational semantics of $\lambda$_object will be simplified. Henceforth we will not distinguish among the terms "tag", "atomic type" and "class-name" since in $\lambda$_object they coincide.

To resume, in $\lambda$_object objects are "tagged terms" of the form $in^A(\mathrm{M})$ where $A$ is the tag and $M$ represents the internal state. When we have an overloaded application $M \bullet N$ we first reduce $M$ to a term $(M_1 \& M_2)$ and $N$ to a tagged term, and then we perform the branch selection according to the obtained tag, that is the name of the class of the object. The selected method must be able to access to the instance variables of the object, i.e. to get inside the *in* construct. To this purpose we use a function denoted *out* that composed with *in* gives the identity.

**Pretypes**
We use $A$ and $B$ (possibly subscripted) to denote atomic types.

$$\mathrm{T} ::= \mathrm{A} \mid \mathrm{T} \times \mathrm{T} \mid \mathrm{T} \to \mathrm{T} \mid \{(\mathrm{A}_1 \times \ldots \times \mathrm{A}_{m_1}) \to \mathrm{T}_1, \ldots, (\mathrm{B}_1 \times \ldots \times \mathrm{B}_{m_n}) \to \mathrm{T}_n\}$$

**Terms**
Terms are composed by an expression preceded by a (possibly empty) suite of declarations. We use the metavariable $M$ to range over expressions and $P$ to range over terms:

$$\begin{aligned} M ::= & \; x^T \mid \lambda x^T.M \mid MM \mid \varepsilon \mid M\&^T M \mid M \bullet M \\ & \mid <M, M> \mid \pi_1(M) \mid \pi_2(M) \mid \mu x^T.M \\ & \mid \mathbf{coerce}^A(M) \mid \mathbf{super}^A(M) \mid in^A(M) \mid out^A(M) \end{aligned}$$

$$P ::= M \mid \mathbf{let}\ A \le A_1, \ldots, A_n\ \mathbf{in}\ P \mid \mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P$$

Declarations cope with atomic types: they can be used to define the subtyping relation on atomic types and to declare a new atomic type by associating it to a representation type (i.e. the type of the internal state). More precisely 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 automatically defines two constructors $in^A{:}T \to A$ and $out^A{:}A \to T$ which form a retraction pair from $T$ to $A$.

**Tagged values**
We have to be a little more precise about tagged values: a tagged value is everything an overloaded function can perform its selection on. Thus it can be an object of the form $in^A(\mathrm{M})$ but also the coercion of an object, the super of an object and, since we have multiple dispatching, a tuple of objects. Thus a tag is either an atomic type or a product of atomic types. We use the metavariable $D$ to range over tags; tagged values are ranged over by $G^D$ where $D$ is the tag.

$$G^D ::= in^D(\mathrm{M}) \mid \mathbf{coerce}^D(\mathrm{M}) \mid \mathbf{super}^D(\mathrm{M}) \mid <G_1^{A_1}, G_2^{A_2}, \ldots, G_n^{A_n}>$$

In the last production $D \equiv (A_1 \times \ldots \times A_n)$

**Operational Semantics**
We define the *values* of $\lambda$_object, i.e. those terms which are considered as results; values are ranged over by $G$.

$$G ::= x \mid (\lambda x^T.M) \mid \varepsilon \mid (M_1 \&^T M_2) \mid <G_1, G_2> \mid \mathbf{coerce}^A(M) \mid \mathbf{super}^A(M) \mid in^A(M)$$

The operational semantics for $\lambda$_object is given by the reduction $\Rightarrow$; this reduction includes a type constraint system[10] $C$ that is built along the reduction by the declarations (**let** $A \le A_1 \ldots A_n$ **in** $P$) and that is used in the rule(s) for the selection of the branch. In the rules we use $\circ$ to denote either $\cdot$ or $\bullet$, $I$ to denote $\{D_1 \to T_1, \ldots, D_n \to T_n\}$ and $\overline{D}$ to denote the $\min_{i=1..n}\{D_i | C \vdash D \le D_i\}$

*Axioms*

$$\begin{array}{ll}
(C,\ out^{A_1}(in^{A_2}(M))) \Rightarrow (C,\ M) & \\
(C,\ out^{A_1}(\mathbf{coerce}^{A_2}(M))) \Rightarrow (C,\ out^{A_1}(M)) & \\
(C,\ out^{A_1}(\mathbf{super}^{A_2}(M))) \Rightarrow (C,\ out^{A_1}(M)) & \\
(C,\ \mu x.M) \Rightarrow (C,\ M[x := \mu x.M]) & \\
(C,\ (\lambda x.M) \cdot N) \Rightarrow (C,\ M[x := N]) & \\
(C,\ (M_1 \&^I M_2) \bullet G^D) \Rightarrow (C,\ M_1 \bullet G^D) & \text{if } D_n \neq \overline{D} \\
(C,\ (M_1 \&^I M_2) \bullet G^D) \Rightarrow (C,\ M_2 \cdot G^D) & \text{if } D_n = \overline{D} \text{ and } G^D \not\equiv \mathbf{super}^D(M) \\
(C,\ (M_1 \&^I M_2) \bullet G^D) \Rightarrow (C,\ M_2 \cdot M) & \text{if } D_n = \overline{D} \text{ and } G^D \equiv \mathbf{super}^D(M)
\end{array}$$

[10] At this stage it would be more correct to call it a "tag constraint system"

$(C\ ,\ \mathbf{let}\ A \leq A_1 \ldots A_n\ \mathbf{in}\ P) \Rightarrow (C \cup (A \leq A_1) \cup \ldots \cup (A \leq A_n)\ ,\ P)$
$(C\ ,\ \mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P) \Rightarrow (C\ ,\ P)$

*Context Rules*

$$\frac{(C\ ,\ M) \Rightarrow (C\ ,\ M')}{(C\ ,\ M \circ N) \Rightarrow (C\ ,\ M' \circ N)} \qquad \frac{(C\ ,\ M) \Rightarrow (C\ ,\ M')}{(C\ ,\ (N_1 \& N_2) \bullet M) \Rightarrow (C\ ,\ (N_1 \& N_2) \bullet M')}$$

$$\frac{(C\ ,\ M) \Rightarrow (C\ ,\ M')}{(C\ ,\ out^A(M)) \Rightarrow (C\ ,\ out^A(M'))}$$

The semantics for pairs is the standard one and thus it has been omitted. Three axioms and a rule give the behavior of *out* and let it accede to the internal state of an object. Functional application is implemented by call-by-name; anyway, this is not a central point of our paper and the call-by-value would fit as well.

The three axioms and two rules for overloaded functions deserve more attention: in an overloaded application we first reduce the function to an &-term and then its argument to a tagged value; then the reduction is performed according to the index of the &-term. In a sense, we perform a "call-by-tagged-value" (but for well typed programs this notion coincides with the usual call-by-value: see corollary 5). It is worth noting that this selection does not use types: no type checking is performed, only a match of tags and some constraints is done; indeed, we still do not have any "type" here, but just some tags indexing the terms. Note the difference when the tagged value is a super: in that case the argument of the super is passed to the selected branch instead of the whole tagged value.

Finally, the declaration ($\mathbf{let}\ \ A \leq A_1 \ldots A_n\ \mathbf{in}\ \ P$) modifies the type constraints in which the body $P$ is evaluated, while ($\mathbf{let}\ \ A\ \mathbf{hide}\ T\ \mathbf{in}\ P$) serves only to the type checker and thus, operationally, it is simply discarded.

**Programs and type errors**

The operational semantics above is given for untyped terms. Thus we define which terms are the programs of $\lambda_$object and when a reduction ends by a type error.

**Definition 2.** A program in $\lambda_$object is a closed term $P$ different from $\varepsilon$.

We use the notation $P \Rightarrow P'$ to say that $(C, P) \Rightarrow (C', P')$ for some $C$ and $C'$ and we denote by $\stackrel{*}{\Rightarrow}$the reflexive and transitive closure of $\Rightarrow$. Given a term $M$, we say that it is in *normal form* iff it does not exist $N$ such that $M \Rightarrow N$. Let $P$ be a closed term in normal form. If $P$ is not a value then it is always possible to use the context rules of the operational semantics to decompose $P$ to find the *least* subterm which is not a value and where the reduction is stuck. Let call this subterm the *critical subterm* of $P$. For example consider the following term:

$$(\,(M_1 \& M_2) \bullet ((\mathbf{super}^A(M)) \cdot (N))\,) \quad \cdot \quad (M')$$

This term is in normal form. Indeed, since it is an application we first try to reduce $((M_1 \& M_2) \bullet ((\mathbf{super}^A(M)) \cdot (N)))$; then for the second context rule we try to reduce $(\mathbf{super}^A(M)) \cdot (N)$; again, for the first context rule one tries to reduce $(\mathbf{super}^A(M))$; but it is a value different from a $\lambda$-abstraction and we are stuck. Thus, in this case, the critical subterm is $(\mathbf{super}^A(M)) \cdot (N)$. Note that the critical subterm (of a closed normal non-value term) always exists and is unique, since it is found by an algorithm which is deterministic (since the operational semantics is deterministic) and terminating (since the size of the term at issue always decreases).

**Definition 3 (type-error).** Let $P$ be a program. If $P \stackrel{*}{\Rightarrow} P'$, $P'$ is in normal form and it is not a value then we say that $P$ *produces a type error*. Furthermore if the critical subterm of $P'$ is of the form $((M_1 \&^T M_2) \bullet G^D)$ then we say that $P$ *produces an "undefined method" type error*.

The "undefined method" error is raised when we try to reduce an overloaded application of a &-term to a tagged value, and $\overline{D}$ is not defined. This means that it is not possible to select a branch for the object passed to the function. This can be due either because the set $\{D_i | D \leq D_i\ ,\ i = 1..n\}$ is empty or because it has no minimum. In object-oriented terms the former case means that the wrong message has been sent to the object and in the latter that the condition of multiple inheritance has not been respected.

### 4.1 The type system

We have defined programs and how to compute them; then we have singled out those computation that produce a "type error". Now we have to justify the use of the adjective *type* in front of the word "error". To this purpose we define a type system for the raw terms, so that the well-typed programs will not produce these errors.

**Types**

As in the case of $\lambda\&$-calculus and of our toy language we first define an order on the pretypes and then we select those that satisfy the conditions for covariance, multiple inheritance and input type uniqueness. The subtyping relation on pretypes and the good formation for types are exactly the same as those defined for our toy language in section 2.8, with the only modification that the set of atomic types is relative to a program and it is formed by all the pretypes that have been declared by a **let** ... **hide** definition (appendix A.1 contains the formal definition).

**Type checking rules**

The type checking rules are summarized in Appendix A.2. They are parametric in a type constraint system $C$ and a function $S$ from atomic types to types. These are used respectively to store the type constraints and the implementation types defined in the declarations in the rules [NewType] and [Constraint].

We want to interpret the construct `extend`; in $\lambda\&$ we can only add a new branch to an overloaded function but we cannot replace an existing branch by another of the same type. To obtain it in $\lambda$_object we use a weaker type system where we modify the typing of overloaded functions. The rules [Taut], [$\rightarrow$ Intro], [$\rightarrow$ Elim$_{(\leq)}$], [Taut$_\varepsilon$], [{}Elim] are the same as in $\lambda\&$ (with the obvious modifications to consider $C$ and $S$). The only rule we have to change is [{}Intro]: in the new version of the rule we use the meta operator on overloaded types $\cup$ which, we recall, denotes the usual set theoretic union.

Compare the rule [{}Intro] of appendix A.2 with the one of $\lambda\&$ in section 3. While the indexes are formed in the same way by simple juxtaposition, the type is obtained by using the union operator. When you concatenate to an overloaded function a branch whose type is different from the types of the branches already present, then the rule behaves as usual. If on the contrary the type of the new branch is already present in the type of the overloaded function then by $\cup$ the type of the function remains the same as before. However by the axioms of the operational semantics if there are two branches with the same type at index then the reduction always selects the rightmost of these branches, i.e. the latest

added. Furthermore, if you try to add to an overloaded type a branch whose input type is the same as the input type of a branch already present, then the type you obtain is well-formed if and only if the two branches have also the same output type: indeed if the two branches have different output types, then the (pre)type obtained by the union does not satisfy the condition (d) of well-formation for types. Worth to be mentioned are also the rules [IN] and [OUT]: note that the constructors of a type can be used only if that type has been previously defined by a **let_hide** declaration (i.e. if it belongs to $dom(S)$).

### 4.2 Main results

**Proposition 4.** *Let $P{:}T$; if $P$ is closed and in normal form then $P$ is a value.*

**Corollary 5.** *If a program is in normal form and possesses a product of atomic types then it is a tagged value.*

Recall that it is not possible to reduce inside a $\lambda$-abstraction. Therefore if in the evaluation of a program we reduce a term of the form $M \bullet N$, then in particular $N$ must be closed, To perform the selection of a branch $N$ must also be a value; thus, by the corollary above it must be a tagged value. Therefore in a well-typed program overloaded application is implemented by the usual call-by-value, since the only values allowed are tagged values.

**Theorem 6 (Subject Reduction).** *Let $P{:}T$; if $P \stackrel{*}{\Rightarrow} P'$ then $P'{:}T'$ and $T' \leq T$*

**Proposition 7.** *If $P \Rightarrow P'$ and $P$ is closed then also $P'$ is closed*

**Corollary 8.** *Let $P$ be a well-typed program. If $P \stackrel{*}{\Rightarrow} P'$ and $P'$ is in normal form then $P'$ is a value*

The last corollary states that well-typed programs stop only on values, and thus do not produce type errors.

**Encodings**

We hint how to encode updatable records in $\lambda$_object. For the definition of updatable records see [10]. We have no space to explain the encoding in detail. We just give its crude definition.

Let $L_1, L_2, \ldots$ be an infinite list of atomic types. Assume that they are isolated (i.e., for every type $T$, if $L_i \leq T$ or $T \leq L_i$, then $L_i = T$), and introduce for each $L_i$ a *constant* $\ell_i{:}\, L_i$. Then set $\langle\!\langle \ell_1 : T_1, \ldots, \ell_n : T_n \rangle\!\rangle = \{L_1 \to T_1, \ldots, L_n \to T_n\}$; the empty record $\langle\ \rangle = \varepsilon$, the updating $\langle r \leftarrow \ell_i = M \rangle = (r \& \lambda x^{L_i}.M)$ (where $x \notin FV(M)$); the field selection $r.\ell_i = r \bullet \ell_i$.

## 5 Translation

As we already said, we do not give a direct semantics to the toy language. Instead we translate its programs into $\lambda$-object. The formal translation is summarized in appendix B. In this section we give only the intuitive rules of the translation, which has the property to preserve the type; that is, a well-typed program of the toy language is translated in a well-typed term of $\lambda$_object of the same type (see theorem 9). This property validates the algorithm of type-checking we have defined for our object-oriented language since it assures that type-errors can never occur when running well typed programs.

- A message is simply an identifier of an overloaded function; thus it is translated in a variable possessing a (raw) overloaded type; i.e. $[\![m]\!] = m^{\{A_i \rightsquigarrow V_i\}_{i \in I}}$ where $\{A_i | i \in I\}$ is the set of the classes where the message $m$ has been defined, and the $V_i$'s are the corresponding types appearing in the interfaces.
- Message passing is the application of an overloaded function: $[\![ \texttt{[[}\mathit{exp}_0\ \mathit{exp}\ \mathit{exp}_1, \ldots, \mathit{exp}_n\ \texttt{]]} ]\!] = [\![\mathit{exp}]\!] \bullet [\![(\mathit{exp}_0, \mathit{exp}_1, \ldots, \mathit{exp}_n)]\!]$
- In the definition of a method, `self` represents the receiver of the message which invoked the method. Thus we translate a method $msg = exp$ into $\lambda self^A.[\![exp]\!]$, where $A$ is the current class. This will form a branch of the overloaded function denoted by (the translation of) the message $msg$.
- `new(`$A$`)` defines a value of type $A$. It is translated into $in^A(r)$ where $r$ is the record value containing the initial values of the instance variables of the class $A$.
- `update` unpacks *self* in its representation (record) type, modifies its value (i.e. the internal state) and packs it again in its original type. Thus for example $[\![\texttt{(update \{x= 3\})}]\!] = in^A(\langle out^A(self^A) \leftarrow x = 3\rangle)$; again $A$ is the current class.
- `super[`$A$`](`$exp$`)` and `coerce[`$A$`](`$exp$`)` are translated into $\mathbf{super}^A([\![exp]\!])$ and $\mathbf{coerce}^A([\![exp]\!])$ respectively.
- The operation `extend` corresponds to add a branch to an overloaded function. It has the following intuitive translation: $[\![$ `extend` $A\ m = exp$ `[[` ... `]]` `in` $exp' ]\!] = (\mathbf{let}\ m = (m \& \lambda self^A.[\![exp]\!])\ \mathbf{in}\ [\![exp']\!])$. For the case of multi-methods see the next point.
- Finally we have the most complex construct: the class definition. By a class definition we define a new atomic type, a set of type-constraints on this atomic type and some branches of overloaded function. The intuitive interpretation of say (`class` $A$ `is` $A_1$`,`$A_2$ `{`$x$`:Int=3}` $msg = exp$ `[[` $msg : T$ `]]` `in` $p$), when $exp$ is not a multi-method, is:

  $\mathbf{let}\ A\ \mathbf{hide}\ \langle\!\langle x : Int \rangle\!\rangle\ \mathbf{in}$
  $\mathbf{let}\ A \leq A_1, A_2\ \mathbf{in}$
  $\mathbf{let}\ msg = (msg \& \lambda self^A.[\![exp]\!])\ \mathbf{in}\ [\![p]\!]$

  If it is a multi-method then $exp$ must be of the form `&`...`&`.... For example $exp$ may be:

```
mesg = & fn(x1:C1; x2:C2) => exp1
       & fn(y1:C1; y2:C3) => exp2
       & fn(z:C2)         => exp3
```

  Then, using some pattern-matching in lambda calculus, the multi-method is translated into

$$\begin{aligned}\mathbf{let}\ mesg = (&mesg\\ &\&\lambda(self^A, x_1^{C_1}, x_2^{C_2}).[\![\texttt{exp1}]\!]\\ &\&\lambda(self^A, y_1^{C_1}, y_2^{C_3}).[\![\texttt{exp2}]\!]\\ &\&\lambda(self^A, z^{C_2}).[\![\texttt{exp3}]\!]\\ &)\end{aligned}$$

  Of course the initial value 3 of $x$ must be recorded during the translation so that this value can be used in the translation of `new(`$A$`)`. Finally if there is more then one method, then they can be mutually recursive; thus we need to use a fixpoint operator in their definition.

More formally the interpretation function will be parameterized by an environment of the initial states, an environment of instance variables and by the type of the current object (see appendix B). So that the theorem of correction of the type system for the toy language is formulated as follows:

**Theorem 9.** *For every type constraint $C$, type environment $\Gamma$ and for every $I \in InitState$ and $S : ClassNames \rightarrow$* **RecordTypes** *such that for any $A$ atomic $I(A){:}S(A)$, if $C;S;\Gamma \vdash p{:}T$ then $C;S \vdash [\![p]\!]_{\star\, I\star\, (self)}{:}T$*

**Acknowledgments**
Many ideas of this work come from several discussions with Luca Cardelli, Giorgio Ghelli, Giuseppe Longo, Eugenio Moggi and Benjamin Pierce. We are especially grateful to Allyn Dimock, Maribel Fernández and Benjamin Pierce for their precise as well as useful comments on an earlier version of this paper and to Jean-Christophe Filliatre and François Pottier who implemented with the author an interpreter for λ_object.

## References

1. H.P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*. North-Holland, 1984. Revised edition.
2. G. Castagna. $F^{\&}_{\leq}$ : integrating parametric and "ad hoc" second order polymorphism. In *Proc. of the 4th International Workshop on Database Programming Languages*, Workshops in Computing, New York City, September 1993. Springer-Verlag.
3. G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping, 1992. To appear in *Information and Computation*. An extended abstract has appeared in the proceedings of the *ACM Conference on LISP and Functional Programming*, pp.182-192; San Francisco, June 1992.
4. G. Castagna, G. Ghelli, and G. Longo. A semantics for λ&-*early*: a calculus with overloading and early binding. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664 in LNCS, pages 107–123, Utrecht, The Netherlands, March 1993. Springer-Verlag. TLCA'93.
5. W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. *17th Ann. ACM Symp. on Principles of Programming Languages*, January 1990.
6. G. Ghelli. A static type system for message passing. In *Proc. of OOPSLA '91*, 1991.
7. S.K. Keene. *Object-Oriented Programming in* Common Lisp*: A Programming Guide to CLOS*. Addison-Wesley, 1989.
8. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall International Series, 1988.
9. G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1, 1975.
10. Mitchell Wand. Complete type inference for simple objects. In *2nd Ann. Symp. on Logic in Computer Science*, 1987.

## A Type system of λ_object

### A.1 Types

1. $A \in_{C,S}$ **Types** for each $A \in dom(S)$
2. if $T_1, T_2 \in_{C,S}$ **Types** then $T_1 \rightarrow T_2 \in_{C,S}$ **Types** and $T_1 \times T_2 \in_{C,S}$ **Types**
3. if for all $i, j \in I$
   **(a)** $(D_i, T_i \in_{C,S}$ **Types**$)$
   **(b)** if $C \vdash D_i \leq D_j$ then $C \vdash T_i \leq T_j)$
   **(c)** for all maximal type $D$ in $LB_C(\{D_i, D_j\})$ there exists $h{\in}I$ s.t. $D_h = D$
   **(d)** if $i \neq j$ then $D_j \neq D_j$
   then $\{D_i \rightarrow T_i\}_{i\in I} \in_{C,S}$ **Types**

### A.2 Typing rules

$$[\textsc{NewType}] \quad \frac{C\ ,\ S[A \leftarrow T] \vdash P : U}{C, S \vdash \textbf{let}\ A\ \textbf{hide}\ T\ \textbf{in}\ P : U} \quad A \notin dom(S), T \in_{C,S} \textbf{Types} \text{ and } T \text{ not atomic}$$

$$[\textsc{Constraint}] \quad \frac{C \cup (A \leq A_i), S \vdash P : T}{C, S \vdash \textbf{let}\ A \leq A_1, \ldots, A_n\ \textbf{in}\ P : T} \quad C \vdash S(A) \leq S(A_i) \text{ and } A \text{ do not appear in } C$$

$$[\textsc{Taut}] \quad C, S \vdash x^T : T$$

$$[\rightarrow \textsc{Intro}] \quad \frac{C, S \vdash M : T'}{\lambda x^T.M : T \rightarrow T'} \quad T \in_{C,S} \textbf{Types}$$

$$[\rightarrow \textsc{Elim}_{(\leq)}] \quad \frac{C, S \vdash M : U \rightarrow T \quad N : W}{C, S \vdash MN : T} \quad C \vdash W \leq U$$

$$[\textsc{Taut}_\varepsilon] \quad C, S \vdash \varepsilon : \{\}$$

$$[\{\}\textsc{Intro}] \quad \frac{C, S \vdash M : W_1 \leq \{U_i \rightarrow T_i\}_{i \leq (n-1)} \quad C, S \vdash N : W_2 \leq U_n \rightarrow T_n}{C, S \vdash (M \&^{\{U_i \rightarrow T_i\}_{i \leq n}} N) : \{U_i \rightarrow T_i\}_{i \leq (n-1)} \cup U_n \rightarrow T_n} \quad \{U_i \rightarrow T_i\}_{i \leq (n-1)} \cup U_n \rightarrow T_n \in_{C,S} \textbf{Types}$$

$$[\{\}\textsc{Elim}] \quad \frac{C, S \vdash M : \{U_i \rightarrow T_i\}_{i \in I} \quad C, S \vdash N : U}{C, S \vdash M \bullet N : T_j} \quad U_j = \min_{i \in I}\{U_i | C \vdash U \leq U_i\}$$

$$[\textsc{Pair}] \quad \frac{C, S \vdash M : T_1 \quad C, S \vdash N : T_2}{C, S \vdash < M\ , N > : T_1 \times T_2}$$

$$[\textsc{Proj}] \quad \frac{C, S \vdash M : T_1 \times T_2}{C, S \vdash \pi_i(M) : T_i} \quad \text{for } i = 1, 2$$

$$[\textsc{Coerce}] \quad \frac{C, S \vdash M : B}{C, S \vdash \textbf{coerce}^A(M) : A} \quad C \vdash B \leq A,\ A \in_{C,S} \textbf{Types}$$

$$[\textsc{Super}] \quad \frac{C, S \vdash M : B}{C, S \vdash \textbf{super}^A(M) : A} \quad C \vdash B \leq A,\ A \in_{C,S} \textbf{Types}$$

$$[\textsc{In}] \quad \frac{C, S \vdash M : T}{C, S \vdash in^A(M) : A} \quad C \vdash T \leq S(A),\ A \in_{C,S} \textbf{Types}$$

$$[\textsc{Out}] \quad \frac{C, S \vdash M : B}{C, S \vdash out^A(M) : S(A)} \quad C \vdash B \leq A,\ A \in_{C,S} \textbf{Types}$$

$$[\textsc{Fix}] \quad \frac{C, S \vdash M : T}{\mu x^T.M : T} \quad T \in_{C,S} \textbf{Types}$$

## B Translation

Let $Envs = Vars \rightarrow \textbf{RawTypes}$, $InitState = ClassNames \rightarrow RecordValues$ then we have the following definitions:

**Definition 10.** $\mathcal{T}[\![.]\!] : \mathcal{L} \rightarrow Vars \rightarrow \textbf{Types}$

1. $\mathcal{T}[\![\texttt{class}\ B\ \texttt{is}\ A_1 \ldots A_q\ \ r : R\ \ \texttt{m}_1\texttt{=}exp_1 \ldots \texttt{m}_n\texttt{=}exp_n\ [\![\ \texttt{m}_1 : V_1 \ldots \texttt{m}_n : V_n\ ]\!]\ \texttt{in}\ p]\!](m) =$
$= \begin{cases} \mathcal{T}[\![p]\!](m_j) \cup \{B \rightsquigarrow V_j\} & \text{for } m = m_j \\ \mathcal{T}[\![p]\!](m) & \text{else} \end{cases}$
2. $\mathcal{T}[\![.]\!]$ is the function which returns $\{\}$ in all the other cases.

**Definition 11.** Let $p'$ denote the program $\texttt{class}\ B\ \texttt{is}\ A_1, \ldots, A_q\ \ r : R\ \ \texttt{m}_1\texttt{=}exp_1 \ldots \texttt{m}_n\texttt{=}exp_n\ [\![\ \texttt{m}_1 : V_1 \ldots \texttt{m}_n : V_n\ ]\!]\ \texttt{in}\ p$ then define:

– $\mathcal{M}[\![p']\!]_{\star\ I\ A}(m) = \begin{cases} \pi_j(M) & \text{for } m = m_j \\ \mathcal{M}[\![p]\!]_{\star\ I\ A}(m) & \text{else} \end{cases}$

Where $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \cup \{B \rightarrow T_i\}]_{i=1..n}$ and $M$ has the following definition:
$M \equiv \mu(m_1^{\mathcal{T}[\![p']\!](m_1)}, \ldots, m_n^{\mathcal{T}[\![p']\!](m_n)}).(M_1, \ldots, M_n)$
where for $j \in [1..n]$, $M_j$ has the following definition:

1. If $V_j$ is a raw type then $M_j$ has the following form
$$((\mathcal{M}[\![p]\!]_{\star'\ I\ A}(m_j)) \&^{\mathcal{T}[\![p]\!](m_j) \oplus \{B \rightsquigarrow V_j\}} \lambda self^B.\Im[\![exp_j]\!]_{\star'[self \leftarrow B]\ I[B \leftarrow r]\ B})$$
2. If $j \in [1..n]$ and $V_j \equiv \#\{D_i \rightarrow T_i\}_{i=1..h}$, then $exp_j$ must be of the following form:
$$\texttt{\&\ fn(x}_1\texttt{:}\ D_1\texttt{)\ =>}\ exp_{j_1} \ldots \texttt{\&\ fn(x}_h\texttt{:}\ D_h\texttt{)\ =>}\ exp_{j_h}$$
then $M_j$ is defined in the following way:
$$(\cdots((\mathcal{M}[\![p]\!]_{\star'\ I\ A}(m_j) \&^{\mathcal{T}[\![p]\!](m_j) \oplus \{B \times D_{\sigma(1)} \rightarrow T_{\sigma(1)}\}} \lambda(self^B, x_{\sigma(1)}^{D_{\sigma(1)}}).\Im[\![exp_{j_{\sigma(1)}}]\!]_{\star'[self \leftarrow B]\ I[B \leftarrow r]\ B}) \vdots \&^{(\mathcal{T}[\![p]\!](m_j) \cup \ldots \cup \{B \times D_{\sigma(h-1)} \rightarrow T_{\sigma(h-1)}\}) \oplus \{B \times D_{\sigma(h)} \rightarrow T_{\sigma(h)}\}} \lambda(self^B, x_{\sigma(h)}^{D_{\sigma(h)}}).\Im[\![exp_{j_{\sigma(h)}}]\!]_{\star'[self \leftarrow B]\ I[B \leftarrow r]\ B})$$
where $\sigma$ is a permutation generated by any algorithm with the property that if $h < k$ then $D_k \not\leq D_h$

– $\mathcal{M}[\![.]\!]$ is the function which returns $\varepsilon$ in all the other cases.

**Definition 12 (Translation).**

1. $\Im[\![\texttt{x}]\!]_{\star\ I\ A} = x^{\star(x)}$
2. $\Im[\![exp_1\,(exp_2)]\!]_{\star\ I\ A} = \Im[\![exp_1]\!]_{\star\ I\ A} \Im[\![exp_2]\!]_{\star\ I\ A}$
3. $\Im[\![\texttt{fn(x:}T\texttt{)\ =>}\ exp]\!]_{\star\ I\ A} = \lambda x^T.\Im[\![exp]\!]_{\star[x \leftarrow T]\ I\ A}$
4. $\Im[\![\texttt{let x:}T\ \texttt{=}exp\ \texttt{in}\ exp'\texttt{'}]\!]_{\star\ I\ A} = (\lambda x^T.\Im[\![exp]\!]_{\star[x \leftarrow T]\ I\ A})(\Im[\![exp'\texttt{'}]\!]_{\star\ I\ A})$
5. $\Im[\![(exp_1, \ldots exp_n)]\!]_{\star\ I\ A} = <\Im[\![exp_1]\!]_{\star\ I\ A}, \ldots, \Im[\![exp_n]\!]_{\star\ I\ A}>$
6. $\Im[\![\texttt{fst(}exp\texttt{)}]\!]_{\star\ I\ A} = \pi_1(\Im[\![exp]\!]_{\star\ I\ A})$
7. $\Im[\![\texttt{snd(}exp\texttt{)}]\!]_{\star\ I\ A} = \pi_2(\Im[\![exp]\!]_{\star\ I\ A})$
8. $\Im[\![\texttt{new(B)}]\!]_{\star\ I\ A} = in^B(I(B))$
9. $\Im[\![\texttt{[}exp_0\ \ exp\ \ exp_1,\ \ldots,\ \ exp_n\texttt{]}]\!]_{\star\ I\ A} = \Im[\![exp]\!]_{\star\ I\ A} \bullet \Im[\![(exp_0, exp_1, \ldots, exp_n)]\!]_{\star\ I\ A}$
10. $\Im[\![\texttt{super[}B\texttt{](}exp\texttt{)}]\!]_{\star\ I\ A} = \mathbf{super}^B(\Im[\![exp]\!]_{\star\ I\ A})$
11. $\Im[\![\texttt{coerce[}B\texttt{](}exp\texttt{)}]\!]_{\star\ I\ A} = \mathbf{coerce}^B(\Im[\![exp]\!]_{\star\ I\ A})$
12. $\Im[\![\texttt{self}]\!]_{\star\ I\ A} = self^A$
13. $\Im[\![\texttt{self.}\ell]\!]_{\star\ I\ A} = (out^A(self^A)).\ell$
14. $\Im[\![\texttt{(update}\ r\texttt{)}]\!]_{\star\ I\ A} = in^A(\langle out^A(self^A) \leftarrow \ell_1 = \Im[\![exp_1]\!]_{\star\ I\ A} \ldots \leftarrow \ell_n = \Im[\![exp_n]\!]_{\star\ I\ A}\rangle)$
where $r \equiv \{\ell_1 = exp_1; \ldots; \ell_n = exp_n\}$
15. $\Im[\![\texttt{extend}\ B\ \texttt{m}_1\texttt{=}exp_1\texttt{...;m}_n\texttt{=}exp_n\ \ \texttt{[[m}_1\texttt{:}V_1\texttt{;...;m}_n\texttt{:}V_n\texttt{]]}\ \texttt{in}\ exp]\!]_{\star\ I\ A} = (\lambda m_1^{\star(m_1) \cup \{B \rightsquigarrow V_1\}} \ldots \lambda m_n^{\star(m_n) \cup \{B \rightsquigarrow V_n\}}.\Im[\![exp]\!]_{\star'\ I\ A})\ M_1 \ldots M_n$

where

(a) If $V_j$ is a raw type then
$$M_j \equiv ((m_j^{\star(m_j)} \&^{\star(m_j) \oplus \{B \rightsquigarrow V_j\}} \lambda self^B.\Im[\![exp_j]\!]_{\star'[self \leftarrow B]\ I[B \leftarrow r]\ B})$$

(b) If $j \in [1..n]$ and $V_j \equiv \#\{D_i \rightarrow T_i\}_{i=1..h}$, then $exp_j$ must be of the following form: $\texttt{\&\ fn(x}_1\texttt{:}\ D_1\texttt{)\ =>}\ exp_{j_1} \ldots \texttt{\&\ fn(x}_h\texttt{:}\ D_h\texttt{)\ =>}\ exp_{j_h}$
then $M_j$ is defined in the following way:
$$(\cdots((m_j^{\star(m_j)} \&^{\star(m_j) \oplus \{B \times D_{\sigma(1)} \rightarrow T_{\sigma(1)}\}} \lambda(self^B, x_{\sigma(1)}^{D_{\sigma(1)}}).\Im[\![exp_{j_{\sigma(1)}}]\!]_{\star'[self \leftarrow B]\ I[B \leftarrow r]\ B}) \vdots \&^{(\star(m_j) \cup \ldots \cup \{B \times D_{\sigma(h-1)} \rightarrow T_{\sigma(h-1)}\}) \oplus \{B \times D_{\sigma(h)} \rightarrow T_{\sigma(h)}\}} \lambda(self^B, x_{\sigma(h)}^{D_{\sigma(h)}}).\Im[\![exp_{j_{\sigma(h)}}]\!]_{\star'[self \leftarrow B]\ I[B \leftarrow r]\ B})$$
where $\sigma$ has the usual property.

16. Let $p \equiv$ `class` $B$ `is` $A_1 .. A_q$ $r{:}R$ $\mathtt{m}_1{=}exp_1 \ldots \mathtt{m}_n{=}exp_n$ `[[` $\mathtt{m}_1{:}T_1 \ldots \mathtt{m}_n{:}T_n$ `]]` `in` $p'$
then $\Im[\![p]\!]_{\star\ I\ A} =$ **let** $B$ **hide** $R$ **in let** $B \leq A_1 ... A_q$ **in**
$\Im[\![p']\!]_{\star\ I[B \leftarrow r]\ A}[m_i^{(\mathcal{T}[\![p]\!](m_i))} := \mathcal{M}[\![p]\!]_{\star\ I\ A}(m_i)]_{i=1..n}$

This article was processed using the LaTeX macro package with LLNCS style