A meta-language for typed object-oriented languages

Giuseppe Castagna

C.N.R.S.
LIENS, 45 rue d'Ulm, 75005 Paris. FRANCE
e-mail: castagna@dmi.ens.fr

In [13] we defined the $\lambda\&$-calculus, a simple extension of the typed λ-calculus to model typed object-oriented languages. This paper is the continuation or, rather, the companion of [13] since it analyzes the practical counterpart of the theoretical issues introduced there. Indeed, to develop a formal study of type systems for object-oriented languages we define a meta-language based on $\lambda\&$ and we show, by a practical example, how it can be used 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 correctness 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. An important novelty with respect to previous systems is that we show how to model multiple dispatch also in the presence of a notion of *receiver* (i.e. of a privileged argument to which the message is passed), a notion that is absent in languages like CLOS.

1 Introduction

In [13] we introduced the $\lambda\&$-calculus. It is a simple extension of the typed lambda calculus to deal with overloaded functions, subtyping and late binding.

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

An extended abstract of this paper has appeared in the *13th Conference on Foundations of Software Technology and Theoretical Computer Science*. Bombay, December 1993. LNCS 761. Springer-Verlag

Article published in Theoretical Computer Sciences 151(2):297-352, 1995

The main motivation of its definition was to give a kernel calculus possessing the key properties of object-oriented programming. In the same paper, we showed how this calculus could be intuitively used to model some features of object-oriented programming. This 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 had a completely new vision of some of the mechanisms more or less present in object-oriented programming: it improved our comprehension of some features like class definition, inheritance and code sharing. And 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 (such as first class messages or a generalization of "`super`") the usual models could not. But above all it suggested a type discipline for objects, easy to understand (also by a programmer, even if he/she has no knowledge of second order logic) or, at least, to explain.

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 λ_object. This language is still based on the key mechanisms of $\lambda\&$ (essentially, overloading and late 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, to modify the discipline of dispatching etc.) that are necessary to reproduce the constructs of a programming language and that $\lambda\&$ lacks.

We also show, by a practical example, how to use λ_object to prove properties of an object-oriented language. To this purpose we define a simple toy object-oriented language. This language is a mix of Objective-C and CLOS constructs: there is a notion of *receiver* (i.e. a privileged object to which the message is sent), but also the possibility of performing multiple dispatch. [3]. We also define an algorithm to type-check the programs of this language. We then translate the programs of the toy object-oriented language into λ_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, this implies that the reduction of the translated program never goes wrong on a type error. In particular this proves the correctness of the type-checker for the toy language.

---

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

[3] In λ_object, the modelling of multiple dispatch à la CLOS (i.e. without identifying a receiver) is even simpler.

This paper constitutes the continuation or, rather, the companion of [13], since it shows the practical counterpart of the theoretical issues introduced in [13]. Consequently, the logical order of the paper would be the definition of λ_object, of the toy language, of the translation and the proof of the properties. But we do not follow this order in the presentation, where we start with the toy language. Indeed, the reader may not be acquainted with the model induced by $\lambda\&$ (and in particular with the implementation of method lookup as the resolution of overloading). Thus we prefer to recall this model along the lines describing the toy language (whose constructs should be familiar to the reader), rather than introduce it directly by the meta-language. Therefore, 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 λ_object: we define its operational semantics, its type discipline and we prove the subject reduction theorem. In section 5 we describe the translation of the toy language into λ_object, and, via this translation, we prove the correctness of the type discipline for the toy language. In section 6 we state the precise correspondence between λ_object and $\lambda\&$. Section 7 describes the addition of polymorphism to the toy language. Section 8 concludes the paper and suggests some directions for future work. For space reasons we cannot give full details of every issue in the paper. These can be found in the author's PhD. thesis [10].

2 The toy language

In this section we briefly discuss (a certain kind of) object-oriented programming by gradually introducing a toy functional object-oriented language. For the functional core of this language we use the syntax of an explicitly typed version of ML. The syntax of the object-oriented components is inspired by Objective C (see [21] and [20]). This does not aim to be a comprehensive presentation of object-oriented features. Far from that, it tends to present some kernel features of object-oriented programming from our particular perspective, which is the one we acquired in defining and developing the $\lambda\&$-calculus, the basic calculus our model. In this section we just give an informal presentation of the language. The formal presentation can be found in appendix A.

2.1 Objects and messages

Object-oriented programs are built around *objects*. An object is a programming unit that associates data with the operations that can use or affect these data. These operations are called *methods*; the data they affect are the *instance variables* of the object. In short, an object is a programming unit formed by a data structure and a group of procedures which affect it. The instance variables of an object are private to the object itself; they can be accessed only

through the methods of the object. An object can only respond to *messages* that are *sent* or *passed* to it. A *message* is simply the name of a method that was defined for that object.

Message passing is a key feature of object-oriented programming: the execution of an object-oriented program proceeds by the exchange of messages between objects. Every language has its own syntax for messages. For our toy language we use the following one:

`[` *receiver message* `]`

The *receiver* is an object (or more generally an expression returning an object). When it receives a message, the run-time system selects among the methods defined for that object the one whose name corresponds to the passed message. The existence of such a method should be statically checked (i.e. verified at compile time) by a type checking algorithm.

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 [19]). The other is to consider message passing as functional application, where the message is (the identifier of) the function and the receiver is its argument (this technique is used by the languages CLOS [16] and Dylan [2]). In this paper we choose this second solution. However, in order to formalize this approach, ordinary functions do not suffice. The fact that a method *belongs* to a specific object implies that the implementation of *message passing* is different from that of the customary function application. Two main characteristics distinguish messages from ordinary functions:

- *Overloading*: Two objects can 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. However, all the objects of a given *class* (e.g. all objects of *class* `matrix`) respond to a message in the same way [4]. If we assume that the type of an object is its class, then this amounts to saying that *messages are identifiers of overloaded functions*, since the code to execute is chosen according to the type (the class) of the argument (the receiver). Each method associated to the message m constitutes a *branch* of the overloaded function referred to by m.
- *Late 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 message can be decided only at run-time when the receiving object is known. This feature, called *late-binding* [5], is one of

[4] This is not true in delegation-based object-oriented languages.

[5] Object-oriented literature, usually prefers the to call it *dynamic binding*. For the difference between late and dynamic binding and the reason why we use the adjective "late" see [12,10].

the most powerful characteristics of object-oriented programming, since it allows incremental definition and code reuse. The advantage of late binding is shown by the following example: suppose that a graphical editor is coded using an object-oriented style; it uses the classes *Line* and *Square* which are subclasses (subtypes) of *Picture*; suppose also that a method *draw* is defined on all three classes. If the selection of the methods is performed at compile time (we call this discipline of selection "early binding"), then an overloaded function application like the following one

$$\lambda x^{Picture}.(\ldots [\ x\ draw\ ] \ldots)$$

is always executed using the *draw* code for pictures, since the compile time type of x is *Picture*. Using late binding, the code for *draw* is chosen only when the x parameter has been bound and evaluated, on the basis of the run-time type of x, i.e. according to whether x is bound to a line or a square or a picture.

Therefore, in our model, overloading and late binding are the basic mechanisms. [6]

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 object-oriented programming; 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
```

[6] The use of late 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 7.

```
]]
```

A `2DPoint` object represents a point of the cartesian plane: two instances variables define the position of the object; it responds to messages to return its *norm*, to *erase* its x-coordinate and to *move*.

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

A *program* in our toy language is a sequence of definitions 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 the reuse of code written for other classes and which concerns the definition of the objects. The other is subtyping, which is the mechanism that allows the use of 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 [14]). 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 substitutivity given by subtyping) [7] . We use the keyword `is` in the class definition to define the *subtype* relation among classes. A typical example of its use is:

```
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 methods `norm` and `erase` are *inherited* from `2DPoint`. The method `move` is redefined (overridden) so that if a colored point is moved, then its color is set to white. 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

[7] In the overloading-based model, pure inheritance can be dealt with by the introduction of union types in a second order framework: see chapter 11 of [10]

(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:

(i) *state coherence*: The set of the instance variables of a given class should contain those of all its superclasses. Moreover common variables must appear with the same type.

(ii) *covariance*: A method that overrides another method must specialize it, in the sense that the type of the new method must be a subtype of the type of the old method.

(iii) *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

To avoid ambiguity in the selection, 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 mathematically cleaner.

2.4 Extending classes

Refinement is not the only way to specialize classes. It would be very annoying if every time we have to add a method to a class we were obliged to define a new class: the existing objects of the old class could not use the new method. The same is true also in the case that a method of a class must be redefined: *overriding* would not suffice. For this reason, some object-oriented languages offer the capability to add new methods to existing classes or to redefine the old ones (this capability is very important in persistent systems). In our toy language this can be done by the following expression:

```
extend classname
       methodDefinitions
       interface
   in expr
```

the newly defined methods are available in the expression *expr*. Remark that, by this construction, we do not define a new class but only new methods; in other terms we do not modify the existing types but only (the environment of) the expressions. This is possible in our system since the type of an object is not bound to the procedures that can work on it (this is the peculiar feature that distinguish it from the abstract data types and the "objects as records" approaches). Finally, 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. [8]

[8] Addition and redefinition of methods are implemented by some object-oriented languages (e.g. Objective-C [21], CLOS [16] and Dylan [2]). Anyway it must be clear that these features constitute a trade-off between encapsulation and flexibility, and

The use of the reserved keyword `self` is well-known: it denotes, in a method, the receiver of the message that invoked that 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 necessarily appear in the receiver position, but it is a first-class value (i.e. it can appear in any context its type it 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 [9]. 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 coercions. By a coercion one changes 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 coerce changes it for the whole life of the object. The syntax is the same as that of `super`: thus we write `coerce[`A`](`*exp*`)` to change to A the type of the expression *exp*. A short example can clarify the behavior of `super` and `coerce`: suppose we have these three classes

- a class A in which we define a method m_1
- a class B subtype of A in which we define a method m_2 whose body contains the expression `[self` m_1`]`
- a class C subtype of B in which we override both m_1 and m_2.

Let M be an object of type C. Consider now `[ super[`B`](`M`)` m_2 `]` and `[ coerce[`B`](`M`)` m_2 `]`. In both cases the method selected is the one defined in B. But in the body of m_2 the meaning of `self` is, in the former case, M, while in the latter it is `coerce[`B`](`M`)`: therefore the method used for `[self` m_1`]` will be the one defined in C when using `super` and the one in A when

---

thus should be coupled with some further mechanism of protection. For example Dylan has a function `freeze-methods` which prevents certain methods associated with a message from being replaced or removed.

[9] For instance, this is what is done in Fibonacci [1], developed at the University of Pisa

using `coerce`. To sum up, `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) [10].

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. [18]). 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 section).

The number of parameters 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 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

[10] 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 messages passed to it.

functions accepting or calculating messages (indeed overloaded functions) and to write message passing of the form `[`*receiver* $f(x)$`]` (see [10] 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 type of each parameter of each branch must be 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: use the same interface as in the section before but without "`#`" i.e. `[[ msg:{Int -> (Int -> Bool), Int x Int -> Bool}]]`; the absence of `#` indicates that `msg` is now a ordinary method returning an overloaded function (thus the expression `[3 [ M msg ] ] 4` selects the first branch while `[ (3,4)[ M msg ] ]` selects the second one). [11]

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 (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*

[11] 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 `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 [18]).

type uniqueness).

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

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

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

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 $T \cup \{U_n \to T_n\}$ 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.

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 be substituted for another overloaded function if for every branch of the latter there is at least one in the former that can substitute for 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:

$$\frac{\text{for all } i \in I, \text{ there exists } j \in J \text{ such that } C \vdash D''_i \leq D'_j \text{ and } C \vdash U'_j \leq U''_i}{C \vdash \{D'_j \to U'_j\}_{j \in J} \leq \{D''_i \to U''_i\}_{i \in I}}$$

where C is a *type constraint system*, formed by the union of constraints like $(A_1 \leq A_2)$, that records the subtyping relation on the atomic types. The subtyping rules are summarized in appendix A.2.

Well-formed types

By subtyping relation above we select among the raw types those that satisfy the conditions given for the refinement in section 2.3. In particular the last two conditions, reformulated in terms of overloading, become:

(i) *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.

(ii) *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.

The inheritance condition, as we formulated it in the previous section, said that methods in common to more than one unrelated ancestor must be redefined to disambiguate the selection. To see that this is equivalent to the rule (2.) we have written above, note that when we define a class by refinement of some other classes, this exactly corresponds to defining a common subtype, which is also maximal (since it is not possible in the language to construct a type greater than another which has already been defined). If two *unrelated* ancestors respond to a same message then they both appear as input types in the type of this message and, thus, the condition says that a new branch (method) must be defined for the new maximal subtype

The types that satisfy the two conditions above are called *well-formed types* (the condition of state coherence concerns the definition of a class and will be checked directly on terms). We denote the set of well-formed types by **Types**. Since the membership to **Types** depends on the definition of the subtyping relation on the atomic types, we index the symbol of membership by a type constraint system.

Notation 1 *Let* $S \subseteq$ **Types**. *We denote by* $LB_C(S)$ *the set* $\{T \in_C \textbf{Types} \mid \forall T' \in S,\ C \vdash T \leq T'\}$ *of lower bounds of* S *with respect to the subtyping relation defined by* C. □

Definition 2 (well-formed types)

(i) $A \in_C$ **Types** *for each* A *atomic*
(ii) if $T_1, T_2 \in_C$ **Types** *then* $T_1 \to T_2 \in_C$ **Types** and $T_1 \times T_2 \in_C$ **Types**
(iii) if for all $i, j \in I$
- **(a)** $D_i, T_i \in_C$ **Types**
- **(b)** *if* $C \vdash D_i \leq D_j$ *then* $C \vdash T_i \leq T_j$
- **(c)** *for all maximal types* D *in* $LB_C(\{D_i, D_j\})$ *there exists* $h \in I$ *such that* $D_h = D$
- **(d)** *if* $i \neq j$ *then* $D_i \neq D_j$

then $\{D_i \to T_i\}_{i \in I} \in_C$ **Types**

By analogy we will denote by **AtomicTypes** the set of atomic types (the names of all classes) and by **RecordTypes** the set of the record types whose fields are associated to well-formed types.

Rules for Terms

(i) The type of an object is (the name of) its class.
(ii) The type of a coercion is the class specified in it, provided that it is a supertype of the type of the argument.
(iii) The type of a super is the class specified in it, provided that it is a supertype of the type of the argument

(iv) The type of `self` is the name of the class whose definition `self` appears in.

(v) The type of an overloaded function is the set of the types of its branches

(vi) The type of an overloaded application is the output type of the branch whose input type "best approximates" the *static* 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. Consider again message passing: we said that the receiver is the argument of the overloaded function the selection is based on. 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:

(i) The method `msg=`*exp* is not a multi-method and returns (according to the interface) the type T. This corresponds to adding to the overloaded function denoted by `msg` the branch `fn(self:`C`).`*exp* whose type is $C \to T$.

(ii) We have the multi-method

$$\texttt{msg = \& fn(x}_1\texttt{:A}_1\texttt{,...,x}_i\texttt{:A}_i\texttt{) =>}\ expr_1 \quad \vdots \quad \texttt{\& fn(y}_1\texttt{:B}_1\texttt{,...,y}_j\texttt{:B}_j\texttt{) =>}\ expr_n$$

which returns the type `#{(`$A_1 \times \ldots \times A_i$`)` $\to T_1$`,...,(`$B_1 \times \ldots \times B_j$`)` $\to T_n$`}`. This corresponds to adding to the overloaded function denoted by `msg` the n branches `fn(self:`C`,` $\texttt{x}_1\texttt{:}A_1, \ldots, \texttt{x}_i\texttt{:}A_i$`) =>` $expr_1 \ldots$ `fn(self:`C`,` $\texttt{y}_1\texttt{:}B_1$`, ...,` $\texttt{y}_j\texttt{:}B_i$`) =>` $expr_n$ of types `(`$C \times A_1 \times \ldots \times A_i$`)` $\to T_1, ...,$ `(`$C \times B_1 \times \ldots \times B_j$`)` $\to T_n$

In conclusion a message denotes an overloaded function that possesses one branch for every class in which a method has been defined for it, and one branch for every branch of a multi-method associated to it. Message passing is typed as the overloaded application. The selection of the branch corresponds to the search for 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 mechanism of Smalltalk [17]).

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

the rules are summarized in appendix A.3.

[New] $$C; S; \Gamma \vdash \texttt{new}(A) : A \qquad A \in dom(S)$$

The type of a new object is the name of its class. Of course this class must have been previously defined, and thus we check that $A \in dom(S)$.

[Read] $$C; S; \Gamma \vdash \texttt{self}.\ell : T \qquad S(\Gamma(\texttt{self})) = \langle\!\langle ...\ell : T... \rangle\!\rangle$$

The expression $\texttt{self}.\ell$ reads the value of an instance variable of an object and thus it must be contained inside the body of a method. Then $\Gamma(\texttt{self})$ is the type (i.e., the class-name) of the current object and $S(\Gamma(\texttt{self}))$ is the record type of its internal state.

[Write] $$\frac{C; S; \Gamma \vdash r : R}{C; S; \Gamma \vdash (\texttt{update } r) : \Gamma(\texttt{self})} \qquad C \vdash R \Subset S(\Gamma(\texttt{self}))$$

As in the previous rule this expression must be contained in a method. When by $(\texttt{update } r)$ we update some instance variables, we have to check that the fields specified belong to the instance variables of the current class ($R \Subset S(\Gamma(\texttt{self}))$; see appendix A.2 for $\Subset$); note that we need to specify only the instance variables we want to modify. The type of the expression is then the current class (which is recorded in $\Gamma(\texttt{self})$).

[OvAbst] $$\frac{C; S; \Gamma \vdash exp_1 : T_1 \quad \ldots \quad C; S; \Gamma \vdash exp_n : T_n}{C; S; \Gamma \vdash \ \&exp_1\&\ldots\&\ exp_n : \{T_1\texttt{ , }\ \ldots\ \texttt{ , }T_n\}} \{T_1\texttt{ , }\ \ldots\ \texttt{ , }T_n\} \in_C \mathbf{Types}$$

The type of an overloaded function is the set of the types of its branches (the T_i's are arrow types). Also, one has to check that the obtained type is well formed.

[OvAppl] $$\frac{C; S; \Gamma \vdash exp : \{D_i \to T_i\}_{i \in I} \qquad C; S; \Gamma \vdash exp_j : A_j \ \ (j=0..n)}{C; S; \Gamma \vdash \texttt{[}exp_0\ exp\ \ exp_1, \ldots, exp_n\ \texttt{]} : T_h}$$
$$\text{if } D_h = \min_{i \in I}\{D_i \mid C \vdash A_0 \times A_1 \times \ldots \times A_n \leq D_i\}.$$

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 $(exp_0\texttt{,}expr_1\texttt{,}\ldots\texttt{,}expr_n)$ and the selected branch is the branch h such that $D_h = \min_{i \in I}\{D_i \mid C \vdash A_0 \times A_1 \times \ldots \times A_n \leq D_i\}$. Note that if the set $\{D_i \mid C \vdash A_0 \times A_1 \times \ldots \times A_n \leq D_i, i \in I\}$ is not empty the the *min* exists thanks to the condition of *multiple inheritance* in the type of *exp*. If it is empty then the expression is not well-typed.

[Coerce] $$\frac{C; S; \Gamma \vdash exp : A}{C; S; \Gamma \vdash \texttt{coerce[}A'\texttt{](}exp\texttt{)} : A'} \qquad C \vdash A \leq A'$$

The construct $\texttt{coerce[}A'\texttt{](}exp\texttt{)}$ 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 $A \leq A'$. The same rule can be use for `super`, too.

Then, we have a special rule for multi-methods

[Multi] $$\frac{C; S; \Gamma \vdash exp_1 : T_1 \ldots C; S; \Gamma \vdash exp_n : T_n}{C; S; \Gamma \vdash \ \&exp_1\&\ldots\&\ exp_n : \#\{T_1\texttt{ , }\ \ldots\ \texttt{ , }T_n\}} \{T_1\texttt{ , }\ \ldots\ \texttt{ , }T_n\} \in_C \mathbf{Types}$$

Note that this rule and [OvAbst] assign two different types to the same expression $\&exp_1\&\ldots\&\ exp_n$; however this ambiguity is solved by the use of `#` in the interfaces. If that expression is to be used as an overloaded function (and thus it is applied to an argument) then it must be typed by [OvAbst].

The rule [MULTI], instead, is used to type multi-methods; it cannot be used otherwise, since there is no elimination rule for `#` (the `#` disappears thanks to the definition of "$\rightsquigarrow$"—see the rule [CLASS]—where the branches are "distributed" on the more general type of the message, which has no `#`).

Finally let us consider the typing of a class definition. We need to take a short detour. A class definition is always of the form:

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

where we use the notation $r{:}\,R$ 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 $\texttt{m}_i$ in the interface such that V_i is a raw type we must update its current type $\Gamma(\texttt{m}_i)$ in the following way: $\Gamma(\texttt{m}_i) := \Gamma(\texttt{m}_i) \cup \{A \to V_i\}$ (where we use the convention that $\Gamma(\texttt{m}_i) = \{\}$ if $\texttt{m}_i \notin dom(\Gamma)$). 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]). For example, if in the interface above $\texttt{m}_i{:}\texttt{\#}\{D \to U, D' \to T\}$ then we have to perform the following updating: $\Gamma(\texttt{m}_i) := \Gamma(\texttt{m}_i) \cup \{(A \times D) \to U, (A \times D') \to T\}$. 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 Γ gets: $\Gamma(\texttt{m}_i) := \Gamma(\texttt{m}_i) \cup \{A \rightsquigarrow V_i\}$.

Now we can 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
- $I \equiv$ `[[` $m_1 : V_1, \ldots, m_m : V_m$ `]]` the interface of the class
- $\Gamma' \equiv \Gamma[m_i \leftarrow \Gamma(m_i) \cup \{A \rightsquigarrow V_i\}]_{i=1..m}$ the environment Γ where the (overloaded) type of the messages is updated with the type of the new methods (branches) added by the class-definition

$$\frac{C; S; \Gamma \vdash r{:}\,R \qquad C'; S'; \Gamma'[\texttt{self} \leftarrow A] \vdash exp_j{:}\,V_j \ \ (j{=}1..k) \qquad C'; S'; \Gamma' \vdash p : T}{C; S; \Gamma \vdash \texttt{class}\ A\ \texttt{is}\ A_1,\ldots,A_n\ r{:}\,R\ \ m_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}m_k\texttt{=}exp_k\ \ I\ \texttt{in}\ p : T}$$

if $A \notin dom(S)$, for $i = 1..n$; $C \vdash R \le_{strict} S(A_i)$ and for $i = 1..k$; $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_{C'}$ **Types**

Let us examine, in more detail, the single parts of this rule. First we assure 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 ($C; S; \Gamma \vdash r: R$) and we verify that the type of the internal state of the class is compatible with (i.e. it is an extension of) the states of its ancestors ($C \vdash R \leq_{strict} S(A_i)$ for $i = 1..n$ [12]) (see appendix A.2 for the definition of $\leq_{strict}$, and [10] or [5] for motivations). Then we check that the defined messages possess well-formed overloaded types ($\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_{C'}$ **Types**), i.e. that they satisfy the conditions of covariance, multiple subtyping 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 object and in Γ' also the types of the new methods (for a possible mutual recursion). Finally we type the rest of the program where the class is declared. In order to implement the protection mechanisms we restore in the environment the old values for `self`.

Finally, the rule [Extend] (see appendix A.3) is a special case of the rule [Class] where there are no type constraints and no instance variables to check; we have just to check that the class in the `extend` expression has been already defined (i.e. $A \in dom(S)$). Note that because of the definition of $\cup$ and the condition of input type uniqueness, if m_i has already been defined for the class A then $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in$ **Types** if and only if $\{A \rightsquigarrow V_i\} \in \Gamma(m_i)$. In other terms if we redefine a method the new definition has to possess the same type as the old one.

3 The $\lambda\&$-calculus

In this section we briefly recall the main definitions of the $\lambda\&$-calculus, defined in [13]. What we present here is a slight variant of the calculus defined in [13]; this variant has been studied in [10]. For a more detailed discussion the reader may refer to [13,12,10].

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 lists, starting by an *empty* overloaded function denoted by ε, 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

[12] The ancestor must have been already declared, otherwise S is not defined

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

where $\rhd^+$ means "reduces in one or more steps to".

A set of arrow types $\{U_h \to T_h\}_{h \in H}$ is an overloaded type if and only if, for all U_i and U_j in $\{U_h\}_{h \in H}$, it satisfies these two conditions:

(1) if $U_i \leq U_j$ then $T_i \leq T_j$
(2) if U is maximal in $LB(U_i, U_j)$ then there exists a unique $h \in H$ such that $U_h = U$

These are exactly the conditions of 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 *late 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 $\qquad T ::= A \mid T \to T \mid \{T'_1 \to T''_1, \ldots, T'_n \to T''_n\}$

Subtyping
We define a partial order on the pretypes starting from a given order for the atomic types and we extend it to higher pretypes in the following way:

$$\frac{U_2 \leq U_1 \;\; T_1 \leq T_2}{U_1 \to T_1 \leq U_2 \to T_2} \qquad \frac{\forall i \in I, \exists j \in J \;\; U'_j \to T'_j \leq U''_i \to T''_i}{\{U'_j \to T'_j\}_{j \in J} \leq \{U''_i \to T''_i\}_{i \in I}}$$

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

$$M ::= x^T \mid \lambda x^T M \mid M \cdot M \mid \varepsilon \mid M \&^T M \mid M \bullet M$$

The type indexing the & is used for the selection of the branch in overloaded application and to type check overloaded functions.

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 an overloaded function. We do not need any type context Γ since the variables are indexed by their type.

[Taut] $\qquad x^T : T$

[$\to$Intro] $\qquad \dfrac{M : T}{\lambda x^U . M : U \to T}$

[$\to$Elim$_\leq$] $\qquad \dfrac{M : U \to T \;\; N : W \leq U}{M \cdot N : T}$

$$[\text{TAUT}_\varepsilon] \qquad \varepsilon : \{\}$$

$$[\{\}\text{INTRO}] \qquad \frac{M : W_1 \leq \{U_i \to T_i\}_{i\in I} \quad N : W_2 \leq U \to T}{(M \&^{\{U_i\to T_i\}_{i\in I}\oplus(U\to T)} N) : \{U_i \to T_i\}_{i\in I} \oplus (U \to T)}$$

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

whereevery pretype in the rules is a well-formed type (i.e. it belongs to **Types**) and $\{U_1 \to T_1, \ldots, U_n \to T_n\} \oplus (U \to T)$ has the following definition:

$$= \begin{cases} \{U_1 \to T_1, \ldots, U_{i-1} \to T_{i-1}, U_{i+1} \to T_{i+1}, \ldots U_n \to T_n, U \to T\} & \text{if } U = U_i \\ \{U_1 \to T_1, \ldots, \ldots U_n \to T_n, U \to T\} & \text{otherwise} \end{cases}$$

Reduction

The reduction $\rhd$ is the *compatible closure* of the following *notion of reduction* (for definitions see [4]):

β) $(\lambda x^T.M)N \rhd M[x^T := N]$

$\beta_\&$) If $N : U$ is closed and in normal form, and $U_j = min_{i=1..n}\{U_i | U \leq U_i\}$ then

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

In the $\lambda\&$-calculus there are infinitely many *fixed point combinators* (see [4]). For example we can define a Turing's fixed point combinator Θ_T for every type T. Recall that the Turing's fixed point combinator is characterized by the fact that $\Theta F \rhd^* F(\Theta F)$. Then define $\mathcal{S} \equiv (T \to T) \to T$. If $\mathcal{E}_\mathcal{S}$ is a closed term of type $\mathcal{S}$ then

$$A_T \equiv (\mathcal{E}_\mathcal{S} \&^{\{\{\}\to\mathcal{S},\{\{\}\to\mathcal{S}\}\to\mathcal{S}\}} \lambda x^{\{\{\}\to\mathcal{S}\}}.\lambda y^{T\to T}.y((x\bullet x)y)$$

has type $\{\{\} \to \mathcal{S}, \{\{\} \to \mathcal{S}\} \to \mathcal{S}\}$. Define $\Theta_T \equiv A_T \bullet A_T : \mathcal{S}$. Then for $F : T \to T$ we obtain

$$\Theta_T F \equiv (A\bullet A)F \rhd^* F((A\bullet A)F) \equiv F(\Theta_T F)$$

And Θ_T has type $(T \to T) \to T$ (for more details see chapter 3 of [10]).

One of the most compelling extensions of $\lambda\&$ is the one with explicit coercions ($\lambda\&$+coerce). Informally, an explicit coercion is a term that changes the type of its argument which, however, maintains its functionalities. This feature is crucial in $\lambda\&$, where types determine the computation: the capability to change types implies a greater control over the execution. In particular, it is possible to drive the selection towards a given branch by applying an explicit coercion to the argument of an overloaded function.

More formally, the extension of $\lambda\&$ by explicit coercions is obtained by adding to the terms $\mathbf{coerce}^T(M)$, by adding the following typing rule and

reduction:

$$[\textsc{Coerce}] \qquad \frac{\vdash M : S \leq T}{\vdash \mathbf{coerce}^T(M) : T}$$

$$(\text{coerce}) \qquad \mathbf{coerce}^T(M) \circ N \rhd M \circ N$$

where by $\circ$ we denote either $\bullet$ or $\cdot$.

For the $\lambda\&$-calculus and it extension with explicit coercions we proved, in [13,10], some fundamental theorems like the Church-Rosser property, the theorem of subject reduction, and the strong normalization of some relevant sub-calculi.

4 λ_object

In this section we define the meta-language λ_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 as if we had the λ-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 λ-calculus (e.g. as the one between the SECD machine and the λ_V: see [22]); 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. To this end we provide λ_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 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 correctness of the type-checker) which plays a key role, being λ_object intended for typed object-oriented languages.

The main decision in the definition of λ_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 λ_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 uniquely 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 λ_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 λ_object. Furthermore the operational semantics of λ_object will be simplified. Henceforth we will not distinguish among the terms "tag", "atomic type" and "class-name" since in λ_object they coincide.

To resume, in λ_object objects are "tagged terms" of the form in^A(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 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.

4.0.1 Pretypes

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

$$T ::= A \mid T \times T \mid T \to T \mid \{(A_1 \times \ldots \times A_{m_1}) \to T_1, \ldots, (B_1 \times \ldots \times B_{m_n}) \to T_n\}$$

where in the last production $n, m_i \geq 1$

4.0.2 Terms

Here we define the raw terms of the language, i.e. terms that have not been type checked yet. 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:

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

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

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 to it 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 introduces two constructors $in^A{:}T \to A$ and $out^A{:}A \to T$ which form a retraction pair from T to A.

4.0.3 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 dispatch, 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.

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

In the last case of the production above D is $(A_1 \times \ldots \times A_n)$

4.0.4 Operational Semantics

We define the *values* of λ_object, i.e. those terms that are considered as results; values are ranged over by G.

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

The operational semantics for λ_object is given by the reduction $\Rightarrow$; this reduction includes a type constraint system [13] C that is built along the reduction by the declarations (**let** $A \leq A_1 \ldots A_n$ **in** P) and that is used in the rule(s) for the selection of the branch. In the following, we use $\circ$ to denote either $\cdot$ or $\bullet$ and $\overline{D}(C)$ to denote the $\min_{i=1..n}\{D_i | C \vdash D \leq D_i\}$

Axioms

$(C\,,\ \pi_i(<G_1\,, G_2>))\ \Rightarrow\ (C\,,\ G_i)$ i=1,2

$(C\,,\ out^{A_1}(in^{A_2}(M)))\ \Rightarrow\ (C\,,\ M)$

$(C\,,\ out^{A_1}(\mathbf{coerce}^{A_2}(M)))\ \Rightarrow\ (C\,,\ out^{A_1}(M))$

$(C\,,\ out^{A_1}(\mathbf{super}^{A_2}(M)))\ \Rightarrow\ (C\,,\ out^{A_1}(M))$

$(C\,,\ \mu x.M)\ \Rightarrow\ (C\,,\ M[x := \mu x.M])$

$(C\,,\ (\lambda x.M)\cdot N)\ \Rightarrow\ (C\,,\ M[x := N])$

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

$$(C\ ,\ (M_1 \&^{\{D_1 \to T_1, \ldots, D_n \to T_n\}} M_2) \bullet G^D\ ) \ \Rightarrow\ (C\ ,\ M_1 \bullet G^D\ ) \qquad \text{if } D_n \neq \overline{D}(C)$$

$$(C\ ,\ (M_1 \&^{\{D_1 \to T_1, \ldots, D_n \to T_n\}} M_2) \bullet G^D\ ) \ \Rightarrow\ (C\ ,\ M_2 \cdot G^D\ ) \qquad \text{if } D_n = \overline{D}(C) \text{ and } G^D \not\equiv \mathbf{super}^D(M)$$

$$(C\ ,\ (M_1 \&^{\{D_1 \to T_1, \ldots, D_n \to T_n\}} M_2) \bullet G^D\ ) \ \Rightarrow\ (C\ ,\ M_2 \cdot M\ ) \qquad \text{if } D_n = \overline{D}(C) \text{ and } G^D \equiv \mathbf{super}^D(M)$$

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

$$(C\ ,\ \mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P\ ) \ \Rightarrow\ (C\ ,\ P\ )$$

Context Rules

$$\frac{(C\ ,\ M\ ) \Rightarrow (C\ ,\ M'\ )}{(C\ ,\ <M\ ,N>\ ) \Rightarrow (C\ ,\ <M'\ ,N>\ )} \qquad \frac{(C\ ,\ M\ ) \Rightarrow (C\ ,\ M'\ )}{(C\ ,\ <G\ ,M>\ ) \Rightarrow (C\ ,\ <G\ ,M'>\ )}$$

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

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

The semantics for pairs is the standard one. Three axioms and a rule describe the behavior of *out* and give it access to the internal state of an object. Functional application is implemented by call-by-name; anyway, this is not a necessary condition 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 (the term on the left) 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 7). 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 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 (**let** $A \leq A_1 \ldots A_n$ **in** P) modifies the type constraints in which to evaluate the body P, while (**let** A **hide** T **in** P) serves only to the type checker, and thus, operationally, is simply discarded.

4.0.5 Programs and type errors

The operational semantics above is given for untyped terms, thus its computations may genrate type errors. Now we define which terms are the programs of λ_object and when a reduction ends by a type error.

Definition 3 *A program in λ_object is a closed term P different from ε.*

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* if and only if there does not exist an 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 that is not a value and where the reduction is stuck. Let us call this subterm the *critical subterm* of P. For example, consider:

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

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 sixth context rule we try to reduce $(\mathbf{super}^A(M)) \cdot (N)$; again for the fifth context rule we try to reduce $(\mathbf{super}^A(M))$; but it is a value different from a λ-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 4 (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 $\overline{D}(C)$ (i.e. $\min_{i=1..n}\{D_i | C \vdash D \leq D_i\}$) 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 a wrong message has been sent to the object and in the latter that the conditions on multiple inheritance have 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. The complete definitions of this section are summarized in appendix B.

4.1.1 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 among them 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 and by the definition 2, 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

Definition 5
(i) $A \in_{C,S}$ **Types** *for each* $A \in dom(S)$
(ii) if $T_1, T_2 \in_{C,S}$ **Types** *then* $T_1 \to T_2 \in_{C,S}$ **Types** and $T_1 \times T_2 \in_{C,S}$ **Types**
(iii) 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 each maximal type* D *in* $LB_C(\{D_i, D_j\})$ *there exists* $h \in I$ *such that* $D_h = D$
(d) *if* $i \neq j$ *then* $D_i \neq D_j$
then $\{D_i \to T_i\}_{i \in I} \in_{C,S}$ **Types**

4.1.2 Type checking rules

The type checking rules 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; this is performed by the following rules

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

[Constraint] $$\frac{C \cup (A \leq A_i)_{1=1..n}, S \vdash P{:}\,T}{C, S \vdash \textbf{let}\ A \leq A_1, \ldots, A_n\ \textbf{in}\ P{:}\,T} \quad \text{if } C \vdash S(A) \leq S(A_i) \text{ and } A \text{ does not appear in } C$$

In the [NewType] rule we require that the representation type of a class is not another class; this is very reasonable, for the new atomic type would be completely equivalent to the one of its representation, but it would require a further *in* and *out* to access the internal state. In the last rule we require that A does not appear in any type constraint. In this way the ordering on atomic types is defined stepwise in the top-down sense. In this way the subtyping relation forms a dag. [14]

The rules for the terms of λ_object that already belongs to the $\lambda\&$ syntax are the same as the corresponding one in $\lambda\&$ (just add some C and S in the right places). The rules for the expressions that do not belong to the syntax of $\lambda\&$ are:

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

[14] Equivalently we could have defined C so that to satisfy this property.

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

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

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

Note that an atomic type A can be used in an expression like $\mathbf{coerce}^A$, $\mathbf{super}^A$ and so on, only if $A \in_{C,S}$ **Types**, i.e. it has been previously defined by a **let_hide** declaration.

4.2 Some results

Proposition 6 *Let $M : T$; if M is closed and in normal form then M is a value.*

Proof. The proof is obtained by induction on M. □

A consequence of this proposition is the following corollary which justifies the rules for the overloaded application in the operational semantics:

Corollary 7 *If a program is in normal form and it is typed by a (possibly unary) product of atomic types, then it is a tagged value.*

Recall that it is not possible to reduce inside a λ-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 (the $\beta_\&$-reduction) 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 as arguments by the type checker are tagged values.

Lemma 8 *(substitution lemma) Let $C, S \vdash M : U$, $C, S \vdash N : T'$ and $C \vdash T' \leq T$; then $C, S \vdash M[x^T := N] : U'$, where $C \vdash U' \leq U$*

Proof. By induction on M. The only difficult case is $M \equiv M_1 \bullet M_2$, whose proof follows the pattern of the corresponding case in the next theorem. □

Theorem 9 (Subject Reduction) *Let $C, S \vdash P : T$; if $(C, P) \stackrel{*}{\Rightarrow} (C', P')$ then $C', S \vdash P' : T'$ and $C' \vdash T' \leq T$.*

Proof. The proof consists in an induction on P where we use the substitution lemma above.

It suffices to prove the theorem for $\Rightarrow$; the thesis follows by a simple induction on the number of steps of the reduction. Thus, we proceed by induction on the structure of P. When P is a value then the thesis is trivially satisfied. When P is of the form (**let** ... **in** P') or of the form $\pi_i(M)$, then the proof is a straightforward use of the induction hypothesis. The remaining cases are (in the rest of the proof we omit C and S since they do not change):

$P \equiv out^A(M)$. Where $M : A' \leq A$. The only case of reduction is that $M \Rightarrow$

M' and $P' \equiv out^A(M')$; but from the induction hypothesis it follows that $M' : B \leq A' \leq A$; thus also P' is well-typed and possess the same type as P.

$P \equiv M_1 M_2$ where $M_1 : U \to T$ and $M_2 : W \leq U$. We have two subcases:

(i) $M_1 \Rightarrow M_1'$, then by induction hypothesis $M_1' : U' \to T'$ with $U \leq U'$ and $T' \leq T$. Since $W \leq U \leq U'$, then by rule $[\to\text{ELIM}_{(\leq)}]$ we obtain $M_1' M_2 : T' \leq T$

(ii) $M_1 \equiv \lambda x^U . M_3$ and $P \Rightarrow M_3[x := M_2]$, with $M_3 : T$. Thus, by Lemma 8, $M_3[x := M_2] : T'$ with $T' \leq T$.

$P \equiv M_1 \bullet M_2$ where $M_1 : \{D_i \to U_i\}_{i \in I}$ and $M_2 : D$.

Let $D_h = \min_{i \in I}\{D_i | D \leq D_i\}$. Thus $T = U_h$. We have three subcases:

(i) $M_1 \Rightarrow M_1'$ then by induction $M_1' : \{D_j' \to U_j'\}_{j \in J}$ with $\{D_j' \to U_j'\}_{j \in J} \leq \{D_i \to U_i\}_{i \in I}$ Let $D_k' = \min_{j \in J}\{D_j' | D \leq D_j'\}$. Thus $M_1' \bullet M_2 : U_k'$. Therefore we have to prove that $U_k' \leq U_h$

Since $\{D_j' \to U_j'\}_{j \in J} \leq \{D_i \to U_i\}_{i \in I}$, then for all $i \in I$ there exists $j \in J$ such that $D_j' \to U_j' \leq D_i \to U_i$. For $i = h$ we choose a certain $\tilde{h} \in J$ which satisfies this condition that is:

$$D_{\tilde{h}}' \to U_{\tilde{h}}' \leq D_h \to U_h \tag{1}$$

We now have the following inequalities:

$$D \leq D_h \tag{2}$$

by hypothesis, since $D_h = \min_{i \in I}\{D_i | D \leq D_i\}$;

$$D_h \leq D_{\tilde{h}}' \tag{3}$$

follows from (1);

$$D \leq D_{\tilde{h}}' \tag{4}$$

follows from (2) and (3);

$$U_{\tilde{h}}' \leq U_h \tag{5}$$

follows from (1);

$$D_k' \leq D_{\tilde{h}}' \tag{6}$$

by (4), since $D_{\tilde{h}}'$ belongs to a set with D_k' as least element;

$$U_k' \leq U_{\tilde{h}}' \tag{7}$$

follows from (6) and the covariance rule on $\{D_j' \to U_j'\}_{j \in J}$

Finally, by (5) and (7), one has that $U_k' \leq U_h$

(ii) $M_2 \Rightarrow M_2'$ then by induction hypothesis $M_2' : D'$ with $D' \leq D$. Let $D_k = \min_{i \in I}\{D_i | D' \leq D_i\}$. Thus $M_1 \bullet M_2' : U_k$. Since $D' \leq D \leq D_h$ then

$D_k \leq D_h$; thus, by the covariance rule in $\{D_i \to U_i\}_{i\in I}$, we obtain $U_k \leq U_h$.

(iii) $M_1 \equiv (N_1 \& N_2)$ and M_2 is a tagged value. Then we have three cases, that is $M \Rightarrow (N_1 \bullet M_2)$ (case $D_h \neq D_n$) or $M \Rightarrow (N_2 M_2)$ (case $D_h = D_n$ and M_2 different from super) or $M \Rightarrow (N_2 M_3)$ (case $D_h = D_n$ and $M_3 \equiv \mathbf{super}^D(M_2)$). According to the case it easy to use [{}ELIM] or [$\to$ELIM$_{(\leq)}$] or [$\to$ELIM$_{(\leq)}$] and [SUPER] to show that the terms have type U_h or smaller.

□

Proposition 10 *If $P \Rightarrow P'$ and P is closed then also P' is closed*

Proof. A simple induction on the rules of the operational semantics. □

Corollary 11 *Let P be a well-typed program. If $P \stackrel{*}{\Rightarrow} P'$ and P' is in normal form then P' is a value*

Proof. By theorem 9 P' is well-typed and by proposition 10 it is closed. The thesis follows from proposition 6. □

This corollary states that well-typed programs reduce to values, and thus do not produce type errors.

4.3 Encoding of record

In λ_object it is possible to encode the updatable records defined in [25]. They are constructed starting from an empty record value, denoted by $\langle\ \rangle$, and by two elementary operations:

- *Overwriting* $\langle r \leftarrow \ell_i = M\rangle$; if ℓ_i is not present in r, then it adds a field of label ℓ_i and value M to the record r; otherwise replaces the value of the field with label ℓ_i by the value M.
- *Extraction* $r.\ell_i$; extracts the value corresponding to the label ℓ_i, provided that a field having that label is present.

The encoding is defined as follows. Let $L_1, L_2, \ldots$ be an infinite list of *isolated* types[15], and introduce for each L_i a constant $\ell_i : L_i$. Then a record type is encoded in the following way:

$$\langle\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\rangle \quad \equiv \quad \{L_1 \to V_1, \ldots, L_n \to V_n\}$$

while the encoding of record values is given by:

$$\begin{array}{lcll} \langle\ \rangle & = & \varepsilon & \\ r.\ell_i & = & r \bullet \ell_i & \\ \langle r \leftarrow \ell_i = M\rangle & = & (r\ \&^I \lambda x^{L_i}.M) & \text{where } I \equiv (S \oplus \{L_i \to T\}) \\ & & & \text{if } r : S \text{ and } M : T \end{array}$$

Where $\oplus$ is the one defined in section 3. For the properties and the limits of this encoding see [10].

[15] A type T is *isolated* if for every type S, $S \leq T$ or $T \leq S$ implies $S = T$.

5 Translation

As we have already said, we will not give a direct semantics to the toy language. Instead we translate its programs into λ_object.

The key theorem of this section states that a well-typed program of the toy language is translated into a well typed term of λ_object ; this result validates the algorithm of type-checking we have given for the toy object-oriented language in section 2.8, since it assures that type errors can never occur during the computation of well-typed programs.

We split the definition of the interpreter in three parts: we first translate programs where methods are neither mutually recursive nor multi-methods; then by slight modifications we introduce also multi-methods and finally, in the third subsection, we introduce also recursive methods. In order to ease the presentation we only give the intuitive rules that constitute the translation. The detailed presentation of the formal translation is postponed to appendix C.

5.1 Simple methods without recursion

We first give the intuitive translation of all the object-oriented commands of the language:

- A message is an identifier of an overloaded function; thus it is translated in a variable possessing a (raw) overloaded type; i.e. $[\![m]\!] = m^{\{A_i \to T_i\}_{i \in I}}$ where $\{A_i | i \in I\}$ is the set of the classes where the message `m` has been defined, and the T_i's are the corresponding types appearing in the interfaces.
- Message passing is the application of an overloaded function; i.e. $[\![$ `[`exp_0 exp $exp_1, \ldots, exp_n$ `]`$]\!] = [\![exp]\!] \bullet [\![$`(`$exp_0, exp_1, \ldots, exp_n$`)`$]\!]$
- In the definition of a method, `self` represents the receiver of the message that 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. More exactly it defines $in^A(r)$ 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 $[\![$`(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 respectively translated into $\mathbf{super}^A([\![exp]\!])$ and $\mathbf{coerce}^A([\![exp]\!])$.
- The operation `extend` corresponds to adding a branch to an overloaded function. It has the following intuitive interpretation
  $[\![$ `extend` $A\ m = exp$ `[[...]] in` $exp'[\![=$

(**let** $m = (m\&\lambda self^A.[\![exp]\!])$ **in** $[\![exp']\!]$).

– Finally we have the most complex construct: the class definition. By a class definition one defines 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}` m `=` exp `[[` $m : T$ `]]` `in` p) is:

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

Of course the initial value 3 of x must be recorded during the translation so that this value could be used in the translation of `new(`A`)`.

5.2 With multi-methods

Let us now add multi-methods. Intuitively we have to change only three things:

– The type of a message must take into account also the multi-methods, thus $[\![\ \mathtt{m}\ ]\!] = m^{\{A_i \rightsquigarrow T_i\}_{i \in I}}$—note in the index the use of $\rightsquigarrow$ in the place of $\rightarrow$ used in the previous section—where again $\{A_i | i \in I\}$ is the set of the classes where the message `m` has been defined, and the T_i's are the corresponding types appearing in the interfaces.

– The method *msg=exp* is translated as before into $\lambda self^A.[\![exp]\!]$, if it is a normal method (A is the current class). 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 the lambda calculus, the multi-method is translated into

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

– In the translation of `extend` we have to translate the multi-methods in the same way as above.

5.3 With recursive methods

We now give the translation in the case that methods can be defined mutually recursively. The only thing we have to change is the interpretation of the methods, and then apply it both to the translation of the class definition and the one of `extend`. Intuitively without multi-methods if we have:

`extend` B `with`

$m_1 = exp_1$
$m_2 = exp_2$
$\vdots$
$m_n = exp_n$
`[[ ...]] in` *exp*

this is translated into

let $(m_1, m_2, \ldots, m_n) = \mu(m_1, m2, \ldots, m_n).$
$((m_1 \& \lambda self^B.[\![exp_1]\!]) , (m_2 \& \lambda self^B.[\![exp_2]\!]) , \ldots , (m_n \& \lambda self^B.[\![exp_n]\!]))$
in $[\![exp]\!]$

Of course we have to put the right types to the variables and the ampersands, and to deal with multi-methods. This is handled in appendix C.3

5.4 Correctness of the type-checking

We next prove that every well-typed program of the toy-language is translated in a well-typed term of λ_object. The semantics of the toy-language is given in terms of the translation we have just defined; also the notion of type error for the toy language comes from this translation: a program is type safe when its translation, if it stops, stops on a value. Thus, by the results of section 4.2 the translation of a well-typed program is type safe, which means that the type checker for λ_object is correct.

More formally the interpretation function $[\![]\!]$ of a program p will be parameterized by an environment of free variablesΓ an environment of the initial states I, and by the type of the current object A that will index the interpretation (i.e. $[\![p]\!]_{\Gamma\, I\, A}$: see appendix C). So that the theorem of correctness of the type system for the toy language is formulated as follows:

Theorem 12 *For every type constraint C, type environment Γ 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 $C; S \vdash [\![p]\!]_{\Gamma\, I\, \Gamma(self)}: T$*

6 λ_object and $\lambda\&$

In this section we show the exact correspondence between λ_object and $\lambda\&$ by presenting how the former can be encoded in $\lambda\&$+coerce. We are not able to translate the whole λ_object; we have to restrain our attention to those programs that do not contain **super**. This was quite predictable since the introduction of **super** had required the modification of the rule $\beta_{\&}$. First, we recall the encoding of surjective pairings in $\lambda\&$, which are similar to the encoding of record types. Distinguish two isolated types P_1 and P_2 together with two constants $\pi_1 : P_1$ and $\pi_2 : P_2$ then $(T_1 \times T_2) \equiv \{P_1 \rightarrow T_1, P_2 \rightarrow T_2\}$, $\pi_i(M) \equiv M \bullet \pi_i$, and $< M_1 , M_2 > \equiv (\varepsilon \& \lambda x^{P_1}.M_1 \& \lambda x^{P_2}.M_2)$ (for $x^{P_i} \notin FV(M_i)$). The rules of subtyping, typing and reduction are the *special cases* of the rules of $\lambda\&$ (and thus of $\lambda\&$+coerce).

We start by codifying the types of λ_object. Recall that in λ_object every atomic type is associated to a type used for its representation. This association is always relative to a program in which it is described. Thus given a well-typed program P we define

(i) The set of atomic types defined in the program P:

$$\mathcal{A}_P = \begin{cases} A \cup \mathcal{A}_{P'} & \text{if } P \equiv \textbf{let } A \textbf{ hide } T \textbf{ in } P' \\ \mathcal{A}_{P'} & \text{if } P \equiv \textbf{let } A \leq A_1, \ldots, A_n \textbf{ in } P' \\ \emptyset & \text{otherwise} \end{cases}$$

(ii) The set of type constraints generated in the program P:

$$\mathcal{C}_P = \begin{cases} (A \leq A_1) \cup \ldots \cup (A \leq A_n) \cup \mathcal{C}_{P'} & \text{if } P \equiv \textbf{let } A \leq A_1, \ldots, A_n \textbf{ in } P' \\ \mathcal{C}_{P'} & \text{if } P \equiv \textbf{let } A \textbf{ hide } T \textbf{ in } P' \\ \emptyset & \text{otherwise} \end{cases}$$

(iii) The function that for every atomic type A in $\mathcal{A}_P$ returns the representation type associated in P.

$$\mathcal{S}_P = \begin{cases} \mathcal{S}_{P'}[A \leftarrow T] & \text{if } P \equiv \textbf{let } A \textbf{ hide } T \textbf{ in } P' \\ \mathcal{S}_{P'} & \text{if } P \equiv \textbf{let } A \leq A_1, \ldots, A_n \textbf{ in } P' \\ \emptyset & \text{otherwise} \end{cases}$$

Then the translation of the types of λ_object relative to a program P is defined in the following way [16]

Definition 13 *For every well-typed program P, we translate a type $T \in_{\mathcal{C}_P, \mathcal{S}_P}$* **Types** *into the set of λ&-pretypes generated from the po-set of atomic types $(\mathcal{A}_P, \leq)$ where $\leq$ is the transitive and reflexive closure of $\mathcal{C}_P$. The translation is defined by induction on the structure of T:*

$$\begin{aligned} [\![A]\!] &= A \times [\![\mathcal{S}_P(A)]\!] \\ [\![A_1 \times A_2]\!] &= [\![A_1]\!] \times [\![A_2]\!] \\ [\![S \to T]\!] &= [\![S]\!] \to [\![T]\!] \\ [\![\{S_i \to T_i\}_{i \in I}]\!] &= \{[\![S_i]\!] \to [\![T_i]\!]\}_{i \in I} \end{aligned}$$

The definition above is well defined. To prove it associate to every $T \in_{\mathcal{C}_P, \mathcal{S}_P}$ **Types** the weight $w(T)$ defined as follows

$w(A) = n$ if A has been the n-th atomic type defined in the program P.
$w(S \to T) = w(S \times T) = \max\{w(S), w(T)\}$

[16] As a matter of facts there cannot be in λ_object only user defined atomic types; there must be at least one predefined atomic type `*` together with a constant `? : *` to start the definitions (see the implementation in CamlLight of λ_object described in [10]). This does not change the essence of what follows. Just imagine that also λ&contains `*` and `?` and that they are translated by the identity.

$w(\{S_i \to T_i\}_{i\in I}) = \max_{i\in I}\{w(S_i), w(T_i)\}$

Then it is easy to verify that, thanks to rules for typing and type good formation of λ_object, the translation of a type is always given in terms of the translations of types with a minor weight or with the same weight but a less deep syntax tree (remember that the translation is given w.r.t. a *well-typed* program P, and thus the definitions **let** ... **hide** ... cannot be circular)

The weight above is also used to prove the following proposition

Proposition 14 $\quad \mathcal{C}_P \vdash S \leq T \quad \Leftrightarrow \quad [\![S]\!] \leq [\![T]\!]$

Proof. Let $d(T)$ denote the depth of the syntax tree of T and associate to every subtyping judgment $S \leq T$ the pair $(w(S) + w(T), d(S) + d(T))$. Then the result follows from a straightforward induction on the lexicographical order of the pairs. The only non trivial case is when $S \leq T$ is $A_1 \leq A_2$:
($\Leftarrow$) If $A_1 \leq A_2$ in $\lambda\&$ then this must have been obtained by transitivity and reflexivity from $\mathcal{C}_P$. Thus $\mathcal{C}_P \vdash A_1 \leq A_2$
($\Rightarrow$) Vice versa if $\mathcal{C}_P \vdash A_1 \leq A_2$ then

$$\begin{aligned}[\![A_1]\!] \leq [\![A_2]\!] &\Leftrightarrow A_1 \times [\![\mathcal{S}_P(A_1)]\!] \leq A_2 \times [\![\mathcal{S}_P(A_2)]\!] \\ &\Leftrightarrow A_1 \leq A_2 \ \wedge \ [\![\mathcal{S}_P(A_1)]\!] \leq [\![\mathcal{S}_P(A_2)]\!]\end{aligned}$$

The first factor (i.e. $A_1 \leq A_2$) follows from $\mathcal{C}_P \vdash A_1 \leq A_2$ and definition 13. The second (i.e. $[\![\mathcal{S}_P(A_1)]\!] \leq [\![\mathcal{S}_P(A_2)]\!]$) follows from the induction hypothesis since the left component of the associated pair strictly decreases (indeed $w([\![\mathcal{S}_P(A_i)]\!]) \lneq w(A_i)$ for $i = 1, 2$). $\quad \Box$

This proposition has the following important corollary

Corollary 15 $U_j = min_{i\in I}\{U_i | U \leq U_i\} \Leftrightarrow [\![U_j]\!] = min_{i\in I}\{[\![U_i]\!] | [\![U]\!] \leq [\![U_i]\!]\}$

We can now define precisely the target calculus of the translation that we call $\mathbf{TARGET}_P$

Definition 16 *The target calculus* $\mathbf{TARGET}_P$ *of the translation relative to a well-typed program P has as raw terms the set of the $\lambda\&$+coerce terms constructed from a denumerable set of variables, the constants to encode pairings, and a constant c^A of each $A \in \mathcal{A}_P$. Its set of types is formed by $\mathcal{A}_P$ plus the pretypes that are in the image of the translation of definition 13 plus the types to encode pairings and to type fixpoint combinators. The subtyping relation is the one generated from $(\mathcal{A}_P, \leq)$ on the pretypes. The typing rules are those of $\lambda\&$+coerce.*

Thus $\mathbf{TARGET}_P$ is $\lambda\&$+coerce but without some types. In particular $A \times [\![T]\!]$ belongs to the types of $\mathbf{TARGET}_P$ if and only if **let** A **hide** T appears in P.

This is precisely stated by the following theorem

Theorem 17 *The translation of a well-formed type of λ_object satisfies the condition of type formation of $\lambda\&$(+coerce)*

Proof. The result follows nearly immediately from definition 16 and from proposition 14. Just note that the types added for fixpoint combinators do

not interfere with the condition **(c)** in definition 2. □

Note that the statement of the theorem would not hold if we had not restricted the types of $\mathbf{TARGET}_P$. This because $A_1 = A_2 \sqcap A_3$ does not imply $\mathcal{S}_P(A_1) = \mathcal{S}_P(A_2) \sqcap \mathcal{S}_P(A_3)$.

6.2 The encoding of the terms

We can now give the translation for the terms

Definition 18 *We give the translation relative to a well-typed program P, of a term of λ_object that does not contain* **super**.

$$
\begin{array}{lll}
[\![x^T]\!] & = x^{[\![T]\!]} & \\
[\![\mathrm{in}^A(M)]\!] & = \mathbf{coerce}^{[\![A]\!]}(\,(c^A\,,\,[\![M]\!])\,) & c^A \text{ is the constant of type } A \\
[\![\mathrm{out}^A(M)]\!] & = \pi_2([\![M]\!]) & \\
[\![\mathbf{coerce}^A(M)]\!] & = \mathbf{coerce}^{[\![A]\!]}([\![M]\!]) & \\
[\![\lambda x^T.M]\!] & = \lambda x^{[\![T]\!]}.[\![M]\!] & \\
[\![(M \&^T N)]\!] & = ([\![M]\!] \&^{[\![T]\!]} [\![N]\!]) & \\
[\![M \circ N]\!] & = [\![M]\!] \circ [\![N]\!] & \\
[\![< M\,, N >]\!] & = < [\![M]\!]\,, [\![N]\!] > & \\
[\![\pi_i(M)]\!] & = \pi'_i([\![M]\!]) & i = 1, 2 \\
[\![\mu x^T.M]\!] & = \Theta_{[\![T]\!]}(\lambda x^{[\![T]\!]}.[\![M]\!]) & \\
[\![\mathbf{let} \ldots \mathbf{in}\ P']\!] & = [\![P']\!] &
\end{array}
$$

where Θ is defined as in section 3

Strictly speaking we should have constructed $\mathcal{S}_P$ along the translation in the following way: $[\![\mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P']\!]_S = [\![P']\!]_{S[A \leftarrow T]}$; however, in the rest of this section, the declarations play a secondary role thus we prefer to deal with them more informally; consequently in the following we omit all the type constraint systems, understanding that they are all relative to the type system of a given program. Note also that we have distinguished two different pairings: one denoted by $(\ ,\ )$ with projections π_i, the other $<\ ,\ >$ with projections π'_i. The former is used to codify objects, the latter to encode the pairings of λ_object. We differentiated them so that they cannot interfere one with the other.

Theorem 19 *If $M : T$ then there exists T' such that $[\![M]\!] : [\![T']\!] \leq [\![T]\!]$*

Proof. The proof consists in a straightforward induction on the structure of the program and uses proposition 14. Just note that $[\![M]\!] : [\![T]\!]$ does not hold because of the definition of $[\![out^A(M)]\!]$ □

To conclude this section we have to prove the correctness of our translation, i.e. that if a program of λ_object reduces to a value then its translation reduces to the translation of the value. To prove it we need two technical lemmas. Let us denote by $N\!\downarrow$ the normal form of N.

Lemma 20 *Let N be a tagged value. If $N : D$ then $[\![N]\!]$ has a normal form and $[\![N]\!]\!\downarrow : [\![D]\!]$*

Proof. A trivial induction on the structure of tagged terms. Note that the coercion in the translation of in^A blocks the type. □

Lemma 21 (substitution) $[\![M[x^T := N]]\!] = [\![M]\!][x^{[\![T]\!]} := [\![N]\!]]$

Proof. A straightforward induction on M. Just note for the proof that Θ is a closed term. □

Theorem 22 *If* $M \Rightarrow N$ *then* $[\![M]\!] \rhd^+_{\beta \cup \beta_{\&} \cup (coerce)} [\![N]\!]$

Proof. By induction on the definition of $\Rightarrow$. It suffices to prove the theorem for the axioms of λ_object. The result then follows by a straightforward use of the induction hypothesis. We have six cases (the axioms for the declarations are trivially solved and we do not consider the axioms for **super**).

(i) $\pi_i(<G_1\,, G_2>) \;\Rightarrow\; G_i$ straightforward

(ii) $out^{A_1}(in^{A_2}(M)) \;\Rightarrow\; M$

$$\begin{aligned}
[\![out^{A_1}(in^{A_2}(M))]\!] &= \quad \pi_2([\![in^{A_2}(M)]\!]) \\
&= \quad \pi_2(\mathbf{coerce}^{[\![A_2]\!]}(\,(c^{A_2}\,, [\![M]\!])\,) \\
&\equiv \quad (\mathbf{coerce}^{[\![A_2]\!]}(\,(c^{A_2}\,, [\![M]\!])\,))\bullet\pi_2 \\
&\rhd_{(coerce)} \; \pi_2(\,(c^{A_2}\,, [\![M]\!])\,) \\
&\rhd^+_{\beta_{\&} \cup \beta} \quad [\![M]\!]
\end{aligned}$$

(iii) $out^{A_1}(\mathbf{coerce}^{A_2}(M)) \;\Rightarrow\; out^{A_1}(M)$

$$\begin{aligned}
[\![out^{A_1}(\mathbf{coerce}^{A_2}(M))]\!] &= \quad \pi_2([\![\mathbf{coerce}^{A_2}(M)]\!]) \\
&= \quad \pi_2(\mathbf{coerce}^{[\![A_2]\!]}([\![M]\!])) \\
&\rhd_{(coerce)} \; \pi_2([\![M]\!]) \\
&= \quad [\![out^{A_1}(M)]\!]
\end{aligned}$$

(iv) $\mu x^T.M \;\Rightarrow\; M[x^T := \mu x^T.M]$

$$\begin{aligned}
[\![\mu x^T.M]\!] &= \; \Theta_{[\![T]\!]}(\lambda x^{[\![T]\!]}.[\![M]\!]) \\
&\rhd^* \; (\lambda x^{[\![T]\!]}.[\![M]\!])(\Theta_{[\![T]\!]}(\lambda x^{[\![T]\!]}.[\![M]\!])) \\
&\rhd_\beta \; [\![M]\!][x^{[\![T]\!]} := [\![\mu x^T.M]\!]] \\
&= \; [\![M[x^T := \mu x^T.M]]\!] \qquad \text{by lemma 21}
\end{aligned}$$

(v) $(\lambda x^T.M)\cdot N \;\Rightarrow\; M[x^T := N]$

$$\begin{aligned}
[\![(\lambda x^T.M)\cdot N]\!] &= \; (\lambda x^{[\![T]\!]}.[\![M]\!])[\![N]\!] \\
&\rhd_\beta \; [\![M]\!][x^{[\![T]\!]} := [\![N]\!]] \\
&= \; [\![M[x^T := N]]\!] \qquad \text{by lemma 21}
\end{aligned}$$

(vi) $(M_1 \&^T M_2) \bullet G^D \;\Rightarrow\; M_i \circ G^D$ immediate from corollary 15 and lemma 20
□

7 Adding polymorphism to the toy language

Consider again the definitions of the classes `2DPoint` and `2DColorPoint` given in sections 2.2 and 2.3. In `2DPoint` a method for the message `erase` is defined; this method returns a copy of the receiver, with the field x set to zero. Thus, after the definition of `2DPoint`, and according to the typing rules of section 2.8, the message `erase` has the following type:

$$erase : \{2DPoint \to 2DPoint\}$$

In words, this type assignment means that the message `erase` can be sent to any object of a class smaller than or equal to `2DPoint`, and that the result has type `2DPoint`.

The method for `erase` can be used by the objects of class `2DColorPoint`. Therefore, the method is not redefined (overridden) in the definition of this class, but, instead, it is *inherited*. Since the definition of `erase` persists unchanged, then also its type does so. This has a nasty consequence: if we send the message `erase` to an object of class `2DColorPoint` then, for the type-checker, the result is an object of type `2DPoint`, rather than `2DColorPoint`. Therefore the following expression

```
let aColorPoint = [ new(2DColorPoint) erase ]
   in [ aColorPoint isWhite ]
```

would be rejected by the type-checker, since this one would assign to `aColorPoint` the type `2DPoint`. Note that this problem is already present in $\lambda\&$ and, thus, in λ_object: consider an overloaded function *copy* whose definition is of the form $(\ldots \&\ \lambda self^{C_1}.\, self\ \ \& \ldots)$. Thus

$$copy\colon \{..., C_1 \to C_1, ...\}$$

Let C_2 be a subtype of C_1 and suppose that the application of *copy* to a term of type C_2 selects the branch defined for C_1. Then the result of the application has type C_1, rather than C_2, as it would be natural.

This problem is well known in the field of type theoretic research on object-oriented programming, where it is designated as the "loss of information problem". It was already pointed out for the record-based models in Cardelli's seminal paper [9]. And it is also present in some commercial object-oriented languages (e.g. O_2: see [3]). This problem was tackled—and overcome— for the record-based approach, by many authors, notably [8,25,26,23,15]. In all these propositions the solution is to pass to formalism with second order types. In [11] we adopted the same solution for the overloading-based model. Thus we defined a formalism—called $F^{\&}_{\leq}$—to model second order "ad hoc" poly-

morphism [17] . Very briefly, the rough idea is to define a type system that types the previous function *copy* in the following way:

$$copy\colon \{..., \forall X \leq C_1.X \to X, ...\}$$

In words, it means that *copy* is no longer a message that, if applied to an argument of type smaller than or equal to C_1, returns a result of type C_1, but, instead, it is a function that, if applied to an argument of type X, smaller than or equal to C_1, returns a result of the same type, X, of the argument.

The same technique can be applied to `erase`. Indeed, to transfer the theoretic results of $F^{\&}_{\leq}$ (foremost, the absence of loss of information) to object-oriented programming we do not have to redefine the toy object-oriented language from scratch: few modifications to the toy language of section 2 suffice. Informally, from the point of view of our toy object-oriented language, the gain of considering a second order system is embodied by the fact that we can use the reserved keyword `Mytype` in the class interfaces. This keyword denotes the type of the receiver of a message (we borrowed `Mytype` from [6]; examples of other keywords with the same use are "`like current`" [19] and `myclass` [7]). Note that by inheritance the type of the receiver of a message can be smaller than the class(-name) for which the method has been defined: in our previous example the receiver of the method defined for `2DPoint` was a `2DColorPoint`. The use of this keyword can be shown by slightly modifying the definitions of the classes `2DPoint` and `2DColorPoint` of sections 2.2 and 2.3.

```
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: Mytype;
    move: (Int x Int) -> 2DPoint
  ]]



class 2DColorPoint is 2DPoint
  {
    x:Int = 0;
    y:Int = 0;
    c:String = "black"
  }
```

[17] According to the classification of [24], the mechanism of overloading is often referred as *"ad hoc" polymorphism*.

```
  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) -> Mytype
]]
```

Remark that we have modified only the interfaces, using in two places the keyword `Mytype`. Recall that in the original toy language, the type system assigned to the term `[new(2DColorPoint) erase]` the type `2DPoint`. Now, the keyword `Mytype` in the interface says that the type returned by sending `erase` is the same as the type of the receiver. In the case of `[new(2DColorPoint) erase]`, therefore, the type inferred is `2DColorPoint`.

Note also that, in the interface of `2DColorPoint`, the message `move` returns `Mytype` instead of `2DPoint`. The other way round is not allowed, i.e. it is not possible to replace `Mytype` by a class-name. For example the following definition

```
extend 2DColorPoint
   erase = new(2DColorPoint)
[[ erase: 2DColorPoint ]]
```

would not be well-typed since the method `erase` in `2DPoint` returned `Mytype`. Indeed, the identifier—more precisely, the type variable—`Mytype` that occurs in the interface of `2DPoint` may assume any type smaller than or equal to `2DPoint`, and thus, in particular, also a type smaller than `2DColorPoint`. In that case the covariance condition would not be respected [18] .

Let see how the intuitive interpretation of these constructs is formally reflected in the type discipline of our toy language.

We just consider a restricted version of the toy language, without multiple dispatching and in which messages (overloaded functions) are not first class, i.e. they can be neither the argument nor the result of a function. We impose this restriction in order to maintain to a minimum the modifications we have to make to the type system. However, on the basis of the results in [11], the is no conceptual problem to include also these features (see [10]).

We modify the language by adding the following productions

$$\textbf{Terms} \qquad v ::= \texttt{new(Mytype)}$$

$$\textbf{Raw Types}\ T ::= \texttt{Mytype}$$

We do not detail the modifications to make to the definitions of the types and of the terms in order to exclude first class messages. They are very simple and the reader can easily find them out.

[18] Of course in the previous example it would have been more reasonable that `move` in `2DPoint` returned `(Int x Int) -> Mytype` rather than `(Int x Int) -> 2DPoint`.

Recall that `Mytype` denotes the type of the receiver of the message, and that, in the body of a method, the receiver is denoted by `self`. Thus `self` is of type `Mytype`, and we have to modify the type-checking rule [TAUT] in order to take into account the new type of `self`; we split the rule [TAUT] (see appendix A.3) in two rules

$$[\textsc{TautVar}] \qquad C; S; \Gamma \vdash x : \Gamma(x) \qquad \text{for } x \in Vars$$

$$[\textsc{TautSelf}] \qquad C; S; \Gamma \vdash \texttt{self} : \texttt{Mytype}$$

As before, we use $\Gamma(\texttt{self})$ to record the current class (see the rule [CLASS] below).

Then we must modify the rule [WRITE] since now the update of the internal state returns a value of type `Mytype`.

$$[\textsc{Write}] \qquad \frac{C; S; \Gamma \vdash r\colon R}{C; S; \Gamma \vdash \texttt{(update } r\texttt{)\colon Mytype}} \qquad \text{if } C \vdash R \Subset S(\Gamma(\texttt{self}))$$

We must also extend the rule [NEW] to include the case `new(Mytype):Mytype`.

Consider the typing of the body of a method: an expression of type `Mytype` can appear inside this body. We know that `Mytype` is the type of the receiver, and, thus, it will be instantiated by a type smaller than or equal to the type of the current class (the method at issue will be inherited only by subclasses of the current class). Thus, in the typing of a method we have to record that `Mytype` is smaller than the current class, i.e. that $\texttt{Mytype} \leq \Gamma(\texttt{self})$. This constraint is, for example, used to type expressions like `[`v *message*`]` or `super[`A`](`v`)` when v : `Mytype`. Thus, we have to replace the old rule [CLASS] by the following one

$$\frac{\begin{array}{c} C; S; \Gamma \vdash r\colon R \\ C' \cup \{\texttt{Mytype} \leq \Gamma(\texttt{self})\}; S'; \Gamma'[\texttt{self} \leftarrow A] \vdash exp_j\colon V_j \quad (j=1..m) \\ C'; S'; \Gamma' \vdash p : T \end{array}}{C; S; \Gamma \vdash \texttt{class } A \texttt{ is } A_1\texttt{,...,}A_n\ r\colon R\ \ m_1\texttt{=}exp_1\texttt{;...;}m_m\texttt{=}exp_m\ \ I \texttt{ in } p : T}$$

if $A \notin dom(S)$, for $i = 1..n$ $C \vdash R \leq_{strict} S(A_i)$ and for $i = 1..m$ $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_{C'}$ **Types**

Where C', S' and Γ' are defined as in section 2.8. Similar modifications are required for the rule [EXTEND].

Finally during message passing we have to instantiate `Mytype`, with the type of the receiver

$$[\textsc{OvAppl}] \qquad \frac{C; S; \Gamma \vdash exp_1\colon \{A_i \to T_i\}_{i \in I} \qquad C; S; \Gamma \vdash exp_2\colon A}{C; S; \Gamma \vdash \texttt{[}exp_2\ \ exp_1\ \texttt{]}\colon T_h[\texttt{Mytype}\colon = A]} \qquad A_h = \min_{i \in I}\{A_i \mid C \vdash A \leq A_i\}$$

Note that since overloaded functions are not first class then the type T_h does not contain overloaded types. This avoids possible name-clashes in the substitution $[\texttt{Mytype}\colon = A]$. [19]

[19] However there is no conceptual difficulty to allow also first class overloaded functions. The problem is that it would require us to change our notation for overloaded types. Indeed, to avoid name clashes in the rule [OVAPPL] every overloaded type

In this framework the overloaded type

$$\{A_1 \to T_1, \ldots, A_n \to T_n\} \tag{8}$$

has a completely new meaning, which is quite different from the one it had in the case without polymorphism. Consider the branch corresponding to the type $A_i \to T_i$. Note, first, that `Mytype` can appear "free" in T_i. In the non-polymorphic case $A_i \to T_i$ meant that the branch accepted a receiver of type smaller than A_i and returned the type T_i. With the introduction of the polymorphism—i.e. of the type variable `Mytype`—this type means that the corresponding branch accepts a receiver of type A smaller than or equal to A_i and returns a result of type $T_i[\texttt{Mytype} := A]$. Thus, in order to make explicit the exact functionality hidden under the notation of the type (8) the notation to use should be

$$\{\forall \texttt{Mytype} \leq A_1.\texttt{Mytype} \to T_1, \ldots, \forall \texttt{Mytype} \leq A_n.\texttt{Mytype} \to T_n\}$$

or alternatively (using the notation of [11], where the type of the parameter is quantified externally to the overloaded type)

$$\forall \texttt{Mytype}\{A_1.\texttt{Mytype} \to T_1, \ldots, A_n.\texttt{Mytype} \to T_n\}$$

In both cases `Mytype` is a type variable bound by a quantifier that delimits the range of the overloaded type.

To end this section we have to give the rules of good type formation for this new system. The conditions for multiple inheritance and input type uniqueness (definition 2, **(c)** and **(d)**) persist unchanged. What changes is the covariance condition, since it has to take into account that `Mytype` may occur in the interfaces. More precisely, it must allow, when redefining a method given for a class A, to replace a covariant occurrence of A by `Mytype`: this is what we have done, say, at the beginning of this section, in the definition of `2DColorPoint` where for `move` we replaced `2DPoint` by `Mytype`.

Thus one has to modify the condition **(b)** in the definition 2 of type good formation in the following way

(b) if $C \vdash D_i \leq D_j$ then $C \cup \{\texttt{Mytype} \leq D_i\} \vdash T_i \leq T_j$

The formal correctness of this condition can be found in [11,10].

8 Conclusion and future work

As we already said in the introduction, this paper constitutes the companion of [13]. In that paper we defined a kernel calculus, $\lambda\&$, to study the formal

should bear a type variable along with it: this type variable is the one denoting the type of the receiver. An example of how it can be done can be found in [10].

properties of overloading and late binding. In this paper we defined a meta-language, λ_object, to formalize the correspondence between overloading with late binding, and the object-oriented programming. The two papers are in a sense, mutually recursive: to establish $\lambda\&$ we always kept in mind the formalization of object-oriented programming; to establish λ_object we were driven by the formalization given to the calculus.

An example has been used to show how it is possible, via λ_object, to establish a formal correspondence between the overloading-based model and the object-oriented programming: we defined a toy functional object-oriented language, we translated it into λ_object and we used this translation to prove properties of the toy language.

In the last section we hinted at the formalization of overloaded types in a second order framework and we showed how to apply the theoretical results of such a study to the definition and the typing of object-oriented languages. The typing rules written in that section are not pulled out of thin air, nor do they rely on just intuition, but they are based on the formal analysis developed in [11] (see for more details [10]). In a sense, with the second order system, we followed the same path that we used for the case of simple typing: driven by our intuition of object-oriented languages we first defined a formal calculus—$\lambda\&$ for simple typing and $F^{\&}_{\leq}$ for polymorphic types—, then we applied the results coming from the formalization to the practice of object-oriented programming. The final step consists of giving a formal proof of the correspondence between the theory and its application, by defining a meta-language in which to interpret and translate, object-oriented languages. This is what we did, with this paper, for the case of simple typing; this is what is still missing for the second order case. The work developed here is the base for such a definition. Indeed, as the passage to the second order has required just a few changes to the definition of the toy language, in the same way the meta-language for polymorphic object-oriented languages should be obtained by introducing a few modifications to λ_object. The passage, however, cannot be so smooth as it was for the toy language: in that case we modified a particular language, while in the new case we have to define a meta-language that must be valid for a wide class of (polymorphic) object-oriented languages. Many problems must first be solved, foremost the introduction of *implicit* parametric polymorphism, which we are currently working on. [20] Also, some mechanisms to deal with the dynamic definitions of new classes (see the discussion in Appendix C.1) should be introduced.

[20] Indeed, the polymorphism we used in our toy language is somewhat an half-way implicit-explicit polymorphism, since we have *a* type variable in the language —i.e. Mytype— but types are not explicitly passed to functions. For a wider discussion see chapter 11 of [10]

Acknowledgments

Many ideas of this work come from several discussions with Luca Cardelli, Giorgio Ghelli, Giuseppe Longo, Eugenio Moggi and Benjamin Pierce. I am especially grateful to Allyn Dimock, Maribel Fernández and Benjamin Pierce for their precise as well as useful comments on an early version of this paper. Finally, I want also to thank the referees of TCS for their useful comments. In particular one of them, whose report was the most accurate and detailed report I have ever seen.

References

[1] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of the 19th VLDB Conference*, Dublin, 1993.

[2] Apple Computer Inc., Eastern Research and Technology. *Dylan: an object-oriented dynamic language*, April 1992.

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

[4] H.P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*. North-Holland, 1984. Revised edition.

[5] K. B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *20th Ann. ACM Symp. on Principles of Programming Languages*. ACM Press, 1993.

[6] K.B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[7] P.S. Canning, W.R. Cook, W.L. Hill, and W.G. Orthoff. Interfaces for strongly-typed object-oriented programming. In *OOPSLA '89*, New Orleans, October 1989.

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

[9] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. A previous version can be found in Semantics of Data Types, LNCS 173, 51-67, Springer-Verlag, 1984.

[10] G. Castagna. *Overloading, subtyping and late binding: functional foundation of object-oriented programming*. PhD thesis, Université Paris 7, January 1994. Appeared as LIENS technical report.

[11] G. Castagna. Integration of parametric and "ad hoc" second order polymorphism in a calculus with subtyping. *Formal Aspects of Computing*, 1995. Springer Verlag. To appear. Part of this work appeared in the *Proc. of the 4th International Workshop on Database Programming Languages*, Wordkshop in Computing series, Springer Verlag.

[12] G. Castagna, G. Ghelli, and G. Longo. A semantics for $\lambda\&$-*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 Lecture Notes in Computer Science, pages 107–123, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[13] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995. A preliminary version has been presented at the *1992 ACM Conference on LISP and Functional Programming*, San Francisco, June 1992.

[14] 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.

[15] P. L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and the type checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1), 1992.

[16] L.G. DeMichiel and R.P. Gabriel. Common lisp object system overview. In Bézivin, Hullot, Cointe, and Lieberman, editors, *Proc. of ECOOP '87 European Conference on Object-Oriented Programming*, number 276 in LNCS, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[17] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Mass., 1983.

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

[19] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[20] NeXT Computer Inc. *NeXTstep-concepts. Chapter 3: Object-Oriented Programming and Objective-C*, 2.0 edition, 1991.

[21] L.J. Pinson and R.S. Wiener. *Objective-C: Object-Oriented Programming Techniques*. Addison-Wesley, 1992.

[22] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1, 1975.

[23] D. Rémy. Typechecking records and variants in a natural extension of ML. In *16th Ann. ACM Symp. on Principles of Programming Languages*, 1989.

[24] C. Strachey. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[25] Mitchell Wand. Complete type inference for simple objects. In *2nd Ann. Symp. on Logic in Computer Science*, 1987.

[26] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *3rd Ann. Symp. on Logic in Computer Science*, 1988.

A Specification of the toy language

A.1 Terms

$$r ::= \texttt{\{}\ell_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}\ell_n\texttt{=}exp_n\texttt{\}}$$

$$\begin{aligned}
exp ::= {} & x \\
| \ & \texttt{fn(}x_1 : T_1, ..., x_n : T_n\texttt{) =>}\ exp \\
| \ & exp_1\texttt{(}exp_2\texttt{)} \\
| \ & \texttt{(}exp, \ldots, exp\texttt{)} \\
| \ & \texttt{fst(}exp\texttt{)} \mid \texttt{snd(}exp\texttt{)} \\
| \ & \texttt{let}\ x\texttt{:}T\ \texttt{=}\ exp\ \texttt{in}\ exp \\
| \ & \texttt{extend}\ classname \\
& \qquad (message\ \texttt{=}\ method\texttt{;})^+ \\
& \qquad interface \\
& \texttt{in}\ exp \\
| \ & \texttt{new(}classname\texttt{)} \\
| \ & \texttt{self} \\
| \ & \texttt{(self.}\ell\texttt{)} \\
| \ & \texttt{(update}\ r\texttt{)} \\
| \ & \texttt{super[}A\texttt{](}exp\texttt{)} \\
| \ & \texttt{coerce[}A\texttt{](}exp\texttt{)} \\
| \ & \texttt{\&}\ \texttt{fn(x}_1\texttt{:}A_1, ..., \texttt{x}_{n_1} : A_{n_1}\texttt{) =>}\ exp_1 \\
& \texttt{\&}\ \texttt{fn(x}_1\texttt{:}A_1, ..., \texttt{x}_{n_2} : A_{n_2}\texttt{) =>}\ exp_2 \\
& \qquad\qquad \vdots \\
& \texttt{\&}\ \texttt{fn(x}_1 : A_1\texttt{,}\ldots\texttt{,}x_{n_m} : A_{n_m}\texttt{) =>}\ exp_m \qquad (m{\geq}1) \\
| \ & \texttt{[}\ exp_0 exp\ exp_1 \ldots exp_n\ \texttt{]} \qquad (n{\geq}0)
\end{aligned}$$

$$\begin{aligned}
p ::= {} & exp \\
| \ & \texttt{class}\ classname\ [\ \texttt{is}\ classname\ (\ ,\ classname\ )^*] \\
& \qquad instanceVariables \\
& \qquad (message\ \texttt{=}\ method\texttt{;})^* \\
& \qquad interface \\
& \texttt{in}\ p
\end{aligned}$$

$$method ::= exp$$

$$message ::= x$$

$$interface ::= \texttt{[[}message : V\texttt{;}...\texttt{;}message : V\texttt{]]}$$

$$instanceVariables ::= \texttt{\{}\ell_1 : T_1\texttt{=}exp_1; \ldots; \ell_n : T_n\texttt{=}exp_n\texttt{\}}$$

A.2 Subtyping

$$C \cup (A_1 \leq A_2) \vdash A_1 \leq A_2$$

$$\frac{C \vdash T_2 \leq T_1 \quad C \vdash U_1 \leq U_2}{C \vdash T_1 \to U_1 \leq T_2 \to U_2}$$

$$\frac{C \vdash U_1 \leq T_1 \ \ldots C \vdash U_n \leq T_n}{C \vdash \ (U_1 \times \ldots \times U_n) \leq (T_1 \times \ldots \times T_n)}$$

$$\frac{\text{for all } i \in I, \text{ there exists } j \in J \text{ such that } C \vdash D''_i \leq D'_j \text{ and } C \vdash U'_j \leq U''_i}{C \vdash \{D'_j \to U'_j\}_{j \in J} \leq \{D''_i \to U''_i\}_{i \in I}}$$

$$\frac{C \vdash U_1 \leq T_1 \ \ldots C \vdash U_k \leq T_k}{C \vdash \langle\!\langle \ell_1 \colon U_1; \ldots; \ell_k \colon U_k; \ldots; \ell_{k+j} \colon U_{k+j} \rangle\!\rangle \leq \langle\!\langle \ell_1 \colon T_1; \ldots; \ell_k \colon T_k \rangle\!\rangle}$$

The (pre)order for all types is given by the reflexive and transitive closure of the rules above.

A.2.1 Auxiliary Notation

$$C \vdash \langle\!\langle \ell_1 \colon T_1; \ldots; \ell_k \colon T_k; \ldots; \ell_{k+j} \colon T_{k+j} \rangle\!\rangle \leq_{strict} \langle\!\langle \ell_1 \colon T_1; \ldots; \ell_k \colon T_k \rangle\!\rangle$$

$$\frac{C \vdash U_1 \leq T_1 \ \ldots C \vdash U_k \leq T_k}{C \vdash \langle\!\langle \ell_1 \colon U_1; \ldots; \ell_k \colon U_k \rangle\!\rangle \Subset \langle\!\langle \ell_1 \colon T_1; \ldots; \ell_k \colon T_k; \ldots; \ell_{k+j} \colon T_{k+j} \rangle\!\rangle}$$

A.3 Typing Rules

Let

$$\Gamma \colon (Vars \cup \{\texttt{self}\}) \to \textbf{Types}$$

$$S \colon \textbf{AtomicTypes} \to \textbf{RecordTypes}$$

Then we have the following typing rules:

[TAUT] $$C; S; \Gamma \vdash x : \Gamma(x) \qquad x \in (Vars \cup \{\texttt{self}\})$$

[FUNCT] $$\frac{C; S; \Gamma[x \leftarrow T] \vdash exp \colon U}{C; S; \Gamma \vdash \texttt{fn(}x\texttt{:}T\texttt{) =>}\ exp : T \to U} \qquad \text{if } T \in_C \textbf{Types}$$

[APPL] $$\frac{C; S; \Gamma \vdash exp_1 \colon T \to U \qquad C; S; \Gamma \vdash exp_2 \colon W}{C; S; \Gamma \vdash exp_1\texttt{(}exp_2\texttt{)} : U} \qquad \text{if } C \vdash W \leq T$$

[Prod] $$\frac{C; S; \Gamma \vdash exp_1 : T_1 \quad \ldots \quad C; S; \Gamma \vdash exp_n : T_n}{C; S; \Gamma \vdash \texttt{(}exp_1\texttt{,}\ldots\texttt{,}exp_n\texttt{)} : (T_1 \times \ldots \times T_n)}$$

[Record] $$\frac{C; S; \Gamma \vdash exp_1 : T_1 \quad \ldots \quad C; S; \Gamma \vdash exp_n : T_n}{C; S; \Gamma \vdash \texttt{\{}\ell_1 = exp_1\texttt{;}\ldots\texttt{;}\ell_n = exp_n\texttt{\}} : \langle\!\langle \ell_1 : T_1; \ldots; \ell_n : T_n \rangle\!\rangle}$$

[Let] $$\frac{C; S; \Gamma \vdash exp' : W \qquad C; S; \Gamma[x \leftarrow T] \vdash exp : U}{C; S; \Gamma \vdash \texttt{let}\ x : T\ \texttt{=}\ exp'\ \texttt{in}\ exp : U} \quad \text{if } C \vdash W \leq T$$

[New] $$C; S; \Gamma \vdash \texttt{new(}A\texttt{)} : A \quad \text{if } A \in dom(S)$$

[Read] $$C; S; \Gamma \vdash \texttt{self}.\ell : T \quad \text{if } S(\Gamma(\texttt{self})) = \langle\!\langle \ldots \ell : T \ldots \rangle\!\rangle$$

[Write] $$\frac{C; S; \Gamma \vdash r : R}{C; S; \Gamma \vdash \texttt{(update}\ r\texttt{)} : \Gamma(\texttt{self})} \quad \text{if } C \vdash R \Subset S(\Gamma(\texttt{self}))$$

[OvAbst] $$\frac{C; S; \Gamma \vdash exp_1 : T_1 \ldots C; S; \Gamma \vdash exp_n : T_n}{C; S; \Gamma \vdash\ \texttt{\&}exp_1\texttt{\&}\ldots\texttt{\&}\ exp_n : \{T_1\texttt{,}\ \ldots\ \texttt{,}T_n\}} \{T_1\texttt{,}\ \ldots\ \texttt{,}T_n\} \in_C \textbf{Types}$$

[OvAppl] $$\frac{C; S; \Gamma \vdash exp : \{D_i \to T_i\}_{i \in I} \qquad C; S; \Gamma \vdash exp_j : A_j \ \ (j=0..n)}{C; S; \Gamma \vdash\ \texttt{[}exp_0\ exp\ \ exp_1, \ldots, exp_n\ \texttt{]} : T_h}$$
if $D_h = \min_{i \in I}\{D_i \mid C \vdash A_0 \texttt{x} A_1 \texttt{x} \ldots A_n \leq D_i\}$.

[Coerce] $$\frac{C; S; \Gamma \vdash exp : A}{C; S; \Gamma \vdash \texttt{coerce[}A'\texttt{](}exp\texttt{)} : A'} \quad \text{if } C \vdash A \leq A'$$

[Super] $$\frac{C; S; \Gamma \vdash exp : A}{C; S; \Gamma \vdash \texttt{super[}A'\texttt{](}exp\texttt{)} : A'} \quad \text{if } C \vdash A \leq A'$$

[Multi] $$\frac{C; S; \Gamma \vdash exp_1 : T_1 \ldots C; S; \Gamma \vdash exp_n : T_n}{C; S; \Gamma \vdash\ \texttt{\&}exp_1\texttt{\&}\ldots\texttt{\&}\ exp_n : \texttt{\#}\{T_1\texttt{,}\ \ldots\ \texttt{,}T_n\}} \ \{T_1, \ldots, T_n\} \in_C \textbf{Types}$$

[Extend] $$\frac{C; S; \Gamma'[\texttt{self} \leftarrow A] \vdash exp_j : V_j \ \ (j=1..k) \qquad C; S; \Gamma' \vdash exp : T}{C; S; \Gamma \vdash \texttt{extend}\ A\ \ m_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}m_k\texttt{=}exp_k\ \ \texttt{[[}m_1 : V_1\texttt{,}\ldots\texttt{,}m_k : V_k\texttt{]]}\ \texttt{in}\ exp : T}$$
$A \in dom(S)$ and for $i = 1..k$ $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_C \textbf{Types}$

[Class] $$\frac{C; S; \Gamma \vdash r : R \qquad C'; S'; \Gamma'[\texttt{self} \leftarrow A] \vdash exp_j : V_j \ \ (j=1..k) \qquad C'; S'; \Gamma' \vdash p : T}{C; S; \Gamma \vdash \texttt{class}\ A\ \texttt{is}\ \ A_1\texttt{,}\ldots\texttt{,}A_n\ r : R\ \ m_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}m_k\texttt{=}exp_k\ \ I\ \ \texttt{in}\ \ p : T}$$
if $A \notin dom(S)$, for $i = 1..n$ $C \vdash R \leq_{strict} S(A_i)$ and for $i = 1..k$ $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_{C'} \textbf{Types}$

Where:

- $A \rightsquigarrow V = \begin{cases} \{(A \times D_i) \to U_i\}_{i \in I} & \text{if } V \equiv \texttt{\#}\{D_i \to U_i\}_{i \in I} \\ \{A \to V\} & \text{otherwise} \end{cases}$
- $S' \equiv S[A \leftarrow R]$
- $C' \equiv C \cup (\bigcup_{i=1..n} A \leq A_i)$
- $I \equiv\ \texttt{[[}m_1 : V_1\texttt{,}\ldots\texttt{,}m_m : V_m\ \texttt{]]}$
- $\Gamma' \equiv \Gamma[m_i \leftarrow \Gamma(m_i) \cup \{A \rightsquigarrow V_i\}]_{i=1..m}$

B Type system of λ_object

B.1 Types

(i) $A \in_{C,S}$ **Types** for each $A \in dom(S)$
(ii) if $T_1, T_2 \in_{C,S}$ **Types** then $T_1 \to T_2 \in_{C,S}$ **Types** and $T_1 \times T_2 \in_{C,S}$ **Types**
(iii) 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 $h \in I$ such that $D_h = D$
(d) if $i \neq j$ then $D_i \neq D_j$
then $\{D_i \to T_i\}_{i \in I} \in_{C,S}$ **Types**

B.2 Typing rules

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

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

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

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

[$\to$ Elim$_{(\leq)}$] $$\frac{C, S \vdash M{:}\,U \to T \;\; N{:}\,W}{C, S \vdash MN{:}\,T} \qquad C \vdash W \leq U$$

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

[{}Intro+] $$\frac{C, S \vdash M{:}\,W_1 \leq \{U_i \to V_i\}_{i \in I} \quad C, S \vdash N{:}\,W_2 \leq U \to V}{C, S \vdash (M \&^{\{U_i \to V_i\}_{i \in I} \oplus (U \to V)} N){:}\,\{U_i \to V_i\}_{i \in I} \oplus (U \to V)}$$
$\{U_i \to V_i\}_{i \in I} \oplus (U \to V) \in_{C,S}$ **Types**

The rules for the expressions that do not belong to the syntax of λ& are:

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

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

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

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

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

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

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

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

C Formal definition of the translation

C.1 Simple methods without recursion

Unfortunately the formal interpretation is not so smooth as the intuitive one. Most of the problems derive from the fact that, in λ_object, the variables are typed. Thus, when we translate a set of methods into an overloaded function, we have to concatenate branches so that the resulting term has the required overloaded type.

Formally, let $\mathcal{L}$ be the set of the programs of the toy-language; we define the translation from $\mathcal{L}$ to **Terms** (the set of the *raw* terms of λ_object) using three functions. The first is the function that describes the translation itself:

$$\Im[\![\_]\!] : \mathcal{L} \to Envs \to InitState \to \mathbf{AtomicTypes} \to \mathbf{Terms}$$

Where:

$Envs = Vars \to \mathbf{RawTypes}$	This parameter records the type of the identifiers. It is ranged over by the metavariable Γ.
$InitState = ClassNames \to RecordValues$	This parameter stores the initial value of the instance variables of each class: it is used in the interpretation of `new`. It is ranged over by the metavariable I.

AtomicTypes This parameter is the *current class*, and it is used in the translation of a method.

Therefore $\Im[\![p]\!]_{\Gamma\, I\, A}$ is the term of λ_object that translates the program p.

The definition of $\Im$ is given in term of two auxiliary functions $\mathcal{M}$ and $\mathcal{T}$. $\mathcal{M}[\![p]\!](m)$ returns the (overloaded) term associated to the message m by the definitions in p; $\mathcal{T}[\![p]\!](m)$ returns the (raw) type that indexes the variable (translation of) m. Of course, if p is well typed we expect that $\mathcal{M}[\![p]\!](m){:}\,\mathcal{T}[\![p]\!](m)$.

It is necessary to introduce these auxiliary functions in order to overcome one of the major drawbacks of $\lambda\&$. Suppose we have three classes A, B and C with C defined by multiple inheritance from A and B ($C \leq A, B$). Suppose also that A and B can respond to the same message m; then by the condition of multiple inheritance one has also to define a branch for m with input type C. In object-oriented languages, as in our toy language, the logical order is to define first the branches for A and B and then, at the moment of the definition of C, to append the new branch for C. Thus the definition of m would be of the form

$$\begin{aligned} m \equiv (&\varepsilon \\ &\&^{\{A\to T_1\}}\lambda self^{A}.M_1\ \&^{\{A\to T_1, B\to T_2\}}\lambda self^{B}.M_2 \\ &\&^{\{A\to T_1, B\to T_2, C\to T_3\}}\lambda self^{C}.M_3) \end{aligned} \tag{C.1}$$

This is very reasonable but unfortunately the term above is not well typed, since the second index $\{A \to T_1, B \to T_2\}$ is not a well formed type. In $\lambda\&$ the branch written to solve the ambiguity of multiple inheritance must always precede at least one of the branches of its direct ancestors. In the case above for example the following definition is well typed

$$m \equiv (\varepsilon\ \&^{\{A\to T_1\}}\lambda self^{A}.M_1\ \&^{\{A\to T_1, C\to T_3\}}\lambda self^{C}.M_3\ \&^{\{A\to T_1, C\to T_3, B\to T_2\}}\lambda self^{B}.M_2)$$

This problem can be framed in the more general problem of the definition of dynamic types. $\lambda\&$ completely lacks the notion of time, or better the order of the definition of types. Atomic types are given all at once, and there is no perception of the temporal dependence of type definitions. Thus dynamic types cannot be modeled, and for this reason in our toy language all the class definitions have to precede the expression to execute. Actually we are working on the definition of a type system in which the types use time stamps, so that the definition of m as in (C.1) is well typed. The idea is that an expression with type $\{A \to T_1, B \to T_2\}$ has a well-formed type if all its sub-expressions use types that are older than the definition of C.

However for the moment we do not have time stamps; thus to translate our toy language we have to use the functions $\mathcal{M}$ and $\mathcal{T}$ that pre-scan the program to translate, and build the messages in the reverse way, from the

latest method defined to the first one. Thus to translate a program we are obliged to scan it twice: once to construct methods by reading the definition in the reverse way, the other to translate the whole program.

Since the function $\mathcal{M}[\![\_]\!]$ uses in its definition the function $\Im[\![\_]\!]$, it needs the same parameters as $\Im$ in order to pass them to it; for $\mathcal{T}[\![\_]\!]$ no parameter is needed. Formally:

Definition 23

$$\mathcal{T}[\![\_]\!] : \mathcal{L} \to Vars \to \textbf{Types}$$

(i) $\mathcal{T}[\![$ `class` B `is` $A_1, \ldots, A_p$ $r : R$ $\mathtt{m}_1$`=`$exp_1 \ldots \mathtt{m}_n$`=`$exp_n$`[[` $\mathtt{m}_1$`:`$T_1 \ldots \mathtt{m}_n$ `:` T_n`]]` `in` $p]\!](m) =$

$$= \begin{cases} \mathcal{T}[\![p]\!](m_j) \oplus \{B \to T_j\} \textit{ for } m = m_j \\ \mathcal{T}[\![p]\!](m) \qquad\qquad \textit{else} \end{cases}$$

(ii) $\mathcal{T}[\![\_]\!]$ *is the function that returns* $\{\}$ *in all the other cases.*

Definition 24

$$\mathcal{M}[\![\_]\!] : \mathcal{L} \to Envs \to InitState \to \textbf{AtomicTypes} \to Vars \to \textbf{Terms}$$

(i) $\mathcal{M}[\![$ `class` B `is` $A_1, \ldots, A_q$ $r{:}R$ $\mathtt{m}_1$`=`$exp_1 \ldots \mathtt{m}_n$`=`$exp_n$`[[` $\mathtt{m}_1$`:`$T_1 \ldots \mathtt{m}_n$ `:` T_n`]]` `in` $p]\!]_{\Gamma\, I\, A}(m) =$

$$= \begin{cases} ((\mathcal{M}[\![p]\!]_{\Gamma'\, I\, A}(m_j)) & \\ \quad \&^{\mathcal{T}[\![p]\!](m_j) \oplus \{B \to T_j\}} \lambda \mathrm{self}^B.\Im[\![\mathrm{exp}_j]\!]_{\Gamma[\mathrm{self} \leftarrow B]\, I[B \leftarrow r]\, B}) & \textit{for } m = m_j \\ \mathcal{M}[\![p]\!]_{\Gamma\, I\, A}(m) & \textit{else} \end{cases}$$

(ii) $\mathcal{M}[\![\_]\!]$ *is the function that returns* ε *in all the other cases.*

Where $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \oplus \{B \to T_i\}]_{i=1..n}$.

Note that in $\Im[\![exp_j]\!]_{\Gamma\, I[B \leftarrow r]\, B}$ we have used Γ instead of Γ' since we stated that the methods are not mutually recursive.

Definition 25 (Translation)

(i) $\Im[\![x]\!]_{\Gamma\, I\, A} = x^{\Gamma(x)}$

(ii) $\Im[\![exp_1\mathtt{(}exp_2\mathtt{)}]\!]_{\Gamma\, I\, A} = \Im[\![exp_1]\!]_{\Gamma\, I\, A} \Im[\![exp_2]\!]_{\Gamma\, I\, A}$

(iii) $\Im[\![\mathtt{fn(}x\mathtt{:}T\mathtt{)\ =>}\ exp]\!]_{\Gamma\, I\, A} = \lambda x^T.\Im[\![exp]\!]_{\Gamma[x \leftarrow T]\, I\, A}$

(iv) $\Im[\![\mathtt{let}\ x\mathtt{:}T = exp_1\ \mathtt{in}\ exp_2]\!]_{\Gamma\, I\, A} = (\lambda x^T.\Im[\![exp_2]\!]_{\Gamma[x \leftarrow T]\, I\, A})(\Im[\![exp_1]\!]_{\Gamma\, I\, A})$

(v) $\Im[\![\mathtt{(}exp_1\mathtt{,}\ldots\mathtt{,}exp_n\mathtt{)}]\!]_{\Gamma\, I\, A} = < \Im[\![exp_1]\!]_{\Gamma\, I\, A}, ..., \Im[\![exp_n]\!]_{\Gamma\, I\, A} >$

(vi) $\Im[\![\mathtt{fst(}exp\mathtt{)}]\!]_{\Gamma\, I\, A} = \pi_1(\Im[\![exp]\!]_{\Gamma\, I\, A})$

(vii) $\Im[\![\mathtt{snd(}exp\mathtt{)}]\!]_{\Gamma\, I\, A} = \pi_2(\Im[\![exp]\!]_{\Gamma\, I\, A})$

(viii) $\Im[\![\mathtt{new(}B\mathtt{)}]\!]_{\Gamma\, I\, A} = \mathrm{in}^B(I(B))$

(ix) $\Im[\![\mathtt{[}exp_0\ \ exp\ \ exp_1,\ \ldots,\ \ exp_n\mathtt{]}]\!]_{\Gamma\, I\, A} = \Im[\![exp]\!]_{\Gamma\, I\, A} \bullet \Im[\![\mathtt{(}exp_0\mathtt{,}exp_1\mathtt{,}\ldots\mathtt{,}exp_n\mathtt{)}]\!]_{\Gamma\, I\, A}$

(x) $\Im[\![\mathtt{super[}B\mathtt{](}exp\mathtt{)}]\!]_{\Gamma\, I\, A} = \mathbf{super}^B(\Im[\![exp]\!]_{\Gamma\, I\, A})$

(xi) $\Im[\![\mathtt{coerce[}B\mathtt{](}exp\mathtt{)}]\!]_{\Gamma\, I\, A} = \mathbf{coerce}^B(\Im[\![exp]\!]_{\Gamma\, I\, A})$

(xii) $\Im[\![\mathtt{self}]\!]_{\Gamma\, I\, A} = \mathrm{self}^A$

(xiii) $\Im[\![\texttt{self}.\ell]\!]_{\Gamma\, I\, A} = (\mathrm{out}^A(\mathrm{self}\ ^A)).\ell$

(xiv) $\Im[\![\texttt{(update}\ r\texttt{)}]\!]_{\Gamma\, I\, A} = \mathrm{in}^A(\langle \mathrm{out}^A(\mathrm{self}\ ^A) \leftarrow \ell_1 = \Im[\![exp_1]\!]_{\Gamma\, I\, A} ... \leftarrow \ell_n = \Im[\![exp_n]\!]_{\Gamma\, I\, A}\rangle)$

where $r \equiv \{\ell_1 = \mathrm{exp}_1; ...; \ell_n = \mathrm{exp}_n\}$

(xv) $\Im[\![\ \texttt{extend}\ B\ \texttt{m}_1\texttt{=}exp_1 \ldots\texttt{;m}_n\texttt{=}exp_n\ \texttt{[[m}_1\texttt{:}T_1\texttt{;}\ldots\texttt{;m}_n\texttt{:}T_n\texttt{]]}\ \texttt{in}\ exp]\!]_{\Gamma\, I\, A} =$

$(\lambda m_1^{\Gamma(m_1)\oplus\{B\to T_1\}} ... \lambda m_n^{\Gamma(m_n)\oplus\{B\to T_n\}}.\Im[\![\mathrm{exp}]\!]_{\Gamma'\, I\, A})$

$(m_1^{\Gamma(m_1)} \&^{\Gamma(m_1)\oplus\{B\to T_1\}} \lambda \mathrm{self}^B.\Im[\![\mathrm{exp}_1]\!]_{\Gamma\, I\, B}) \cdots$

$(m_n^{\Gamma(m_n)} \&^{\Gamma(m_n)\oplus\{B\to T_n\}} \lambda \mathrm{self}^B.\Im[\![\mathrm{exp}_n]\!]_{\Gamma\, I\, B})$

In the last rule $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \oplus \{B \to T_i\}]$.

It still remains to give the translation of the programs.

Let p the program

$\texttt{class}\ B\ \texttt{is}\ A_1, \ldots, A_q\ r\texttt{:}R\ \ \texttt{m}_1\texttt{=}exp_1 \ldots \texttt{m}_n\texttt{=}exp_n\texttt{[[}\ \texttt{m}_1\texttt{:}T_1 \ldots \texttt{m}_n\texttt{:}T_n\texttt{]]}\ \texttt{in}\ p'$

then

$\Im[\![p]\!]_{\Gamma\, I\, A} =$

let B **hide** R **in**

let $B \leq A_1 ... A_q$ **in**

$\Im[\![p']\!]_{\Gamma\, I[B\leftarrow r]\, A}[m_i^{(\mathcal{T}[\![p]\!](m_i))} := \mathcal{M}[\![p]\!]_{\Gamma\, I\, A}(m_i)]_{i=1..n}$

C.2 With multi-methods

Define an arbitrary *total* order $\preceq$ on **Types** with the following property: if $S \leq T$ then $S \preceq T$. Given an overloaded type $\{S_i \to T_i\}_{i=i..n}$ we denote by σ the permutation that orders the S_i's according to $\preceq$. Thus $S_i \leq S_j$ implies $\sigma(i) \leq \sigma(j)$. The intuitive translation given in section 5.2 is then formalized by modifying the definitions of the previous section in the following way:

Definition 26

$$\mathcal{T}[\![\_]\!] : \mathcal{L} \to Vars \to \mathbf{Types}$$

(i) $\mathcal{T}[\![\ \texttt{class}\ B\ \texttt{is}\ A_1, \ldots, A_q\ \ r : R\ \texttt{m}_1\texttt{=}exp_1 \ldots \texttt{m}_n\texttt{=}exp_n\texttt{[[}\ \texttt{m}_1\texttt{:}V_1 \ldots \texttt{m}_n : V_n\texttt{]]}\ \texttt{in}\ p]\!](m) =$

$$= \begin{cases} \mathcal{T}[\![p]\!](m_j) \oplus \{B \rightsquigarrow V_j\} & \textit{for } m = m_j \\ \mathcal{T}[\![p]\!](m) & \textit{else} \end{cases}$$

(ii) $\mathcal{T}[\![\_]\!]$ *is the function that returns* $\{\}$ *in all the other cases.*

The definition of $\mathcal{M}[\![.]\!] : \mathcal{L} \to Envs \to InitState \to \mathbf{AtomicTypes} \to Vars \to \mathbf{Terms}$ gets quite harder:

Definition 27

– $\mathcal{M}[\![\ \texttt{class}\ B\ \texttt{is}\ A_1, \ldots, A_q\ \ r\texttt{:}R\ \ \texttt{m}_1\texttt{=}exp_1 \ldots \texttt{m}_n\texttt{=}exp_n\texttt{[[}\ \texttt{m}_1\texttt{:}V_1 \ldots \texttt{m}_n\texttt{:}V_n\texttt{]]}\ \texttt{in}\ p]\!]_{\Gamma\, I\, A}(m) =$

(i) If $m \equiv m_j$ *for some* $j \in [1..n]$ *and* V_j *is a raw type, then the definition is as before*

$$((\mathcal{M}[\![p]\!]_{\Gamma'\, I\, A}(m_j)) \&^{\mathcal{T}[\![p]\!](m_j)\oplus\{B\to V_j\}} \lambda self^B.\Im[\![\mathrm{exp}_j]\!]_{\Gamma[self\leftarrow B]\, I[B\leftarrow r]\, B})$$

(ii) If $m \equiv m_j$ *for some* $j \in [1..n]$ *and* $V_j \equiv \#\{D_i \to T_i\}_{i=1..h}$*, then we are in the case of multi-method and* $\exp_j$ *must be of the following form:*

$$\begin{array}{l}\texttt{\&\ fn(}x_1\texttt{:}\,D_1\texttt{)\ =>}\ exp_{j_1}\\ \qquad\vdots\\ \texttt{\&\ fn(}x_h\texttt{:}\,D_h\texttt{)\ =>}\ exp_{j_h}\end{array}$$

then $\mathcal{M}$ *is defined in the following way:*

$$\begin{array}{l}(\cdots((\mathcal{M}[\![p]\!]_{\Gamma'\,I\,A}(m_j)\\ \quad \&^{\mathcal{T}[\![p]\!](m_j)\oplus\{B\times D_{\sigma(1)}\to T_{\sigma(1)}\}}\lambda(self^B, x_{\sigma(1)}^{D_{\sigma(1)}}).\Im[\![\exp_{j_{\sigma(1)}}]\!]_{\Gamma[self\leftarrow B]\,I[B\leftarrow r]\,B})\\ \qquad\qquad\vdots\\ \quad \&^{(\mathcal{T}[\![p]\!](m_j)\oplus\ldots\oplus\{B\times D_{\sigma(h-1)}\to T_{\sigma(h-1)}\})\oplus\{B\times D_{\sigma(h)}\to T_{\sigma(h)}\}}\\ \qquad\qquad\qquad\lambda(self^B, x_{\sigma(h)}^{D_{\sigma(h)}}).\Im[\![\exp_{j_{\sigma(h)}}]\!]_{\Gamma[self\leftarrow B]\,I[B\leftarrow r]\,B})\end{array}$$

Where $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \oplus \{B \rightsquigarrow T_i\}]_{i=1..n}$ *and* σ *is the permutation described above*

(iii) Else the definition of $\mathcal{M}$ *is* $\mathcal{M}[\![p]\!]_{\Gamma\,I\,A}(m)$

– $\mathcal{M}[\![.]\!]$ *is the function that returns* ε *in all the other cases.*

Finally we have to modify the definition of $\Im$ for `extend`. We do not write it here since the modifications follow the pattern of those we have done in the definition above. This definition, however can be found in the case 14 of the theorem 30.

C.3 With recursive methods

To cope with recursive (possibly multi-) methods we have to change the definition of $\mathcal{M}$ in the following way:

Definition 28 *Let* p' *denote the program* `class` B `is` A_1`,...,`A_q $r\texttt{:}\,R$ $\texttt{m}_1\texttt{=}exp_1 \ldots \texttt{m}_n\texttt{=}exp_n$`[[` $\texttt{m}_1\texttt{:}V_1 \ldots \texttt{m}_n : V_n$`]] in` p*; then*

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

Where $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \oplus \{B \to T_i\}]_{i=1..n}$ *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

(i) If V_j *is a raw type then*

$$M_j \equiv ((\mathcal{M}[\![p]\!]_{\Gamma'\,I\,A}(m_j))\&^{\mathcal{T}[\![p]\!](m_j)\oplus\{B\rightsquigarrow V_j\}}\lambda self^B.\Im[\![\exp_j]\!]_{\Gamma'[self\leftarrow B]\,I[B\leftarrow r]\,B})$$

(ii) If $j \in [1..n]$ *and* $V_j \equiv \#\{D_i \to T_i\}_{i=1..h}$*, then* $\exp_j$ *must be of the following form:*

$$\begin{array}{l}\texttt{\&\ fn(}x_1\texttt{:}\,D_1\texttt{)\ =>}\ exp_{j_1}\\ \qquad\vdots\end{array}$$

`&` `fn(`x_h`:` D_h`)` `=>` exp_{j_h}

then M_j *is defined in the following way:*

$$(\cdots((\mathcal{M}[\![p]\!]_{\Gamma' I A}(m_j)$$
$$\&^{\mathcal{T}[\![p]\!](m_j)\oplus\{B\times D_{\sigma(1)}\to T_{\sigma(1)}\}}\lambda(self^B, x^{D_{\sigma(1)}}_{\sigma(1)}).\Im[\![\exp_{j_{\sigma(1)}}]\!]_{\Gamma'[self\leftarrow B]\,I[B\leftarrow r]\,B})$$
$$\vdots$$
$$\&^{(\mathcal{T}[\![p]\!](m_j)\oplus\ldots\oplus\{B\times D_{\sigma(h-1)}\to T_{\sigma(h-1)}\})\oplus\{B\times D_{\sigma(h)}\to T_{\sigma(h)}\}}$$
$$\lambda(self^B, x^{D_{\sigma(h)}}_{\sigma(h)}).\Im[\![\exp_{j_{\sigma(h)}}]\!]_{\Gamma'[self\leftarrow B]\,I[B\leftarrow r]\,B})$$

where σ *has the usual property.*

– $\mathcal{M}[\![.]\!]$ *is the function that returns* ε *in all the other cases.*

Similar modifications are to be done in the interpretation of `extend`.

C.4 Correctness of the type-checking

We prove something stronger than the well typing of a term obtained by translating a well-typed program: we prove that the translated program possesses the same type as its translation; note indeed that the types of the toy language are the same as those of λ_object.

Since the definition of $\Im[\![.]\!]$ is mutually recursive with $\mathcal{M}[\![.]\!]$ then the theorem must be proved mutually recursively with a theorem on $\mathcal{M}[\![.]\!]$. Thus the main theorem will be split in two propositions. But first we need some auxiliary notation:

Notation 29 *We denote by* C_p *the set of type constraints declared in* p*, that is* $C_p = \emptyset$ *if* p *is an expression and* $C_{(class\ A\ is\ A_1\ldots A_n\ \ldots\ in\ p')} = (A \le A_1)\cup\ldots\cup(A \le A_n)\cup C_{p'}$*. We denote by* S_p *the stores of the internal states defined in* p*: again* $S_p = \emptyset$ *if* p *is an expression and* $S_{(class\ A\ is\ldots r{:}R\ldots in\ p')} = [A \leftarrow R]\cdot S_{p'}$ *(here* $\cdot$ *denotes simple juxtaposition)*

Theorem 30 *For every type constraint* C*, type environment* Γ*; for every* $I \in InitState$ *and* $S : ClassNames \to$**RecordTypes** *such that* $I(A){:}\,S(A)$ *(for every* A *atomic); if*

$$C; S; \Gamma \vdash p{:}\,T$$

then

(i) for all $m \in Vars \qquad C \cup C_p; S\cdot S_p \vdash \mathcal{M}[\![p]\!]_{\Gamma\, I\,\Gamma(self)}(m){:}\,\mathcal{T}[\![p]\!](m)$

(ii) $C; S \vdash \Im[\![p]\!]_{\Gamma\, I\,\Gamma(self)}{:}\,T$

Proof. We prove the theorem only for the case in which there are no mutually recursive methods; recursive terms do not pose any problem from the viewpoint of type-checking, but render the proof more unreadable. The proof goes by induction on p. When p is formed only by an expression then the part 1 of the theorem is trivially proved by [Taut$_\varepsilon$]. Thus in the rest of the proof we will prove the the part 1 of the theorem only when when p is is of the form `class` A `is ...[[...]]` .

(i) $p \equiv x$ but then $\Im[\![x]\!]_{\Gamma\, I\,\Gamma(self)} = x^{\Gamma(x)}{:}\,\Gamma(x)$ thus we have the result.

(ii) $p \equiv exp_1$`(`exp_2`)` then $C; S; \Gamma \vdash exp_1 : T_1 \to T$ and $C; S; \Gamma \vdash exp_2 : T_2 \le T_1$.

By induction $C; S \vdash \Im[\![exp_1]\!]_{\Gamma\, I\, \Gamma(self)} : T_1 \to T$ and $C; S \vdash \Im[\![exp_2]\!]_{\Gamma\, I\, \Gamma(self)} : T_2$. We obtain the thesis by $[\to\text{ELIM}_{(\leq)}]$.

(iii) $p \equiv$ (`fn` $x : T_1$ `=>`exp) then $C; S; \Gamma[x \leftarrow T_1] \vdash exp : T_2$ where $T \equiv T_1 \to T_2$. By induction $C; S \vdash \Im[\![exp]\!]_{\Gamma[x \leftarrow T_1]\, I\, \Gamma(self)} : T_2$. Therefore $C; S \vdash \lambda x^{T_1}.\Im[\![exp]\!]_{\Gamma[x \leftarrow T_1]\, I\, \Gamma(self)} : T_1 \to T_2$.

(iv) $p \equiv$ (`let` $x : T_1$ `=` exp `in` exp'); combine the techniques of the previous two cases.

(v) $p \equiv$ `snd(`exp`)` a straightforward use of the induction hypothesis.

(vi) $p \equiv$ `fst(`exp`)` a straightforward use of the induction hypothesis.

(vii) $p \equiv$ `new(`A`)`. By hypothesis $I(A) : S(A)$ therefore $in^A(I(A))$ is well typed and has type A.

(viii) $p \equiv$ `[`exp_0 exp exp_1, ...,exp_n `]` then $C; S; \Gamma \vdash exp : \{D_i \to T_i\}_{i \in I}$ and $C; S; \Gamma \vdash exp_i : A_i$ with $D_j = \min_{i \in I}\{D_i | C \vdash A_0 \times ... \times A_n \leq D_i\}$ and $T \equiv T_j$. From the induction hypothesis $C; S \vdash \Im[\![exp]\!]_{\Gamma\, I\, \Gamma(self)} : \{D_i \to T_i\}_{i \in I}$ and $C; S \vdash \Im[\![(exp_0, exp_1, ..., exp_n)]\!]_{\Gamma\, I\, \Gamma(self)} : A_0 \times ... \times A_n$. Then the thesis is obtained by $[\{\}\text{ELIM}]$.

(ix) $p \equiv$ `coerce[`A`](`exp`)` thus $T \equiv A$ and $C; S; \Gamma \vdash exp : T_1 \leq A$. By induction hypothesis $C; S \vdash \Im[\![exp]\!]_{\Gamma\, I\, \Gamma(self)} : T_1$. Thus $\mathbf{coerce}^A(\Im[\![exp]\!]_{\Gamma\, I\, \Gamma(self)})$ is well-typed and has type A.

(x) $p \equiv$ `super[` A`](`exp`)`. As the previous case.

(xi) $p \equiv$ `self` straightforward

(xii) $p \equiv$ (`self.`ℓ) Then $S(\Gamma(\texttt{self})) = \langle\!\langle ...\ell : T... \rangle\!\rangle$.
Since $self^{\Gamma(self)} : \Gamma(self)$ and $out^{\Gamma(self)} : \Gamma(self) \to S(\Gamma(self))$ then $out^{\Gamma(self)}(self^{\Gamma(self)}) : \langle\!\langle ...\ell : T... \rangle\!\rangle$. Thus $(out^{\Gamma(self)}(self^{\Gamma(self)})).\ell : T$.

(xiii) $p \equiv$ `update(`r`)` Then $T \equiv \Gamma(self)$, $C; S; \Gamma \vdash r : R$ and $C \vdash S(\Gamma(self)) \Subset R$. If $r \equiv$ `{`ℓ_1`=`exp_1`;...;`ℓ_n`=`exp_n`}` then by induction hypothesis

$$C; S \vdash \langle \ell_1 = \Im[\![exp_1]\!]_{\Gamma\, I\, \Gamma(self)}; ...; \ell_n = \Im[\![exp_n]\!]_{\Gamma\, I\, \Gamma(self)} \rangle : R$$

By definition $out^{\Gamma(self)}(self^{\Gamma(self)}) : S(\Gamma(self))$. Since $C \vdash S(\Gamma(self)) \Subset R$ then
$(\langle out^{\Gamma(self)}(self^{\Gamma(self)}) \leftarrow \ell_1 = \Im[\![exp_1]\!]_{\Gamma\, I\, \Gamma(self)} ... \leftarrow \ell_n = \Im[\![exp_n]\!]_{\Gamma\, I\, \Gamma(self)} \rangle)$ is well typed and has type $S(\Gamma(self))$.
Therefore also $\Im[\![p]\!]_{\Gamma\, I\, \Gamma(self)} \equiv in^{\Gamma(self)}(\langle out^{\Gamma(self)}(self^{\Gamma(self)}) \leftarrow ... \rangle)$ is well-typed and has type $\Gamma(self)$.

(xiv) We prove w.l.o.g. the case for `extend` with only one multi-method: the case with ordinary methods is a slight modification of this case that can be deduced from the next case; extensions including more than one method can be translated in a suite of extensions with only one method, since, we recall, we do not consider the case of mutually recursive methods.
Let $p \equiv$ `extend` A `m=&`$exp_1 \ldots$ `&`exp_n `in [[m:`V`]] in` exp
where $V \equiv$ `#`$\{D_1 \to T_1, \ldots, D_n \to T_n\}$
and $D_i \equiv A^i_1 \times \ldots \times A^i_{n_i}$
and $exp_i \equiv$ `fn(`$x^i_1 : .A^i_1, \ldots, x^i_{n_i} : .A^i_{n_i}$`) =>` exp'_i `(for` $i = 1..n$`)`

Let exp_1^* denote the following expression:

$$\texttt{fn(self:}\,A,\ x_1^i\texttt{:.}A_1^i\texttt{,...,}x_{n_i}^i\texttt{:.}A_{n_i}^i\texttt{) => }\ exp_i'$$

Then p is translated into:
$(\lambda m^{\Gamma(m)\oplus\{A\rightsquigarrow V\}}.\Im[\![exp]\!]_{\Gamma\, I\, \Gamma(self)})$
$\quad(m^{\Gamma(m)}$
$\quad\&^{\Gamma(m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}}\Im[\![exp_1^*]\!]_{\Gamma\, I\, B}$
$\qquad\vdots$
$\quad\&^{(\Gamma(m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}\oplus\ldots\oplus\{A\times D_{\sigma(n-1)}\to T_{\sigma(n-1)}\})\oplus\{A\times D_{\sigma(n)}\to T_{\sigma(n)}\}}\Im[\![exp_n^*]\!]_{\Gamma\, I\, B}$
$\quad)$
By hypothesis

$$C;S;\Gamma[self\leftarrow A]\vdash \texttt{\&}exp_1\ldots\texttt{\&}exp_n\colon V$$

and thus it is clear that

$$C;S;\Gamma[self\leftarrow A]\vdash \texttt{\&}exp_{\sigma(1)}\ldots\texttt{\&}exp_{\sigma(n)}\colon V$$

Therefore

$$C;S;\Gamma\vdash \texttt{\&}exp^*_{\sigma(1)}\ldots\texttt{\&}exp^*_{\sigma(n)}\colon A\rightsquigarrow V \tag{C.2}$$

Also by hypothesis

$$C;S;\Gamma[m\leftarrow\Gamma(m)\cup\{A\rightsquigarrow V\}]\vdash exp\colon T \tag{C.3}$$

Note now that given an overloaded type V if $V\cup\{S\to T\}$ is a well formed overloaded type then

1. Also $V\oplus\{S\to T\}$ is well formed
2. $V\cup\{S\to T\}=V\oplus\{S\to T\}$

Thus from (C.3) we obtain

$$C;S;\Gamma[m\leftarrow\Gamma(m)\oplus\{A\rightsquigarrow V\}]\vdash exp\colon T$$

We can now apply the induction hypothesis obtaining:

$$C;S\vdash\lambda m^{\Gamma(m)\oplus\{A\rightsquigarrow V\}}.\Im[\![exp]\!]_{\Gamma\, I\, \Gamma(self)}\colon(\Gamma(m)\oplus\{A\rightsquigarrow V\})\to T$$

Thus the thesis holds if we prove that the term

$$(\cdots(m\&^{\Gamma(m)\oplus\ldots}\ldots\&^{(\Gamma(m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}\oplus\ldots)\oplus\{A\times D_{\sigma(n)}\to T_{\sigma(n)}\}}\ldots)$$

has type $\Gamma(m)\oplus\{A\rightsquigarrow V\}$ This can be proved by induction on n: for $n=1$ the thesis is a strightforward application of the induction hypothesis on exp_1^* for (C.2). Consider now

$$(\cdots(m\&^{\Gamma(m)\oplus\ldots}\ldots\&^{(\Gamma(m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}\oplus\ldots)\oplus\{A\times D_{\sigma(i)}\to T_{\sigma(i)}\}}\Im[\![exp_i^*]\!]_{\Gamma\, I\, B})$$

Using the induction hypothesis on (C.2) it easy to see that the thesis fails only if $\Gamma(m) \oplus \{A \times D_{\sigma(1)} \to T_{\sigma(1)}\} \oplus ... \oplus \{A \times D_{\sigma(i)} \to T_{\sigma(i)}\}$ is not a well formed overloaded type. But since $\Gamma(m) \oplus \{A \rightsquigarrow V\}$ is well-formed, thus the previous type (which is a "subset" of this) surely satisfies the conditions of covariance and input type uniqueness. And thanks to the definition of σ it also satisfies the condition of multiple inheritance: if $A \times D_{\sigma(i)}$ has a strict lower bound in common with any other input type, then all the branches with maximal input types (which must already be in $\Gamma(m) \oplus \{A \rightsquigarrow V\}$) are already in $\Gamma(m) \oplus \{A \times D_{\sigma(1)} \to T_{\sigma(1)}\} \oplus ... \oplus \{A \times D_{\sigma(i)} \to T_{\sigma(i)}\}$, for either they are in $\Gamma(m)$ or they are of the form $\{A \times D_{\sigma(j)} \to T_{\sigma(j)}\}$ but then, because of the condition on σ, we have $\sigma(j) < \sigma(i)$.

(xv) As in the previous case we consider a simpler version where we have only one ancestor and one method in the class declaration: the general case can be obtained by adding some indexing in the right places.

$p \equiv$`class` A `is` A' $r{:}R$ $m = exp$ `[[`$m{:}V$`]]` `in` p'. This is the only case where the proof of the first part of the theorem is non-trivial thus:

1. We have to prove that for all $\overline{m} \in \mathit{Vars}$

$$C \cup C_p; S \vdash \mathcal{M}[\![p]\!]_{\Gamma\, I\, \Gamma(self)}(\overline{m}){:}\, \mathcal{T}[\![p]\!](\overline{m})$$

If $\overline{m} \not\equiv m$ then the thesis follows from the induction hypothesis. Otherwise let first consider the case when $m = exp$ is not a multi-method; then $\mathcal{T}[\![p]\!](m) = \mathcal{T}[\![p']\!](m) \oplus \{A \to V\}$. Since p is well-typed then it is easy to prove that $\mathcal{T}[\![p]\!](m)$ is a well-formed type; moreover it holds that

$$\begin{aligned}\mathcal{M}[\![p]\!]_{\Gamma\, I\, \Gamma(self)}(m) = (&\mathcal{M}[\![p']\!]_{\Gamma'\, I\, \Gamma(self)}(m)\\ &\&^{(\mathcal{T}[\![p']\!](m)) \oplus \{A \to V\}} \lambda self^A.\Im[\![exp]\!]_{\Gamma\, I[A \leftarrow r]\, A}\\ &)\end{aligned}$$

By induction hypothesis $C \cup C_{p'}; S \cdot S_{p'} \vdash \mathcal{M}[\![p']\!]_{\Gamma'\, I\, \Gamma(self)}(m){:}\, \mathcal{T}[\![p']\!](m)$. Furthermore by hypothesis we have that

$$C \cup (A \leq A'); S[A \leftarrow R]; \Gamma[\texttt{self} \leftarrow A] \vdash exp{:}\, V$$

By induction hypothesis on the part 2 of the theorem we have

$$C \cup (A \leq A'); S[A \leftarrow R] \vdash \Im[\![exp]\!]_{\Gamma[self \leftarrow A]\, I[A \leftarrow r]\, A}{:}\, V$$

(Note that $r{:}R$ and thus the hypothesis on I and S holds). By construction exp is not affected by the declarations in p' thus one also has

$$C \cup (A \leq A') \cup C_{p'}; S[A \leftarrow R] \cdot S_{p'} \vdash \Im[\![exp]\!]_{\Gamma[self \leftarrow A]\, I[A \leftarrow r]\, A}{:}\, V$$

which is equivalent to

$$C \cup C_p; S \cdot S_p \vdash \Im[\![exp]\!]_{\Gamma[self \leftarrow A]\, I[A \leftarrow r]\, A}{:}\, V$$

But then

$$C \cup C_p; S \cdot S_p \vdash \lambda self^A.\Im[\![exp]\!]_{\Gamma[self \leftarrow A]\, I[A \leftarrow r]\, A}{:}\, A \to V$$

The thesis follows by the rule [{}Intro].

In the case of a multi-method then exp must be of the form (`&` exp_1 ... `&` exp_n) and $V \equiv$ `#`$\{D_1 \to T_1, \ldots, D_n \to T_n\}$. Again since

p is well-typed it can be shown that $\mathcal{T}[\![p]\!]$ is a well-formed type. Then define exp_i^* as in the previous case. Thus we have to prove under the assumptions $C \cup C_p$ and $S \cdot S_p$ that

$$
\begin{array}{l}
(\mathcal{M}[\![p']\!]_{\Gamma' \, I \, \Gamma(self)}(m) \\
\&^{\mathcal{T}[\![p']\!](m) \oplus \{A \times D_{\sigma(1)} \rightarrow T_{\sigma(1)}\}} \Im[\![exp^*_{\sigma(1)}]\!]_{\Gamma[self \leftarrow A] \, I[A \leftarrow r] \, A} \\
\qquad\qquad\qquad\qquad \vdots \\
\&^{(\mathcal{T}[\![p']\!](m) \oplus \ldots \oplus \{A \times D_{\sigma(n-1)} \rightarrow T_{\sigma(n-1)}\}) \oplus \{A \times D_{\sigma(n)} \rightarrow T_{\sigma(n)}\}} \Im[\![exp^*_{\sigma(h)}]\!]_{\Gamma[self \leftarrow A] \, I[A \leftarrow r] \, A} \\
)
\end{array}
$$

has type $\mathcal{T}[\![p']\!](m) \oplus \{A \rightsquigarrow V\}$ This can be shown by induction on n. For $n = 1$ use the induction hypothesis on p'. For $n > 1$ the proof is exactly the same as the corresponding one of the previous case.

2. We know that $C; S; \Gamma \vdash p : T$ and we have to prove that under the hypothesis C and S the following expression

   let A **hide** R **in**
   let $A \leq A'$ **in**
   $\Im[\![p']\!]_{\Gamma \, I \, A}[m^{(\mathcal{T}[\![p]\!](m))} := \mathcal{M}[\![p]\!]_{\Gamma \, I \, A}(m)]$

   has type T. Thus we prove that
   $C \cup (A \leq A'); S[A \leftarrow R] \vdash \Im[\![p']\!]_{\Gamma \, I[A \leftarrow r] \, \Gamma(self)} : T$
   $R \leq S(A')$
   $\mathcal{M}[\![p]\!]_{\Gamma \, I \, \Gamma(self)}(m) : \mathcal{T}[\![p]\!](m)$ so that we substitute the variable $m^{\mathcal{T}[\![p]\!](m)}$ by a term of the same type.

   The first two conditions follow from the fact that $C; S; \Gamma \vdash p : T$ and by induction hypothesis on p'.

   Clearly $m^{\mathcal{T}[\![p]\!](m)}$ appears after the declarations given in p' since in λ_object no expressions can precede a **let ... in** declaration. Thus the thesis follows if we prove the point (iii) in an environment where also the constraints of p' are considered. Thus what we need to prove is that:

   $C \cup (A \leq A') \cup C_{p'}; S[A \leftarrow R] \cdot S_{p'} \vdash \mathcal{M}[\![p]\!]_{\Gamma \, I \, \Gamma(self)}(m) : \mathcal{T}[\![p]\!](m)$

   But since $(A \leq A') \cup C_{p'} = C_p$ and $[A \leftarrow R] \cdot S_{p'} = S_p$ then it is exactly what we have proved in the proposition 1 of the theorem

□