Extensible Objects: a Tutorial[⋆]

Viviana Bono

Università di Torino, Dipartimento di Informatica
corso Svizzera 185, 10149 Torino, Italy
bono@di.unito.it

Abstract. In the object-oriented realm, class-based languages dominate the world of production languages, but object-based languages have been extensively studied to provide the foundations of the object-oriented paradigm. Moreover, object-based languages are undergoing a Renaissance thanks to the growing popularity of scripting languanges, which are essentially object-based.
We focus on *extensible* object-based calculi, which feature method addition, together with classical method override and method invocation. Extensible objects can be seen as a way to bridge the gap between the class-based setting and the pure object-based setting.
Our aim is to provide a brief but rigorous view on extensible objects, following a thread suggested by the concept of "self" (which is the reference to the executing object) and its related typing problems.
This tutorial may be seen as a complementary contribution to the literature which has explored and compared extensively pure object-based and class-based foundations (for example, as in the books by Abadi and Cardelli, and Bruce, respectively), but which generally neglected extensible objects.

Keywords: object-oriented calculi, object-based calculi, extensible objects, types

1 Introduction

Object-oriented programming languages enjoy an ever growing popularity, as they are a tool for designing maintainable and expandable code, and are also suited for developing web applications and mobile code. It is possible to distinguish among class-based languages and object-based languages. Class-based ones relies on class hierarchies and objects are the class instances, while object-based ones offer objects as the only computational entities, on which also inheritance is defined. Production languages, such as Java [AG96], are usually class-based. They are considered suited to design software *in-the-large*, because they are well-coupled with software engineering principles, thanks to the abstraction mechanism offered by the class hierarchies. In object-based languages (such as Self

[⋆] This work has been partially supported by EU FET - Global Computing initiative, project DART IST-2001-33477, and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

[US87], Obliq [Car95], or Cecil [CG]), new objects may be constructed from already existing objects, inheriting properties from the latter, so inheritance plays its rôle directly at the object level. Most object-based languages and calculi offer an *override* operation (to modify the components of the parent object), while few of them feature also an *addition* operation (to add new components to the parent object), giving rise to *extensible objects* and to a more complete form of inheritance which resembles class-based inheritance. Object-based languages have not been popular thus far as production languages, but have been the basis of more than fifteen years of study on the theoretical foundations of object-oriented languages. Object-oriented languages have introduced a number of ideas, concepts and techniques, which have proved to be useful and effective in programming, but which needed (and some of them still need) to be fully formalized, in order to understand, exploit and ameliorate them. Many techniques to prove program soundness have been developed in the object-based setting, because the concept of "object" is seen as more primitive from a formal-mathematical point of view than the concept of "class".

However, object-based languages, and in particular the ones featuring extensible objects, nowadays are becoming interesting also for practical purposes, thanks to the new challenges that come from the Internet. The growing use of a network as a primary environment for developing, distributing and running programs requires new supporting infrastructures. In particular, there is the need of a greater flexibility, allowing software to execute in different environments essentially without changes, and the requirement to develop working prototypical applications in a relatively short time. That is why scripting languages, like JavaScript [Fla99], that offer a set of easy-to-use primitives to program web applications, are becoming increasingly popular. There is a number of ongoing researches on foundations for scripting languages [BDG02,DG03], as well as for prototyping *delegation*-based languages [AD02b,ABDD03] (whose objects' main feature is to keep a link to the objects they were created from, which gives them the ability to *delegate* the execution of a method to another object). Research in these fields is especially oriented towards the design of type systems, balancing the need of a certain degree of safety during the program execution (in the style of "a well-typed program cannot go wrong") and the requirement of flexibility (not to restrain reusability and not to slow down the design of prototypes).

Extensible objects are the right tool to formalize both scripting and fast-prototyping calculi, as well as hybrid object-based/class-based languages (which try to exploit the best of both approaches as in, for instance, [AD02a]), and to experiment with appropriate type systems: therefore, extensible objects are back in use. With this tutorial we would like to give an essential survey on the basics of the topic, which is lacking in the object-oriented literature, that generally focused on pure object-based calculi (a reference on this topic is the book by Abadi and Cardelli [AC96a]), and on class-based languages (a reference on this other topic is the book by Bruce [Bru02]).

In this tutorial we present a functional object-based calculus with *fields* (otherwise called *instance variables*) and methods, equipped with method and field invocation, method and field addition and method and field override, and we analyze it in a classical way, that is, from the point of view of its type system. Prerequisites for this tutorial are a general knowledge of the object-oriented terminology and of functional calculi and their type systems. A book that may provide the appropriate background is the one by Pierce [Pie02]. Another reference that is useful to grasp the basics of the object-based language approach (also from an implementation point of view) is Chapter 4 of [AC96a].

The tutorial is structured as follows. Section 2 discusses extensible objects from the point of view of their typing, and presents some informal examples motivating the introduction of *MyType* inheritance. Section 3 introduces the calculus $\mathtt{Obj}^+$, through its syntax and operational semantics, then in Section 4 $\mathtt{Obj}^+$'s type system is illustrated. Section 5 gives an overview on a semantics by encoding of $\mathtt{Obj}^+$, casting some light on the formal meaning of extensible objects. Section 6 present a way of modelling classes following the "Classes = Extensible Objects + Encapsulation" model of [Fis96]. Finally, in Sections 7 we discuss some related work.

This paper is partly based on the material for the lectures given by the author at the "Mini-Ecole Chambéry-Turin d'Informatique Théorique" (29 January – 1 February 2003).

2 An Overview of *MyType*

A type discipline is important for object-oriented languages to ensure safety (i.e., absence of *message-not-understood* run-time errors), and the design of safe type systems depends on the knowledge of the semantics of the object-oriented language in question. In fact, safety is proved via a *Subject Reduction* theorem based on a (usually intuitive) operational semantics that mimic program execution. Subject Reduction states that if a program is well-typed (with type τ), then the result of its computation is well-typed (with type τ). From this theorem, it is possible to prove a formal safety property ensuring that "well-typed programs cannot go wrong".

Typing object-oriented programs is an hard task, because one must take in account many features which may be in conflict with each other. We will hint at the well-known conflict between inheritance and subtyping in Section 4.2, and instead discuss in this section some other important issues about the typing of object-oriented calculi. In particular, we will concentrate on the concept of *self* (sometimes called *this*, e.g., in Java, and here denoted with SELF), which is of paramount importance in object-oriented languages. SELF is a special variable that allows reference to the object executing the current method, and therefore permits the invocation of the sibling methods and the access to the object's fields.

We now overview briefly some concepts that will lead us to understand how to treat SELF in a functional object-based calculus with extensible objects.

Objects. Records are an intuitive way to model objects since both are collections of name/value pairs. The records-as-objects approach was in fact developed in the pioneering work on object-oriented calculi [CW85], in which inheritance was modeled by record subtyping. Unlike records, however, object methods should be able to modify fields and invoke sibling methods [Coo89]. To be capable of updating the object's internal state, methods must be functions of the host object (SELF). Therefore, objects must be *recursive* records. Moreover, SELF must be appropriately updated when a method is inherited, since new methods and fields may have been added and/or old ones redefined in the new host object.

Object updates. If all object updates are imperative, SELF can be bound to the host object when the object is instantiated from the class. We refer to this approach as *early* SELF binding. SELF then always refers to the same record, which is modified imperatively in place by the object's methods. The main advantage of early binding is that the fixed-point operator (which gives to methods the possibility to reference the host object, i.e., SELF) has to be applied only once, at the time of object instantiation. If functional updates must be supported — which is, obviously, the case for purely functional object calculi — early binding does not work (see, for example, [AC96a], where early binding is called *recursive semantics*). With functional updates, each change in the object's state creates a new object. If SELF in methods is bound just once, at the time of object instantiation, it will refer to the old, incorrect object and not to the new, updated one. Therefore, SELF must be bound each time a method is invoked. We refer to this approach as *late* SELF binding.

Object extension. Object extension in an object-based calculus is typically modelled by an operation that extends objects by adding new methods and fields to them. There are two constraints on such an operation: (*i*) the type system must prevent addition of a component to an object which already contains a component with the same name, and (*ii*) since an object may be extended again after addition, the actual host object may be larger than the object to which a method was originally added. Thus, method bodies must behave correctly in any extension of the original host object. Therefore, they must have a polymorphic type with respect to SELF. The fulfillment of the two constraints can be achieved, for instance, via polymorphic types built on row schemes [BF98] that use kinds to keep track of methods' presence. Even more complicated is the case when object extension must be supported in a functional calculus. In the functional case, all methods modifying an object have the type of SELF as their return type. Whenever an object is extended or has its components redefined (overriden), the type given to SELF in all inherited methods must be updated to take into account new and/or redefined components. Therefore, the type system should include the notion of the "type of SELF", called *MyType* (a.k.a. *SelfType*), so that the inherited methods can be specialized properly. Support for *MyType* generally leads to more complicated type systems, in which forms of recursive types are required. *MyType* can be supported by using row variables combined with recursive types [FHM94,FM95,Fis96], by means of special forms of second-order quantifiers such as the *Self* quantifier of [AC96a], or with match-bound type variables as in [BB99,BBC02] and in the calculus presented in this tutorial.

Our calculus $\texttt{Obj}^{+}$, which will be introduced in Section 3, is chosen to give a complete example of a calculus with extensible objects and a functional semantics of method addition. This is, in our opinion, the most difficult semantics to grasp, and should serve the purpose of giving the reader enough tools to tackle other calculi.

We now introduce two examples to show the gain of introducing an appropriate type *MyType* for the host object SELF. The examples are expressed in a syntactic-sugared functional object-based pseudo-language (which offers the same features of the formal calculus we will introduce in Section 3) where:

1. we distinguish among fields and methods — in particular, each method uses SELF (i.e., it is a implicit function of SELF);
2. a dot '`.`' represents component selection;
3. the symbol '`:=`' stands for (functional) field override, and the keyword `extends` represents component addition;
4. '`;`' indicates concatenation of statements. This is codifiable in any functional language[1].

The first example we present is a simple and standard one.

```
let $Point \equiv \langle\ x = 0,\ \ move(d) = (\textsc{self}.x := (\textsc{self}.x + d); \texttt{return}\ \textsc{self})\ \ \rangle$
    in let ($ColorPoint$ extends $Point$) $\equiv \langle\ \ color = blue,$
                                          $setcolor(c) = (\textsc{self}.color := c; \texttt{return}\ \textsc{self}),$
                                          $\ldots\ \rangle$
     in / * main * /  $ColorPoint.move.setcolor$
```

The object *ColorPoint* inherits the methods from *Point*; in particular, it inherits the method *move* which returns a modified SELF (i.e., a new object is created from the host object by modifying the field x through an update of the field x itself with a value computed from its current value plus a displacement d). The main program executes without errors, but within traditional type systems it is not typable, since *ColorPoint.move* would either be of type of *Point*, and *Point* does not have a *setcolor* method, or of the most general type possible (e.g., in Java it would be `Object`), again making impossible calling the specific methods of *ColorPoint*. The solution adopted in traditional programming languages to make this program type check is to use *typecasts*, which are explicit declarations made by the programmer on the expected actual type of the method result according to the type of the object the method is invoked upon. Typecasts are unsafe—the programmer must be "sure" about the actual type of the returned object, since little or even no static checking is performed on typecasts, as happens, respectively, in Java or in C++. Moreover typecasts certainly do not improve readability of code, which has a negative impact on the debugging phase. An alternative to typecasts is the introduction of *selftype* (otherwise known as *MyType*), with the meaning "the type of the current object",

[1] Of course in call-by-value calculi codification of statement concatenation is easier.

i.e., “the type of self”. If we annotate `move` with the *MyType* as its return type, then *ColorPoint.move* and *ColorPoint* will have the *same* type and the main program will typecheck and work without typecasts, because *MyType specialize* along the *Point*/*ColorPoint* hierarchy (that is why the use of *MyType* is also known as *MyType specialization*).

The second example defines a linked/double-linked list, and it is largely inspired to a class-based example that can be found in Bruce's book [Bru02] on page 41.

```
let Node ≡ ⟨  val = 0,  next = null,
              getVal = (return SELF.val),
              setVal(nv) = (SELF.val := nv; return SELF),
              getNext = (return SELF.next),
              setNext(nn) = (SELF.next := nn; return SELF),
              attachRight(nn) = (return SELF.setNext(nn); ⟩
 in let (DbleNode extends Node) ≡ ⟨  prev = null,  getPrev = (return SELF.prev),
                                     setPrev(np) = (SELF.prev := np; return SELF),
                                     attachRight(nn) =
                                       (return nn.setPrev(SELF.setNext(nn)))  ⟩
  in /* main */
```

We type this example by giving *MyType* as the type for the fields *next* and *prev*[2], as the return type of *setVal*, *getNext*, *setNext*, *attachRight*, *getPrev*, *setPrev*, and also as the type of the parameters *nn* of *setNext* and *attachRight*, and *np* of *setPrev*. Therefore, *setNext* has type *MyType*→ *MyType*, with *MyType* “changing of meaning” according to the object the method is invoked upon: *Node.getNext* has type *Nodetype* → *Nodetype*, and *DbleNode.getNext* has type *DbleNodetype* → *DbleNodetype*. The same “changing of meaning” is reproduced for *attachRight*.

Note that in an imperative setting, where the update of the fields can be done by side-effects, there is no need to return SELF, and, as a consequence, the return types of *setVal*, *setNext*, *attachRight*, *setPrev* can be a “void” type without losing expressive power. However, it would be impossible to give a sensible type to parameters such as *nn*, since *setNext*'s parameter must have the *same* type as the object on which *setNext* is invoked upon (i.e., it must have a *Node* type if it is invoked on a *Node*, or a *DbleNode* type if it is invoked on a *DbleNode*). Methods such as *setNext* are called *binary* methods [BCC$^+$95]. Being able to type binary methods is a special feature of functional calculi supporting *MyType*.

3 The Calculus

Our specimen calculus is called $\texttt{Obj}^+$, and is a fully fledged object-based calculus that supports message passing and constructs for object update and extension. $\texttt{Obj}^+$ is a variant of the *Lambda Calculus of Objects* of [FHM94], the first calculus that provided a formal type system for extensible objects. The differences

[2] We assume to have a term constant *null* whose type is *MyType*.

from the original proposal of [FHM94] are as follows: (*i*) method bodies are ς-abstractions over a SELF variable rather than λ-abstractions, and (*ii*) methods are distinguished from fields, both syntactically and semantically. The use of ς-binders eases the comparisons between our calculus and related calculi in the literature (with ς-binders, the syntax of $\texttt{Obj}^+$ is a proper extension of the untyped ς-calculus [AC96a]). As for the distinction between methods and fields, a part from being a common practice in object-oriented languages and calculi, it arises as a result of a retrospective analysis of the interpretation of objects and object types. In fact, as proven in [BBC02], the qualitative nature of the target theory used in the interpretation changes significantly depending on the kind of overrides (otherwise called *updates* in the literature) supported by the source calculus. Specifically, recursive types suffice for the interpretation of *external* and *self-inflicted* field updates, and *external* method updates[3]. On the other hand, *self-inflicted* method updates require a non-trivial extension of the target theory, one in which recursion and least fixed points are available not only for types, but also for type operators.

$\texttt{Obj}^+$ is an untyped version of the calculus $\texttt{Ob}^+$ presented in [BBC02]. $\texttt{Ob}^+$ served the purpose of being the source calculus of a type preserving and computationally adequate interpretation into a functional calculus. $\texttt{Ob}^+$ was explicitly typed for a technical reason: to ensure that well-typed objects had unique types, a property that was missing in [FHM94] and necessary to have a fully formal encoding. We will give an overview of such encoding results in Section 5, but, since we are not going into technical details in the present tutorial, we preferred to use as a tutorial example an untyped calculus, which is easier to describe and work with.

The syntax of $\texttt{Obj}^+$ terms is defined by the following productions:

$$\begin{array}{lll}
a, b, c ::= \textit{Terms} & & \\
\quad x & & \textit{variable} \\
\quad \langle v_i = c_i{}^{i \in I}, m_j = \varsigma(x)b_j\{x\}^{j \in J}\rangle & & \textit{object} \\
 & & (v_i\textit{'s and } m_j\textit{'s distinct}) \\
\quad a.v & & \textit{field selection} \\
\quad a.v \leftarrow b & & \textit{field update} \\
\quad a.v \longleftarrow\!\!\!\!+\; b & & \textit{field addition} \\
\quad a \circ m & & \textit{method invocation} \\
\quad a \circ m \leftarrow \varsigma(x)b\{x\} & & \textit{method update} \\
\quad a \circ m \longleftarrow\!\!\!\!+\; \varsigma(x)b\{x\} & & \textit{method addition}
\end{array}$$

Terms of the form $\langle v_i = c_i{}^{i \in I}, m_j = \varsigma(x)b_j\{x\}^{j \in J}\rangle$ denote objects, i.e., collections of named fields and methods that can be selected, updated, or added. As in [AC96a], each method is an abstraction of the host object SELF, represented by the ς-bound variable x. The notation $b\{x\}$ emphasizes that the variable x may

[3] The adjective *external* refers to invocation/override operations performed on objects, while the adjective *self-inflicted* refers to invocation/override operations performed on SELF, inside a method body.

occur free in b. Given $b\{x\}$, we write $b\{a\}$ (or, equivalently $b\{x := a\}$) to denote the term that results from substituting the term a for every free occurrence of x in b.

Each of the primitive operations on objects come in two versions, for fields and methods. $a \circ m$ invokes the method associated with the label m in a, while $a\boldsymbol{.}v$ selects the field v; $a \circ m \leftarrow \varsigma(x)b\{x\}$ replaces the current body of m in a with $\varsigma(x)b\{x\}$, while $a\boldsymbol{.}v \leftarrow b$ performs the corresponding operation on the field v; finally, $a \circ m \longleftarrow\!\!+ \varsigma(x)b\{x\}$ extends a by adding a new method m with associated body $\varsigma(x)b\{x\}$, and $a\boldsymbol{.}v \longleftarrow\!\!+ b$ does the same with the field v.

Terms that differ only for renaming of bound variables, or for the relative order of method and field labels are considered equal, i.e., syntactically identical: we write $a \equiv b$ to state that a and b are equal. To ease the notation, we use ℓ to denote method or field labels whenever the distinction between methods and fields may be disregarded soundly, and write $a \cdot \ell$ to denote $a \circ \ell$ or $a\boldsymbol{.}\ell$.

The evaluation of $\texttt{Obj}^+$ expressions is defined by a big-step operational semantics that rewrites closed terms into results[4]. A *result* r is defined to be a term in object form. We write $a \Downarrow_o r$ to denote that evaluating a closed term a returns the result r, and say that a converges – written $a \Downarrow_o$ – if there exists a result r such that $a \Downarrow_o r$. The operational semantics is defined below, rule by rule, and it is an extension of the corresponding semantics in [AC96a] that handles field and method addition. The following notation is used in the operational semantics rules:

$$\begin{gathered}
\langle \ell_i = \mathbf{b}_i\{x\}^{i \in I \cup J} \rangle \triangleq \langle v_i = c_i{}^{i \in I}, m_j = \varsigma(x)b_j\{x\}^{j \in J} \rangle \\
a \cdot \ell \triangleq a \circ \ell \textit{ or } a\boldsymbol{.}\ell \\
a \cdot \ell \leftarrow \mathbf{b}\{x\} \triangleq a \circ \ell \leftarrow \varsigma(x)b\{x\} \text{ or } a\boldsymbol{.}\ell \leftarrow b \\
a \cdot \ell \longleftarrow\!\!+ \mathbf{b}\{x\} \triangleq a \circ \ell \longleftarrow\!\!+ \varsigma(x)b\{x\} \text{ or } a\boldsymbol{.}\ell \longleftarrow\!\!+ b
\end{gathered}$$

The rules are rather intuitive, and for the ones dealing with updates and additions we do not need to distinguish among fields and methods.

$$\mathsf{Results:}\quad r = \langle v_i = c_i{}^{i \in I}, m_j = \varsigma(x)b_j\{x\}^{j \in J} \rangle$$

$$(\text{Select}_\text{v})\ \frac{a \Downarrow_o \langle \ldots, v_j{=}c_j, \ldots \rangle \quad c_j \Downarrow_o r}{a\boldsymbol{.}v_j \Downarrow_o r}$$

$$(\text{Select}_\text{m})\ \frac{a \Downarrow_o \widehat{a} \quad b_j\{\widehat{a}\} \Downarrow_o r \quad (\widehat{a} \equiv \langle \ldots, m_j = \varsigma(x)b_j\{x\}, \ldots \rangle)}{a \circ m_j \Downarrow_o r}$$

$$(\text{Update})\ \frac{a \Downarrow_o \langle \ell_i = \mathbf{b}_i\{x\}^{i \in I \cup J} \rangle \quad k \in I \cup J}{a \cdot \ell_k \leftarrow \mathbf{b}\{x\} \Downarrow_o \langle \ell_i = \mathbf{b}_i\{x\}^{i \in I \cup J - \{k\}}, \ell_k = \mathbf{b}\{x\} \rangle}$$

$$(\text{Extend})\ \frac{a \Downarrow_o \langle \ell_i = \mathbf{b}_i\{x\}^{i \in I \cup J} \rangle \quad \ell \notin \{\ell_i\}^{i \in I \cup J}}{a \cdot \ell \longleftarrow\!\!+ \mathbf{b}\{x\} \Downarrow_o \langle \ell = \mathbf{b}\{x\}, \ell_i = \mathbf{b}_i\{x\}^{i \in I \cup J} \rangle}$$

[4] The definition of a small-step operational semantics can be found in Appendix A.3.

Field selection and method invocation (rules (Select$_\mathrm{v}$) and (Select$_\mathrm{m}$)) are computed by evaluating the recipient of the selection/invocation to a result (that is, to an object), then by selecting and evaluating the sought component body. The only difference is that a retrieved method body must have the host object substituted for the SELF variable (represented formally in the calculus by x) before being evaluated, to accomplish with the intended semantics of objects, so to make it possible to invoke sibling methods and access object fields. Updating (rule (Update)) must ensure that the component we want to update exists in the object which the recipient of the updating operation evaluates to, before updating the component. Vice-versa, for the extension (rule (Extend)) we must ensure that the component is not present yet before adding it.

3.1 Formalized Examples

In this section, we present some examples illustrating the behavior of the calculus $\mathtt{Obj}^+$. We start from an expression representing an infinite computation:

$$\Omega \equiv \langle m = \varsigma(x) x \circ m \rangle \circ m$$

This expression reduces to itself forever, therefore never to a result r (i.e., it does not converge), as the following derivation in the big-step semantics shows.

$$(\mathrm{Select_m})\ \frac{\begin{array}{c}\langle m = \varsigma(x) x \circ m \rangle \Downarrow_o \langle m = \varsigma(x) x \circ m \rangle \\ (x \circ m)\{\langle m = \varsigma(x) x \circ m \rangle\} \equiv \Omega \\ (x \circ m)\{\langle m = \varsigma(x) x \circ m \rangle\} \Downarrow_o ??\end{array}}{\Omega \Downarrow_o ??}$$

It is possible to show formally that $\mathtt{Obj}^+$ is Turing-complete (that is, it can codify all partial recursive functions). In [FHM94], an encoding in the Lambda Calculus of Objects of Object numerals, Zero test, Predecessor and fixed-point operator (using the Ω expression shown above, written in their syntax) is presented. We prefer to take the Abadi-Cardelli's approach (presented on pages 66–67 of [AC96a]), that is, encoding the untyped lambda calculus in $\mathtt{Obj}^+$ (we recall that the symbol ':=' denotes a substitution):

$\llbracket\, x \,\rrbracket \equiv x$ (variable);
$\llbracket\, a\ b \,\rrbracket \equiv \llbracket\, a \,\rrbracket \bullet \llbracket\, b \,\rrbracket$ (application), where $p \bullet q \equiv (p{\centerdot}arg \leftarrow q) \circ val$;
$\llbracket\, \lambda x.a\{x\} \,\rrbracket \equiv \langle\ arg = y, val = \varsigma(x) \llbracket\, a\{x\} \,\rrbracket \{x := x{\centerdot}arg\}\ \rangle$ (lambda abstraction), with y a fresh variable.

The reader may want to check that this encoding preserves β-reduction.

Since $\mathtt{Obj}^+$ is Turing-complete, we can add integers as shortcuts for numerals to $\mathtt{Obj}^+$ terms, in order to ease the task of writing examples. An example that we could not left out is the Point/ColorPoint one, expressed in the $\mathtt{Obj}^+$ syntax:

$\mathtt{Point} \equiv \langle\ pos = 0, move = \varsigma(x)(x{\centerdot}pos \leftarrow ((x{\centerdot}pos) + 1))\ \rangle$
$\mathtt{ColorPoint} \equiv \mathtt{Point}{\centerdot}c \longleftarrow\!\!+\ \ = blue$

It is possible to note that a *move* is performed by a self-inflicted field selection ($x.pos$), that returns the value of *pos* which is then incremented by 1, and through an updating of *pos* with the new value via a self-inflicted override ($x.pos \leftarrow (\ldots)$). Let us consider now some computations:

`ColorPoint` reduces in few operational big-steps to the object:
$\langle\, pos = 0, move = \varsigma(x)(x.pos \leftarrow ((x.pos) + 1)), c = blue\, \rangle$;

`ColorPoint`$\circ$*move* reduces in few operational big-steps to the object:
$\langle\, pos = 1, move = \varsigma(x)(x.pos \leftarrow ((x.pos) + 1)), c = blue\, \rangle$.

We advise the reader to perform both of the above reductions in detail. The second one, in particular, shows how the host-object SELF-substitution works.

4 Types and Typing Rules

In this section we describe the type system of the calculus $\texttt{Obj}^+$. The syntax of types is as follows.

$A, B, C ::=$ *Types*
X, U *variable*
$\texttt{pro}(X)\langle\!\langle v_i : C_i{}^{i\in I}, m_j : B_j\{X\}^{j\in J}\rangle\!\rangle$ *object type (v_i's and m_j's distinct)*

An object type $A \equiv \texttt{pro}(X)\langle\!\langle v_i : C_i{}^{i\in I}, m_j : B_j\{X\}^{j\in J}\rangle\!\rangle$ is the type of all the objects with fields v_i ($i \in I$), and methods m_j ($j \in J$). The keyword `pro` [5] binds all free occurrences of X in the $B_j\{X\}^{j\in J}$. When invoked, each field v_i returns a value of type C_i, and each method m_j returns a value of type $B_j\{A\}$, that is the type B_j with every free occurrence of the variable X substituted by the type A itself. This type substitution reflects the *self-substitution* semantics of method invocation. As in the syntax of terms, we use the notation $\texttt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in I}\rangle\!\rangle$ for `pro`-types, whenever there is no reason to distinguish methods from fields. Object types that differ in the order of the component labels, or for the names of bound variables are considered equal: we write $A \equiv B$ to state that the types A and B are equal.

As hinted in Section 2, functional method addition needs a form of *MyType* inheritance, in order to specialize the method types as methods are inherited. The typing rules for $\texttt{Obj}^+$ rely on the form of match-bounded polymorphism that was studied in [BB99] for the *Lambda Calculus of Objects* [FHM94]. Polymorphic types arise in the typing of methods as a result of (*i*) method bodies being dependent on SELF, and (*ii*) the possibility for a method to be invoked on extensions of the object where it was first installed. To ensure sound typing of method invocation, method bodies are typed in a context that assumes the so-called *MyType* for the SELF variable x. In this context, *MyType* is a *match*-bounded type variable representing the types of the objects resulting from *all the*

[5] The notation `pro` was introduced in [FHM94] and it is maintained here for historical reasons.

possible extensions of the host object with new methods and fields. The *matching* relation ($\lessdot\!\!\#$) was introduced in [Bru94]. A full study about matching and its various application other than *MyType* inheritance can be found in [Bru02].

The most significative rule for matching in our context is the following:

$$\text{(Match } \mathtt{pro}\text{)}\qquad \frac{\Gamma \vdash \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in 1..n+k}\rangle\!\rangle}{\Gamma \vdash \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in 1..n+k}\rangle\!\rangle \lessdot\!\!\#\, \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in 1..n}\rangle\!\rangle}$$

This rule superficially looks like the *subtyping-in-width rule*, but the use of matching in place of the more standard relation of subtyping is central to the type system, because matching, unlike subtyping, does not support subsumption: since objects are extensible, absence of subsumption on `pro`-types is crucial for type soundness (see Section 4.2 for a quick overview on the problem, and [BB99] for a deeper discussion).

All of the typing rules for $\mathtt{Obj}^+$ are collected in Appendix A.4. The most interesting rules are given and commented below.

$$\text{(Val Object: } A \equiv \mathtt{pro}(X)\langle\!\langle v_i : C_i{}^{i\in I}, m_j : B_j\{X\}^{j\in J}\rangle\!\rangle\text{)}\qquad \frac{\Gamma \vdash c_i : C_i \qquad \Gamma, U \lessdot\!\!\# A, x : U \vdash b_j\{x\} : B_j\{U\} \qquad \forall\, i \in I, j \in J}{\Gamma \vdash \langle v_i = c_i{}^{i\in I}, m_j = \varsigma(x) b_j\{x\}^{j\in J}\rangle : A}$$

(Val Object) is the rule for object formation. *MyType* is the type variable U match-bounded in the context of the typing judgements of method bodies, and it is given to the self variable x as its type ($x{:}U$). Note that the return type of each method depends polymorphically on *MyType*: this allows the (return) type of methods to be soundly specialized upon method addition. The rule also emphasizes the distinction between methods and fields: since the latter do not depend on self, their type need not depend on *MyType*.

$$\text{(Val Select)}\qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash A \lessdot\!\!\#\, \mathtt{pro}(X)\langle\!\langle \ell : B\{X\}\rangle\!\rangle}{\Gamma \vdash a \cdot \ell : B\{A\}}$$

(Val Select) is the rule for field and method invocation. An invocation for (the field or) method ℓ on an object a requires a to have a `pro`-type containing the label ℓ. The result of the call has the type B listed in the `pro`-type of a, with A substituted for X (if ℓ is a field, this substitution is vacuous). Note that A may either be a `pro`-type matching $\mathtt{pro}(X)\langle\!\langle \ell : B\{X\}\rangle\!\rangle$, or else an unknown type (i.e., a type variable) occurring (match-bounded) in the context Γ. Rules like the one above are sometimes referred to as *structural rules* [AC96a], and their use is critical for an adequate rendering of *MyType* polymorphism: it is the ability to refer to possibly unknown types that allows methods to act parametrically over any $U \lessdot\!\!\# A$, where U is the type of self, and A is a given `pro`-type.

$$\text{(Val Field Addition: } A^+ \equiv \mathtt{pro}(X)\langle\!\langle \ell : B\{X\}, \ell_i : B_i\{X\}^{i\in I} \rangle\!\rangle\text{)}$$

$$\frac{\varGamma \vdash a : \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in I} \rangle\!\rangle \quad \varGamma \vdash c : B \quad (\ell \neq \ell_i \ \forall i \in I)}{\varGamma \vdash a{\scriptstyle\bullet}\ell \longleftarrow\!\!\!+\; c : A^+}$$

$$\text{(Val Method Addition: } A^+ \equiv \mathtt{pro}(X)\langle\!\langle \ell : B\{X\}, \ell_i : B_i\{X\}^{i\in I} \rangle\!\rangle\text{)}$$

$$\frac{\begin{array}{c}\varGamma \vdash a : \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in I} \rangle\!\rangle \quad \varGamma, U \mathrel{<\!\#} A^+, x : U \vdash b\{x\} : B\{U\} \\ (\ell \neq \ell_i \ \forall i \in I)\end{array}}{\varGamma \vdash a \circ \ell \longleftarrow\!\!\!+\; \varsigma(x)b\{x\} : A^+}$$

(Val Field Addition) and (Val Method Addition) are the typing rules for field and method additions. The label ℓ is assumed to be different from all of the ℓ_i's, $i \in I$, and the type of the object a being extended to be a `pro`-type: since no subtyping is available on `pro`-types, this implies that an object extension is typed with *exact* knowledge of the type of a. Note that in rule (Val Method Addition) the return type of the added method depends polymorphically on *MyType*, while in the rule (Val Field Addition) the field type is not required to depend on *MyType*.

$$\text{(Val Field Update)}$$

$$\frac{\varGamma \vdash a : A \quad \varGamma \vdash A \mathrel{<\!\#} \mathtt{pro}(X)\langle\!\langle v : C \rangle\!\rangle \quad \varGamma \vdash c : C}{\varGamma \vdash a{\scriptstyle\bullet}v \leftarrow c : A}$$

(Val Field Update) is a structural rule: as in (Val Select), the type A of the object a being updated may either be a type variable, or a `pro`-type. When it is a `pro`-type, the update is *external*; when it is a type variable, the update is *self-inflicted*. The judgement $\varGamma \vdash A \mathrel{<\!\#} \mathtt{pro}(X)\langle\!\langle v : C \rangle\!\rangle$ requires A (hence $a : A$) to have a field v with type C, and the remaining judgement ensures that the update preserves the type of the object.

$$\text{(Val Method Update (external only): } A \equiv \mathtt{pro}(X)\langle\!\langle v_i : {C_i}^{i\in I}, m_j : B_j\{X\}^{j\in J} \rangle\!\rangle\text{)}$$

$$\frac{\varGamma \vdash a : A \quad \varGamma, U \mathrel{<\!\#} A, x : U \vdash b\{x\} : B_k\{U\} \quad k \in J}{\varGamma \vdash a \circ m_k \leftarrow \varsigma(x)b\{x\} : A}$$

(Val Method Update) handles the case of updates for methods: unlike the corresponding rule for fields, (Val Method Update) is non-structural, as the type A of the object being updated is required to be a `pro`-type. As a consequence, method updates may *not* be self-inflicted; instead, it is available as an *external* operation which can only be performed from outside the object. This restriction could safely be lifted, without consequences on the operational behavior of the source calculus or on the operational soundness of the type system. The rule accounting both for external and self-inflicted method updates would have the following form:

(Val Method Update' (external and self-inflicted))

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \lessdot\# \, \mathtt{pro}(X)\langle\!\langle m_k : B_k\{X\}\rangle\!\rangle \quad \Gamma, U \lessdot\# \, A, x : U \vdash b\{x\} : B_k\{U\}}{\Gamma \vdash a \circ m_k \leftarrow \varsigma(x)b\{x\} : A}$$

On the one hand, type soundness in the presence of both external and self-inflicted method updates can be proved as in the calculus described in [BB99]. On the other hand, as we mentioned, self-inflicted method updates do have significant impact on the semantic interpretation, so, since we want to account for the semantics by encoding (in Section 5), is worth to present the two rules separately.

We conclude the presentation of the source calculus stating the main properties of the type system. Proofs for the results below are essentially the same as those given in full detail in [BB99].

Theorem 1 (Subject Reduction). *If $\Gamma \vdash a : A$ in $\mathtt{Obj}^+$, and $a \Downarrow_o r$, then also $\Gamma \vdash r : A$.*

Theorem 2 (Soundness). *Let c be a closed expression such that $\varnothing \vdash c : A$ for some type A. Then:*

1. *if c is one of: $a \circ \ell \leftarrow \varsigma(x)b\{x\}$, $a_{\centerdot}\ell \leftarrow b$, $a \cdot \ell$, and $a \Downarrow_o r$, then r is an object containing a label ℓ.*
2. *if c is one of: $a \circ \ell \longleftarrow\!\!+ \varsigma(x)b\{x\}$, $a_{\centerdot}\ell \longleftarrow\!\!+ b$, and $a \Downarrow_o r$, then r is an object that does not contain a label ℓ.*

The soundness theorem states that if an expression is well-typed, no *message-not-understood* error will arise at run-time.

4.1 Some Examples of Typing

As we added integers to the $\mathtt{Obj}^+$ terms, we now consider also their type $\mathtt{int}$ to type the Point/ColorPoint example:

$$\mathtt{Point} \equiv \langle\, pos = 0, move = \varsigma(x)(x_{\centerdot}pos \leftarrow_X ((x_{\centerdot}pos) + 1))\,\rangle \ : \ \mathtt{pro}(X)\langle\!\langle pos{:}\mathtt{int}, move{:}X\rangle\!\rangle;$$

$$\mathtt{ColorPoint} \equiv \mathtt{Point}_{\centerdot}c \longleftarrow\!\!+ \ = \mathtt{blue} \ : \ \mathtt{pro}(X)\langle\!\langle pos{:}\mathtt{int}, move{:}X, c{:}\mathtt{color}\rangle\!\rangle$$

The type derivation is left to the reader as an exercise. We will show, instead, the type derivation for $\langle\, m = \varsigma(x)x \circ m\,\rangle \ : \ \mathtt{pro}(X)\langle\!\langle m{:}X\rangle\!\rangle$ ($\langle\, m = \varsigma(x)x \circ m\,\rangle$ is part of Ω, presented in Section 3.1). We need basically[6] a (Val Object) application and a (Val Select) application (from bottom to top).

[6] We omit checking well-formedness for $x : U$ and $U \lessdot\# \, \mathtt{pro}(X)\langle\!\langle m{:}X\rangle\!\rangle$ in the derivation.

$$\dfrac{\dfrac{x : U \vdash x : U \quad U \lessdot\!\!\# \, \texttt{pro}(X)\langle\!\langle m{:}X \rangle\!\rangle \vdash U \lessdot\!\!\# \, \texttt{pro}(X)\langle\!\langle m{:}X \rangle\!\rangle}{U \lessdot\!\!\# \, \texttt{pro}(X)\langle\!\langle m{:}X \rangle\!\rangle, x : U \vdash x \circ m : X\{U\} \equiv U}}{\mathit{\Gamma} \vdash \langle\ m = \varsigma(x)x \circ m\ \rangle : \texttt{pro}(X)\langle\!\langle m{:}X \rangle\!\rangle}$$

4.2 Subtyping

A concept of paramount importance in the object-oriented realm is *subtyping*: A is a subtype of B, written $A <: B$, iff a value of type A can be used in any context expecting a value of type B. Connected to the notion of subtyping is the *subsumption* rule: if P:A and $A <: B$, then P:B. If $A <: B$, a value of type A can be: (i) used for a parameter of type B; and (ii) assigned to a variable of type B, so that the value of an expression can correspond to a subtype of its static type. We discuss briefly subtyping for the most common structures used in the object-oriented setting, and then some anomalies by showing three examples.

Subtype for records (co-variant) If $r : \{l_i : B_i\}^{i \in 1..k}$ then $r.l_i : B_i$. When is $\{l_j : A_j\}^{j \in 1..n} <: \{l_i : B_i\}^{i \in 1..k}$? Assume $r' : \{l_j : A_j\}^{j \in 1..n}$. We still need $r'.l_i : B_i$, $i \in 1..k$. Therefore, $\{l_j : A_j\}^{j \in 1..n} <: \{l_i : B_i\}^{i \in 1..k}$ if $k \leq n$ (*subtyping-in-width*) and $\forall\ i \in 1..k, A_i <: B_i$ (*subtyping-in-depth*).

Subtype for functions (co/contra-variant) If $F : A \to B$ and $a : A$ then $F(a) : B$. When is $A' \to B' <: A \to B$? If $F' : A' \to B'$, we still need $F'(a) : B$. Therefore, $A' \to B' <: A \to B$ if $A <: A'$ and $B' <: B$, i.e., subtyping behaves contra-variant for parameters and co-variant for results. This rather counter-intuitive behavior is source of "troubles" when subtyping on function types, obviously applied to methods, meets inheritance.

Example 1: inheritance is not subtyping [CHC90]. Can inherited methods break when called on "subobjects"? Unfortunately, yes. Let us consider this fragment of pseudo-code that we assume inserted in a program `let`-defining two $Node$'s called $nd1, nd2$ and a $DbleNode$ called dnd ($Node$ and $DbleNode$ where introduced in Section 2).

```
...in
   let $Break \equiv \langle\ breakit(n1, n2) = n1.attachRight(n2)\ \rangle$
       in /* main */$Break.breakit(nd1, nd2); Break.breakit(dnd, nd1)$
```

The first call $Break.breakit(nd1, nd2)$ is safe, but a problem arises from the second call $Break.breakit(dnd, nd1)$ because it triggers $dnd.attachRight(nd1)$, which in turn tries to set the non-existing $prev$ of $nd1$, making the program crash (since the invoked $attachRight$ is the one of $DbleNode$). This happens being $DbleNode$ not a subtype of $Node$, because of $MyType$ appearing in a contra-variant position in $attachRight$ (i.e., as the type of its parameter)[7].

We can conclude that $\mathtt{pro}(X)\langle\!\langle m_j : A_j\{X\}^{j\in 1..n}\rangle\!\rangle <: \mathtt{pro}(X)\langle\!\langle m_i : B_i\{X\}^{i\in 1..k}\rangle\!\rangle$ if and only if $k \leq n$ and for all $i \in 1..k$, $A_i\{X\} <: B_i\{X\}$, *but* no $B_i\{X\}$ has a contra-variant occurrence of X, that is, no parameters in the methods of the supertype can have the same type $MyType$ of SELF.

Therefore, if two object types are subtypes, then they match, but not vice-versa. For example (using the usual informal notation), $DbleNodetype \mathrel{<\!\!\#} Nodetype$ but not $DbleNodetype <: Nodetype$. For a careful study of the conflict between inheritance and subtyping, see the work of Castagna et al. (for instance, in [Cas96]), where, among other things, another approach for object-oriented languages is presented, alternative to the "objects-as-records" one: the "methods-as-overloaded-functions" model.

Example 2: subtyping-in-width versus method/field addition. In an object with two components ℓ_1 and ℓ_2 of types A_1 and A_2, the component ℓ_1 may require ℓ_2 to be of type A_2. "Forgetting" ℓ_2 by subtyping may result in a possible re-addition of ℓ_2 with another, incompatible, type A_3, making the invocation of ℓ_1 fail (even though the whole expression is well-typed).

Example 3: subtyping-in-depth versus method/field override. Consider the following object:

$$\mathtt{UseLog} \equiv \langle\, n = 10, m = \varsigma(x)Log(x.n)\,\rangle : \mathtt{pro}(X)\langle\!\langle n : posint, m : int\rangle\!\rangle$$

The method m selects the value of the field n and calculates its base-10 logarithm. Now, by subtyping, $posint <: int$, so we may have $\mathtt{UseLog} : \mathtt{pro}(X)\langle\!\langle n : int, m : int\rangle\!\rangle$, which would allow to update n with a negative value. This clearly would make the invocation of m fail (even though the whole expression is well-typed).

In the literature, there are two main ways to make subtyping and operations on objects living together harmlessly.

- Limiting subtyping, for instance by collecting the mutual dependencies among methods and allowing subtyping applications only if they involve methods that are not used by other methods [BL95,Rém98].
- Defining two states in which an object can be. In *state 1*, objects can be extended/overridden (i.e., in this state, objects play a rôle similar to the one of classes), while in *state 2*, objects can be subtyped.

The solution of [FM95] belongs to the second "family", and it is formalized by the following rule:

(Sub `probjFM95`)

$$\frac{\Gamma, Y, X <: Y \vdash B'_i\{X\} <: B_i\{Y\} \qquad (i = 1..n)}{\Gamma \vdash \mathtt{probj}(X)\langle\!\langle \ell_i : B'_i\{X\}^{i\in 1..n+k}\rangle\!\rangle <: \mathtt{obj}(Y)\langle\!\langle \ell_i : B_i\{Y\}^{i\in 1..n}\rangle\!\rangle}$$

[7] These problems are well-known in the recursive type setting, and $MyType$ is nothing else than a form of recursive type.

When an object is in its `pro`-totypical state, it can be modified by override and addition, but subtyping does not apply. When an object changes state, becoming a “true” `obj`-ect, it is as “sealed” and cannot be modified anymore, but then subtyping (both in-width and in-depth, as the above rule shows) applies. The drawback of this approach is that, since the passage from the `pro`-state to the `obj`-state is done by using the same subtyping rule, when an object contains a binary method it cannot be “promoted” from `pro`- to `obj`-, otherwise we run into the problem described in the Example 1 above.

5 Semantics by Encoding

Interpretations of object-oriented programming are typically defined in terms of reductions to procedural or functional programming, and help provide sound and formal foundations to object-oriented languages and their specific constructs and techniques. The reduction is not straightforward: difficulties arise principally at the level of types, when trying to validate the subtyping properties of the source languages. A number of object encodings for the so-called *object-based* calculi have subsequently been proposed by [PT94,AC96a,ACV96,BCP97], and recently by [Cra99]. These interpretations apply to a rich variety of object calculi with constructs for object formation, message send and (functional) method update: they succeed in validating the operational semantics of these calculi as well as the expected subtyping relationships over object types; finally they extend smoothly to the case of *Self Types* and other object-oriented constructs. None of these proposals, however, appears to scale to calculi of extensible objects, where there are two major difficulties in dealing with their interpretations. The first is the presence of the *MyType*. The second difficulty arises from the co-existence of subtyping and object extension, two mechanisms that we have seen being essentially incompatible in Section 4.2, and hence difficult to combine in sound and flexible type systems. Summarizing, the fact that inheritance and client-use of the objects components happen both at the same level, that is, on the same entities (i.e., the objects), complicates the semantics a great deal.

In this section, we summarize the paper [BBC02], where an interpretation of extensible objects is presented that addresses both these problems. In particular, subtyping is accounted for by distinguishing between extensible and nonextensible objects, as proposed by [FM95] (their main rule is (Sub `probjFM95`), presented in Section 4.2). The interpretation is an encoding: the target calculus is a polymorphic λ-calculus with records, recursive types and (higher-order) subtyping. Within this calculus, an extensible object is interpreted as a pair of two components: the object *generator*, which is made available to contexts where the structure of the object is extended with new methods or fields, and the *interface*, a recursive record that provides direct access for the object’s clients to the methods and fields of the object itself. Technically, the two components are collectively grouped into a single recursive record by a technique which is inspired by, and generalizes, the *split-method* interpretation of [ACV96]. The resulting interpretation is faithful to the source calculus in that (i) it preserves the validity of typing judgements, and (*ii*) it validates the operational semantics, i.e., the encoding is computationally adequate. Besides providing a fully formal interpretation of extensible objects, the encoding of [BBC02] also clarifies the relationship between calculi of extensible and nonextensible objects presented in the recent literature. In fact, the encoding specializes smoothly to the case of nonextensible objects and object types, validating the expected subtyping relationships. Although the focus is one particular calculus – specifically, on one approach to combining object extension with subtyping – the translation is sufficiently general to capture other notions of subtyping over object types (a notable example are the rules for covariant subtyping of [AC96a]).

5.1 The Target Calculus $F_{\omega<:\mu}$

The target calculus of the translation is a variant of the polymorphic typed λ-calculus $F^{\omega}_{<:}$. We briefly review the syntax, introducing notation and terminology on type operators and recursive types.

$\mathtt{K} ::=$	*Kinds*	
	$\mathtt{T}$	type
	$\mathtt{K} \Rightarrow \mathtt{K}$	type operator

$\mathsf{A}, \mathsf{B} ::=$	*Constructors*	
	X	constructor variable
	$\mathtt{Top}$	greatest constructor of kind $\mathtt{T}$
	$\mathsf{A} \rightarrow \mathsf{B}$	function type
	$[m_1 : \mathsf{B}, \ldots, m_k : \mathsf{B}]$	record type
	$\forall(X <: \mathsf{A} :: \mathtt{K})\mathsf{A}$	bounded universal type
	$\mu(X)\mathsf{A}$	recursive type
	$\lambda(X :: \mathtt{K})\mathsf{B}$	operator
	$\mathsf{B}(\mathsf{A})$	operator application

$M, N ::=$	*Expressions*	
	x	variable
	$\lambda(x : \mathsf{A})M$	abstraction
	$M\,N$	application
	$\Lambda(X <: \mathsf{A} :: \mathtt{K})e$	type-abstraction
	$M\,\mathsf{A}$	type-application
	$[m_1 = M_1, \ldots, m_k = M_k]$	record
	$M.m$	record selection
	$\mathtt{fold}(\mathsf{A}, M)$	recursive fold
	$\mathtt{unfold}(M)$	recursive unfold
	$\mathtt{let}\ x = M\ \mathtt{in}\ N$	local definition
	$\mathtt{letrec}\ f(x : \mathsf{A}) : \mathsf{B} = M\ \mathtt{in}\ N$	recursive local definition

A type operator is a function from types to types. Types and type operators are collectively called *constructors*. The notation $\mathsf{A} :: \mathtt{K}$ indicates that the constructor A has kind $\mathtt{K}$. The typing rules, found in the paper [BBC02], are standard (see also Chapter 20 of [AC96a]). Type equality is defined by judgements of the form $\Gamma \vdash \mathsf{A} \leftrightarrow \mathsf{B}$, modulo renaming of bound variables. The following notation is used throughout: $Op \equiv \mathtt{T} \Rightarrow \mathtt{T}$ is the kind of type operators, $\mathsf{A} \leq \mathsf{B}$ denotes subtyping over type operators, whereas $\mathsf{A} <: \mathsf{B}$ denotes subtyping over the kind $\mathtt{T}$ of types. We also use the following shorthands from [AC95] to emphasize the relationships between type operators and their fixed points. Given the type operator $\mathsf{A} :: Op$, $\mathsf{A}^* \equiv \mu(X)\mathsf{A}(X)$ is the (least) fixed point of A; dually, given the recursive type $\mathsf{A} :: \mathtt{T} \equiv \mu(X)\mathsf{B}(X)$, $\mathsf{A}^{\mathrm{OP}} \equiv \lambda(X)\mathsf{B}(X) :: Op$ is the type operator whose (least) fixed point is A. As in [AC95], A^{OP} is defined in terms of the syntactic form $\mu(X)\mathsf{B}(X)$ of A: the notation A^{OP} is well-defined because we rely on a weak notion of type equality whereby a recursive type is isomorphic, rather than equal, to its unfoldings. Results, or values, are lambda abstractions, records, and recursive folds. We write $M \Downarrow_f r$ to denote that M (a closed term) evaluates to a result r, and say that M converges – written $M \Downarrow_f$ – if there exists a result r such that $M \Downarrow_f r$. A standard call-by-name operational semantics for this target calculus can be found in [BBC02].

5.2 Encoding of Extensible Objects and Types: an Overview

To understand the encoding and its subtleties, it is useful to proceed by steps, and first discuss solutions that are intuitively simple but do not give a correct encoding. We keep the discussion informal, and look at a simplified case in which objects have no fields, and for which the only available operators are method addition and invocation. Then we extend the analysis to objects with fields, and show how to account for field selection, addition and update. Finally we look at method updates, distinguishing *external* from *self-inflicted* updates: we show that the former can be encoded in $F_{\omega<:\mu}$ with no additional machinery, and discuss the extensions to $F_{\omega<:\mu}$ required to handle the latter. The paper [BBC02] presents the fully-formal encoding.

Failures of Self Application. We first consider objects without fields. Looking at the reduction rules, it would be tempting to interpret these objects as in the *self-application* semantics of [Kam88]. In this semantics, methods are functions of the SELF parameter, objects are records of such functions, and method invocation is field selection plus self-application. This semantics was originally proposed as an interpretation of nonextensible objects, and its properties are well-known [AC96a]: it works well in the untyped case, but fails in the typed case because it does not validate the expected subtyping relationships over object types.

A similar problem arises for our extensible objects, even though `pro`-types can *not* be subtyped. Following the self-application semantics, one would interpret the type $\mathtt{pro}(X)\langle\!\langle m{:}B\rangle\!\rangle$ as the recursive record type $A \equiv \mu(X)[m : X \rightarrow B]$ which solves the type equation $A = [m : A{\rightarrow}B]$. Now, given (the interpretation of) an object $a : A$, consider extending a with (the interpretation of) a new method $m' = \lambda(s)b$. The extension is interpreted as the formation of the new record $[m = a.m, m' = \lambda(s)b]$, whose type is $A^+ \equiv \mu(X)[m : X{\rightarrow}B, m' : X \rightarrow B']$. Typing the new record requires the type of $a.m$ to be subsumed to $A^+ \to B$, but the subtype relationship $A{\to}B <: A^+{\to}B$ fails due to the contravariant occurrence of the types A and A^+.

To circumvent the use of subsumption, a seemingly correct solution would be to refine the self-application semantics by using polymorphic methods and interpreting `pro`-types as recursive types of the form $\mu(X)[m : \forall(U \mathrel{\lessdot\!\!\#} X)U{\to}B\{U\}]$. Unfortunately this attempt fails when trying to reduce matching to subtyping in $F_{\omega<:\mu}$, as in [AC95]: the reason is essentially the same as before, as the universal quantifier is again contravariant in its bound.

In both the previous attempts, the actual source of the problem is the poor interaction between the subtyping rules for recursive types and the contravariant occurrence of the recursion variable in the types of methods: we need to break that problematic dependency.

Methods and Split Labels. Looking at the typing rules of $\texttt{Obj}^+$ (cf. Section 4), one may identify two distinguished views of methods. The rule (Val Object) shows this distinction clearly: in the premises, methods are viewed and typed as concrete values – abstractions of SELF – whereas in the conclusion they are seen as "abstract services" that can be invoked by messages. This observation suggests an interpretation that splits methods into two parts, in ways similar to, but different from, the translation of [ACV96] (see [BBC02]). In the untyped case, the object[8] $\langle m_i = \varsigma(x)b_i{}^{i\in 1..n}\rangle$ can be interpreted as the recursive record that satisfies the following equation:

$$M = [m_i^{gen} = \lambda(x)\,[\![\, b_i \,]\!]\;^{i\in 1..n},\; m_i^{sel} = M.m_i^{gen}(M)\;^{i\in 1..n}]$$

Each method m_i is represented by two components: the *generator* m_i^{gen}, associated with a function representing the actual body of m_i, and the *selector* m_i^{sel}, which results from self-applying m_i^{gen} to the host object and can be directly invoked by selection, without self-application at selection time. Thus, *clients* of the object can access the object's methods by means of the selectors, while *derived objects*, obtained by the addition of new methods, inherit the generators and re-install the corresponding selectors to rebind SELF to the extended structure of the host object. In other words, the set of selectors can be thought of as the *abstract interface* that the object provides for its clients, while the generators are available in contexts where the structure of the object needs to be extended with new methods.

This idea works well also in the typed case. The interface associated with an $\texttt{Ob}^+$ type $A \equiv \texttt{pro}(X)\langle\!\langle m_i : B_i\{X\}^{i\in 1..n}\rangle\!\rangle$ is represented by the type operator A^{IN} defined by $\mathsf{A}^{\text{IN}}(X) \equiv [m_i^{sel} : \mathsf{B}_i\{X\}^{i\in 1..n}]$, that includes the method selectors (here, and below, B_i is the translation of B_i). The type A, in turn, is interpreted as the recursive record type that collects the components representing generators

[8] The form $\langle m_i = \varsigma(x)b_i{}^{i\in 1..n}\rangle$ is a shortcut notation for the object expression $\langle m_i = \varsigma(x)b_i\{x\}^{i\in 1..n}\rangle$.

and selectors for each of the m_i:

$$\mathsf{A} \equiv \mu(X)[m_i^{gen} : \forall(U \leq \mathsf{A}^{\text{IN}})U^*{\rightarrow}\mathsf{B}_i\{U^*\}\ ^{i\in 1..n}, m_i^{sel} : \mathsf{B}_i\{X\}\ ^{i\in 1..n}].$$

The generators have polymorphic types corresponding to the match-bounded types used in the typing rules of the source calculus: following [AC95], matching is interpreted as higher-order subtyping.

The typed translation of terms derives immediately from the untyped translation and the translation of types. The object $\varsigma(X{=}A)\langle m_i = \varsigma(x : X)b_i\{X\}^{i\in 1..n}\rangle$ is interpreted as the following recursive record, where A^{OP} is the type operator corresponding to A:

$$M = [m_i^{gen} = \Lambda(U \leq \mathsf{A}^{\text{IN}})\lambda(x : U^*)\,[\![\, b_i\{U\} \,]\!]\ ^{i\in 1..n},\ m_i^{sel} = M.m_i^{gen}(\mathsf{A}^{\text{OP}})(M)^{i\in 1..n}]$$

Exposing the interface A^{IN} in the (higher-order) subtype constraint $U \leq \mathsf{A}^{\text{IN}}$ of the generators insures that each method may legally invoke its sibling methods via SELF. The use of the interface A^{IN} in the bounded quantifier is critical for well-typedness: besides exposing the selectors for use within each method body, it validates the subtyping relationships

$$\forall(U \leq \mathsf{A}^{\text{IN}})U^*{\rightarrow}\mathsf{B}_i\{U^*\} \leq \forall(U \leq (\mathsf{A}^{+})^{\text{IN}})U^*{\rightarrow}\mathsf{B}_i\{U^*\}$$

needed to inherit the generators upon object extension, as well as the relationship $\mathsf{A}^{\text{OP}} \leq \mathsf{A}^{\text{IN}}$ needed to type the self-application $M.m_i^{gen}(\mathsf{A}^{\text{OP}})$. Note also that the interface hides the generators to reflect that objects cannot be self-extended.

Fields and Field Update. Fields are handled easily in the interpretation: they need no generator components *gen*, as they do not depend on SELF, and their evaluation is independent of the structure of the object and of its extensions. On the other hand, field updates require a different treatment (and interpretation) of the recursive nature of SELF. The problem is well-known [AC96a]: defining objects by *direct* recursion, as we did above, does not quite reflect their computational behavior. Specifically, field updates do not work if the recursion freezes SELF to be the object at the time of creation or extension: subsequent updates on a field are not reflected in the invocation of a method that depended on that field through SELF. The solution, as in [ACV96,AC96a], is to give a recursive definition not of the object itself, but rather of the dependency of the object on its methods. In the untyped case, this correspond to the following interpretation of $\langle m_i = \varsigma(x)b_i{}^{i\in 1..n}\rangle$.

```
letrec $mkobj(f_1, \ldots, f_n)$ =
    $[m_i^{gen} = f_i{}^{i\in 1..n},\ m_i^{sel} = f_i(mkobj(f_1, \ldots, f_n))\ ^{i\in 1..n}]$
in $mkobj(\lambda(x)\,[\![\, b_1 \,]\!], \ldots, \lambda(x)\,[\![\, b_n \,]\!]\,)$
```

Now it is the definition of *mkobj*, i.e., of the function that creates the object, that is recursive, not the object itself. This enables a correct interpretation of field updates that uses the *updaters* of [ACV96]. The interpretation of a, now complete, object of the form $\langle v_i = c_i{}^{i\in 1..n}, m_j = \varsigma(x)b_j{}^{j\in 1..m}\rangle$ can be defined as follows:

$$
\begin{array}{l}
\texttt{letrec}\ mkobj(w_i{}^{i\in 1..n}, f_j{}^{j\in 1..m}) = \\
\qquad\quad [v_i^{sel} = w_i{}^{i\in 1..n}, \\
\qquad\quad\ \ v_i^{upd} = \lambda(z)mkobj(w_1,\ldots,w_{i-1},z,w_{i+1},\ldots,w_n,f_j{}^{j\in 1..m})^{i\in 1..n}, \\
\qquad\quad\ \ m_j^{gen} = f_j{}^{j\in 1..m}, \\
\qquad\quad\ \ m_j^{sel} = f_j(mkobj(w_i{}^{i\in 1..n}, f_j{}^{j\in 1..m}))\ {}^{j\in 1..m}] \\
\texttt{in}\ mkobj(\,[\![\, c_i \,]\!]\,{}^{i\in 1..n}, \lambda(x)\,[\![\, b_j \,]\!]\,{}^{j\in 1..m})
\end{array}
$$

Fields are also split into two components: the selector v^{sel} provides access to the contents of the field, the updater v^{upd} takes the new value and returns a new object with the value installed in place of the original. A field update may then be translated by a simple call to the updater associated with that field.

The translation extends smoothly to the typed case: to allow field selection and update, the interface and the type of the object are extended with new components corresponding to the selectors and updaters associated with the object's fields. If $A \equiv \texttt{pro}(X)\langle\!\langle v : C, \ldots\rangle\!\rangle$, the type of the selector v^{sel} is C, and the type of the updater v^{upd} is $\mathsf{C} \to \mathsf{A}$, that is the type of a function that, given an argument with the same type as the value to be updated, returns an object which has the same type as the object prior to the update.

Method Update. In the untyped case, method updates can be dealt with in exactly the same way as field updates, by introducing an updater m_i^{upd} for each method, and interpreting a method update as a call to the corresponding updater. Unfortunately, the typing of method updaters poses a non-trivial problem.

Self-inflicted updates. Take $A \equiv \texttt{pro}(X)\langle\!\langle m : B\{X\}\rangle\!\rangle$ and $a : A$, and consider updating the method m of a. Using method updaters, the translation of A would be the recursive type:

$$
\begin{array}{rl}
\mathsf{A} = \mu(X)[& m^{gen} : \forall(U \le \mathsf{A}^{\text{IN}})U^* {\to} \mathsf{B}\{U^*\}, \\
& m^{upd} : (\forall(U \le \mathsf{A}^{\text{IN}})U^* {\to} \mathsf{B}\{U^*\}) \to X, \\
& m^{sel} : \mathsf{B}\{X\}]
\end{array}
$$

As in the case of fields, the updater m^{upd} expects an argument of the same type as the actual method body — the type of m^{gen} — and returns a modified copy of the object, preserving the original type. The problem is that to allow self-inflicted method updates the updaters must be exposed in the interface A^{IN}. This leads to a new definition of the interface associated to the type A, as the type operator that satisfies the following equation:

$$
\mathsf{A}^{\text{IN}}(X) = [m_i^{upd} : (\forall(U \le \mathsf{A}^{\text{IN}})U^* {\to} \mathsf{B}_i\{U^*\}) {\to} X,\ m_i^{sel} : \mathsf{B}_i\{X\}]
$$

The problem with this equation is that it involves type operators rather than types: solving equations of this kind requires a significant extension to the target theory $F_{\omega<:\mu}$, one that allows fixed points to be taken not only at types, but also at type operators. To our knowledge, this extended theory has not been studied in the literature, and its soundness is still an open problem. For this reason, in the formal treatment [BBC02], we disregard method updaters, and focus attention on the simpler case in which method update is an operation that may only be performed from outside the object. External updates, which are legal in the source calculus, can still be accounted for in $F_{\omega<:\mu}$, as we discuss below.

External updates. The translation of external method updates relies on the same technique discussed for method addition. Given the object $a \equiv \langle m_i = \varsigma(x)b_i{}^{i\in 1..n}\rangle$, the interpretation of $a \circ m_j \leftarrow \varsigma(x)b$ is the call

$$mkobj(\llbracket\, a \,\rrbracket .m_1{}^{gen}, \ldots, \llbracket\, a \,\rrbracket .m_{j-1}^{gen}, \lambda(x) \llbracket\, b \,\rrbracket \,, \llbracket\, a \,\rrbracket .m_{j+1}^{gen}, \ldots, \llbracket\, a \,\rrbracket .m_n{}^{gen})$$

which forms a new object containing new body for m_j and the methods inherited from a. Note that this interpretation of method updates does *not* work for the self-inflicted case. As we already pointed out, the generators m_i^{gen} cannot be invoked from within a method body, as they are not exposed by the interface of the object: on the other hand, exposing the generators in the interface (i.e. using A^{OP} in place of A^{IN}) would break the subtyping required to type an object extension (for $(\mathsf{A}^{+})^{\text{OP}} \leq \mathsf{A}^{\text{OP}}$ fails due to the contravariant occurrence of the universal quantifier in the type of the generators).

6 From Extensible Objects to Classes

The main insight of the object-based model is that class-based notions need not to be assumed, but instead they can be emulated by more primitive notions. Moreover, these more primitive notions can be combined in more flexible way than in a strict class discipline. Therefore, one way to evaluate object-based calculi is with respect to how well they support class-based programming. The main contribution of the paper [BF98] is the formal study of a calculus of encapsulated extensible objects (that uses bounded existential quantifier), which are used to model class hierarchies. This calculus is imperative, and simplifies the type system for the calculus described in Fisher's dissertation [Fis96] (which is functional and supporting *MyType*). The paper [BF98] presents an (imperative) operational semantics, and gives a sound and complete typing algorithm. We summarize the paper in this tutorial and, to motivate the study, we report a comparison between this approach to modeling classes and the well-known *record-of-premethods* approach. This comparison, which is an overview of [FM98], reveals why extensible calculi are relevant to class-based programming.

6.1 Premethod Model

In the context of object calculi, it seems natural to define inheritance using *premethods*, functions that are written with the intent of becoming object methods, but which are not yet installed in any object. Premethods are functions that explicitly depend on the "object itself," typically assumed to be the first parameter to the function. Following this idea, Abadi and Cardelli encoded classes in a pure object system using records of premethods [AC96a]; these ideas are also used by Reppy and Riecke [RR96]. In this approach, a class is an object that contains a record of premethods and a constructor function used to package these premethods into objects.

The primary advantage of the record-of-premethods encoding is that it does not require a complicated form of object. All that is needed is a way of forming an object from a list of component definitions. However, this approach has some serious drawbacks. We discuss these drawbacks using a list of criteria [FM98] that characterizes the rôle of classes in class-based languages.

Does the class construct provide a coherent, extensible collection? The combination of a record of premethods and a constructor function may be thought of as a coherent, extensible collection. Because premethods are simply fields in a record, nothing requires that they be coherent until a constructor function is supplied. Since the constructor function installs the premethods into an object, however, the fact that a given constructor is typable implies that the premethods it uses are coherent. Notice, however, that nothing requires a given constructor to mention all of the premethods in a given premethod record.

Does the class construct guarantee initialization? In more elaborate record-of-premethod models, the code to initialize private instance variables is guaranteed to run if any of the associated premethods is installed into an object. However, constructor functions cannot be reused usefully in derived classes. A consequence is that if a class designer puts initialization code into a class constructor, that code will not be executed for derived classes. There are several program-development scenarios where this weakness would be a serious problem. For example, class designers may wish to perform some kind of bookkeeping whenever objects are instantiated from a class or its descendants. To achieve it, programmers need a place to put code that will execute whenever an object is instantiated. With the record-of-premethods approach, however, there is no appropriate place: no base class constructor function will be called for derived classes, and a premethod function may be called without creating an object.

Does the class construct provide an explicit type hierarchy? In many existing class-based languages, it is possible to restrict the subtypes of an "implementation" object type (i.e., a class) to classes that inherit all or part of the object's implementation. This restriction may be useful for optimizing operations on objects, allowing access to argument objects in binary methods, and guaranteeing semantic consistency beyond type considerations [KLM94]. A special case of this capability is the ability to define *final* classes, as recognized in work on Rapide [KLM94] and incorporated (presumably independently) as a language feature in Java. This ability is lacking in the record-of-premethods approach since any object whose type is a structural subtype of another type τ can be used as an object of type τ.

Does the class mechanism automatically propagate base class changes? Because derived class constructors must explicitly name the methods that they wish to inherit, the record-of-premethods approach does not automatically propagate base class method changes. In particular, if a derived class `D` is defined from a base class `B` in Java or related languages, then adding a method to `B` will result in an additional method of `D`, and similarly for every other class derived from `B` (and there may be many). With the record-of-premethods approach, derived class constructors must be explicitly rewritten each time base classes change. Since object-oriented programs are typically quite large and maintenance may be distributed across many people, the person who maintains a base class may fail to inform those maintaining its derived classes of its change, causing unpredictable errors. There is no mechanism in this approach to detect such errors automatically.

6.2 Extensible Object Model

While we readily admit that its simplicity is a virtue, the above discussion reveals that several important and desirable features of class-based programming are lost in the record-of-premethods model. Extensible objects provide a rich alternative. A principled way to think about class-based object-oriented languages is as the combination of two orthogonal components [Fis96,FM98]: (*i*), an object system that supports inheritance and message sending and (*ii*), an encapsulation mechanism that provides hiding. We call this model of classes the “Classes = Extensible Objects + Encapsulation” approach. Referring to the class-evaluation checklist we used to evaluate the pre-methods model, we can see that this approach successfully addresses each of the points listed there: it provides an extensible coherent collection, guarantees initialization, supports an explicit type hierarchy, and automatically propagates base class changes.

Extensible, coherent collection. Extensible objects obviate the need for premethods, since collections of methods that are already installed in objects may be extended. Because of this fact, we may impose static constraints on the ways in which one method may be combined with others. For example, if an object contains two mutually recursive methods, then we cannot replace one with another of a different type. In contrast, in the record-of-premethods approach, it is possible to form a record of premethods without a “covering” constructor that checks to be sure that all of the premethods are coherent.

Guaranteed initialization. A second advantage of extensible objects is that class constructors and initialization code can be inherited, i.e., reused in derived classes. For example, to create `ColorPoint` objects, we may invoke a `Point` class constructor and add color methods to the resulting extensible object. This process guarantees that the `Point` class has the opportunity to initialize any inherited components properly. It also guarantees that the designers of the `Point` class have the opportunity to update any bookkeeping information they may be keeping about instantiations of `Point` objects.

Explicit type hierarchy. A further advantage is the rich subtyping structure of this approach. In particular, it provides "implementation" types that subtype along the inheritance hierarchy, "interface" types that subtype via structural subtyping rules, and a hybrid subtyping relation that allows implementation types to be subtypes of interface types. With this subtyping structure, programmers can use implementation types where the extra information is useful and interface types where more generality is required.

Automatic propagation of changes. Another advantage arises with private (or protected) methods. In the extensible-object formulation, methods always remain within an object, even when it is extended. These hidden methods exist in all future extensions, but they can only be accessed by methods that were defined before the method became hidden. Furthermore, these private methods need not be manipulated explicitly by derived class constructors to insure that they are treated properly.

These advantages may be seen in the encoding of the traditional `Point` and `ColorPoint` hierarchy, studied in the paper [BF98]. In this tutorial we present it in its pseudo-type-theory version only, to develop intuitions for the formal model.

6.3 Pseudo-Type-Theoretic `Point`, `ColorPoint` Hierarchy

In the "Classes = Extensible Objects + Encapsulation" model of classes [Fis96,FM98], extensible objects support the inheritance aspects of classes, while an encapsulation mechanism provides the hiding. We illustrate the ideas behind this model by encoding the familiar `Point`, `ColorPoint` hierarchy in pseudo-type theory. The code, which appears in Figure 1, contains two class declarations followed by "*Client Code.*"

To explain the model, we focus on the **Class** encapsulation construct, which provides the outer wrapping for each of the class declarations. In it, the **Class** clause names the abstraction-as-class, `Point` in the first case, `ColorPoint` in the second. The **implements** clause gives the public and protected interfaces supported by the class, *"Point_public_interface"* and *"Point_protected_interface,"* respectively, in the `Point` case. A public interface lists the methods available from instances of its class. Such a list for a simple `Point` class might be of the form $\langle\!\langle \texttt{getX} : int, \texttt{setX} : int \rightarrow unit \rangle\!\rangle$, revealing that objects of the class contain `getX` and `setX` methods of the indicated types. At the discretion of the class designer, a class's public interface may explicitly name its parent class,

```
Class Declarations
    Class Point implements "Point_public_interface", "Point_protected_interface"
    exports newP : int → "extensible obj. type from Point class"
    is
            ⦃ "Point_private_interface"; "code to implement newP" ⦄
    end;

    Class ColorPoint implements "CP_public_interface", "CP_protected_interface"
    exports newCP : color → int → "non-extensible obj. type from ColorPoint class"
    is
            ⦃ "CP_private_interface"; "code to implement newCP" ⦄
    end;

in
   "If desired, restrict return types of non-final constructors;"
   "Client Code"
end
```

Fig. 1. Point, ColorPoint hierarchy.

if one exists. For example, the ColorPoint public interface might be of the form ⟪Point | getC : *color*, setC : *color* → *unit*⟫. The Point portion of this interface indicates that objects created from the ColorPoint class were formed via inheritance from the Point class; hence, they have the Point class methods. In addition, by thus indicating the parent class, the ColorPoint class designer declares that the "implementation type" associated with the ColorPoint class is a subtype of the Point class's "implementation type." Through this declaration mechanism, the model supports an explicit type hierarchy. The second half of the ColorPoint public interface indicates that the ColorPoint class added getC and setC methods.

The protected interface augments the public one with information for deriving classes. In this model, this information consists of method and field names that may not be used in derived classes without introducing name clashes.

The **exports** clause of the encapsulation mechanism reveals the names and types of the non-dynamically dispatched operations defined by the class. In general, this clause lists constructor and "friend" functions. In the example, the Point class designer chose to export a single constructor function, newP of type *int* → "*extensible object type from* Point *class*." By making the return type an extensible object type, the class designer enabled inheritance from this class: a derived class calls newP to get the implementation of the Point class and then adds and redefines components as necessary. Since the ColorPoint class designer made newCP return a non-extensible object, the ColorPoint class is "final," in the sense that no other class can be formed by extending its implementation. The Point class designer opted to make the return type of newP flag its defining class (via the "*from* `Point` *class*" annotation). Because this information is present in the constructor type, the `ColorPoint` class designer can export its parent's identity. Without it, the derived class could reuse its parent's implementation but could not reveal this fact nor make the `ColorPoint` implementation type a subtype of the `Point` implementation type.

The **is** clause of the encapsulation mechanism has two pieces. The first part, the private interface, lists all the methods defined within the class. For a simple `Point` class, this interface might be of the form $\langle\!\langle \mathtt{x} : \mathit{int}, \mathtt{getX} : \mathit{int}, \mathtt{setX} : \mathit{int} \rightarrow \mathit{unit} \rangle\!\rangle$, where `x` is a private field.

The second piece of a class implementation is the code to implement the constructor and friend functions listed in the **exports** clause. In the `Point` class case, this code simply defines an extensible object with field `x` and methods `getX` and `setX`. For the `ColorPoint` class, the constructor implementation first calls `newP` to inherit the `Point` class behavior and then adds color-related fields and methods. If the `ColorPoint` class advertises the fact that it inherits from `Point` in its public interface, then the type system insures that the `ColorPoint` constructor function calls `newP` and returns an extension of the resulting object. Thus the type system guarantees that the `Point` class has a chance to initialize its private variables and set up its desired invariants for any object instantiated from it, either directly or via a derived class.

Because the **Class** construct is an encapsulation mechanism, only the aspects of the **is** code specifically mentioned in the **implements** and **exports** clauses can be used in the rest of the program. Hence in the encoding, this mechanism ensures the privacy of private methods and fields.

After we process all the class declarations in the pseudo-code, we are almost ready to execute the "*Client Code*." Without any further adjustment, however, non-final classes have constructor functions that return extensible objects, which enable run-time inheritance. If we wish to disable this feature, we may restrict the return types of these constructor functions to return "non-extensible" objects instead. This type restriction does not involve changing the values in any way; it simply adjusts the types. The restriction is safe because every extensible object type in the system is a subtype of the corresponding non-extensible version.

7 Related Work

In the Lambda Calculus of Objects [FHM94], *MyType* inheritance is rendered via *row variables* (instead of being modelled by match-bound quantification as it is for $\mathtt{Obj}^{+}$). The type of SELF is a partially non-defined row-type, where the non-defined part is a row-variable representing all possible extensions that the host object may be subjected to. Rows are validated by using *kinds*. Also the calculus of [BF98] we presented informally in Section 6 is based on row-variables.

The work [Liq98] presents a detailed comparison among four type systems for the Lambda Calculus of Objects: the original one [FHM94], the Fisher's thesis one [Fis96], an earlier version of $\mathtt{Obj}^{+}$ [BB99], and a system based on bounded polymorphism.

In the literature, there are proposals that integrate extensible objects in broader contexts. We mention three of them:

- Baby Modula 3 [Aba94] is a toy language that provides extensible objects. In order to ensure safety with respect to subtyping, all of the object extensions must be done before applying any form of subsumption. This language also accounts for a notion of "incomplete objects", for which completions are fixed ahead of time, prior to any addition. A calculus that offers a more complex form of incomplete objects is presented in [BBDCL99].
- The calculus presented in [Rém98] is a version of the Abadi and Cardelli calculus [AC96b] equipped with extensible objects, as it is our $\mathtt{Obj}^+$, but its type system is richer. The underlying idea is to trace subtyping, in such a way method addition and subtyping-in-width can co-exists. This extra-information allows also to model a form of *virtual* methods (i.e., it models a form of incomplete objects). Moreover, when sufficient type information is available, objects play a rôle similar to the one of classes; such information can be then hidden progressively, objects regaining their proper rôle.
- The paper [DHL98] extends the Lambda Calculus of Objects with a form of *self-inflicted* method addition. Relationships between this calculus and foundations for dynamic re-classification of objects [DDDCG02] are under study.

References

[Aba94] M. Abadi. Baby Modula–3 and a Theory of Objects. *Journal of Functional Programming*, 4(2):249–283, 1994.

[ABDD03] C. Anderson, F. Barbanera, M. Dezani-Ciancaglini, and S. Drossopoulou. Can addresses be types? (a case study: objects with delegation). In *WOOD'03*, volume 82.8 of *ENTCS*. Elsevier, 2003.

[AC95] M. Abadi and L. Cardelli. On Subtyping and Matching. In *Proceedings of ECOOP'95: European Conference on Object-Oriented Programming*, volume 952 of *LNCS*, pages 145–167. Springer–Verlag, 1995.

[AC96a] M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.

[AC96b] M. Abadi and L. Cardelli. A Theory of Primitive Objects: Untyped and First-Order System. *Information and Computation*, 125(2):78–102, March 1996.

[ACV96] M. Abadi, L. Cardelli, and R. Viswanathan. An Interpretation of Objects and Object Types. In *Proc. of POPL'96*, pages 396–409, 1996.

[AD02a] C. Anderson and S. Drossopoulou. Babyj - from object based to class based programming via types. In *Proc. of WOOD'03*, volume 82.8 of *ENTCS*, 2002. Workshop of ETAPS'03.

[AD02b] C. Anderson and S. Drossopoulou. δ - an imperative object based calculus. Presented at the workshop USE in 2002, Malaga, 2002.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[BB99] V. Bono and M. Bugliesi. Matching for the Lambda Calculus of Objects. *Theoretical Computer Science*, 212(1/2):101–140, 1999.

[BBC02] V. Bono, M. Bugliesi, and S. Crafa. Typed interpretations of extensible objects. *ACM Transactions on Computational Logic*, 2002.

[BBDCL99] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. A Subtyping for Extensible, Incomplete Objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.

[BCC$^+$95] K.B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On Binary Methods. *Theory and Practice of Software Systems*, 1(3):217–238, 1995.

[BCP97] K.B. Bruce, L. Cardelli, and B. Pierce. Comparing Object Encodings. In *Proc. of TACS'97*, volume 1281 of *LNCS*, pages 415–438. Springer-Verlag, 1997.

[BDG02] V. Bono, F. Damiani, and P. Giannini. A calculus for "environment-aware" computation. In *F-WAN'02*, volume 66.3 of *ENTCS*. Elsevier, 2002.

[BF98] V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *Proc. of ECOOP'98*, volume 1445 of *LNCS*, pages 462–497, 1998. A preliminary version already appeared in Proc. of 5th Annual FOOL Workshop.

[BL95] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL'94*, volume 933 of *LNCS*, pages 16–30. Springer-Verlag, 1995.

[Bru94] K.B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[Bru02] K.B. Bruce. *Foundations of Object-Oriented Languages – Types and Semantics.* The MIT Press, 2002.

[Car95] L. Cardelli. A Language with Distribuite Scope. *Computing Systems*, 8(1):27–59, 1995.

[Cas96] G. Castagna. *Object-Oriented Programming: a Unified Foundation.* Birkauser, 1996.

[CG] UW Cecil Group. UW Cecil Group : Home. Cecil's language home page.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *Proc. of ACM Symp. POPL'90*, pages 125–135. ACM Press, 1990.

[Coo89] W.R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[Cra99] K. Crary. Simple, efficient object encoding using intersection types. Tech. rep., CMU-CS-99-100, Cornell University, 1999.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DDDCG02] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{II}$. *ACM Transactions On Programming Languages and Systems*, 24(2):153–191, 2002.

[DG03] F. Damiani and P. Giannini. Alias types for "environment-aware" computations. In *WOOD'03*, volume 82.8 of *ENTCS*. Elsevier, 2003.

[DHL98] P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Proc. of ACM-SIGPLAN OOPSLA, International Symposium on Object Oriented, Programming, System, Languages and Applications*, pages 166–178. The ACM Press, 1998.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994. A preliminary version appeared in *Proc. of IEEE Symp. LICS'93*.

[Fis96] K. Fisher. *Type Systems for Object-Oriented Programming Languages.* PhD thesis, Stanford University, 1996. Available as Stanford Computer Science Technical Report number STAN-CS-TR-98-1602.

[Fla99] D. Flannanghan. *JavaScript: The definitive guide.* O'Reilly, 1999.

[FM95] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *LNCS*, pages 42–61. Springer-Verlag, 1995.

[FM98] K. Fisher and J.C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(3), 1998. Special Issue on Third Workshop on Foundations of Object-Oriented Languages (FOOL 3). Preliminary version appeared in Marktoberdorf '97 proceedings.

[Kam88] S. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proc. of POPL'88*, pages 80–87. ACM Press, 1988.

[KLM94] D. Katiyar, D. Luckham, and J.C. Mitchell. A type system for prototyping languages. In *Proc. 21st of Symp. on Principles of Programming Languages.* ACM, 1994.

[Liq98] L. Liquori. On Object Extension. In *Proc. of ECOOP, European Conference on Object Oriented Programming*, volume 1445 of *Lecture Notes in Computer Sciences*, pages 498–552. Springer Verlag, 1998.

[Pie02] B.C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[PT94] B. Pierce and D. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994.

[Rém98] D. Rémy. From classes to objects via subtyping. In *Proceedings of ESOP'98*, volume 1381 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.

[RR96] J.H. Reppy and J.G. Riecke. Classes in Object ML via modules. In *Proc. of FOOL3 Workshop*, 1996.

[US87] D. Ungar and R. B. Smith. Self: the Power of Simplicity. In *Proc. of OOPSLA'87*, pages 227–241. ACM Press, 1987.

A $\texttt{Obj}^{+}$

A.1 Notation

$$\langle \ell_i = \mathbf{b}_i\{x\}^{i\in I\cup J}\rangle \triangleq \langle v_i = {c_i}^{i\in I}, m_j = \varsigma(x)b_j\{x\}^{j\in J}\rangle$$

$$a\cdot\ell \triangleq a\circ\ell \; \mathit{or} \; a{\scriptstyle\blacksquare}\ell$$

$$a\cdot\ell \leftarrow \mathbf{b}\{x\} \triangleq a\circ\ell \leftarrow \varsigma(x)b\{x\} \text{ or } a{\scriptstyle\blacksquare}\ell \leftarrow b$$

$$a\cdot\ell \longleftarrow\!\!\!+ \; \mathbf{b}\{x\} \triangleq a\circ\ell \longleftarrow\!\!\!+ \; \varsigma(x)b\{x\} \text{ or } a{\scriptstyle\blacksquare}\ell \longleftarrow\!\!\!+ \; b$$

A.2 $\Downarrow_o$: Big-Step Operational Semantics

Results : $r = \langle v_i = {c_i}^{i\in I}, m_j = \varsigma(x)b_j\{x\}^{j\in J}\rangle$

$$(\mathrm{Select_v})\quad \frac{a \Downarrow_o \langle \ldots, v_j{=}c_j, \ldots\rangle \quad c_j \Downarrow_o r}{a\text{.}v_j \Downarrow_o r}$$

$$(\mathrm{Select_m})\quad \frac{a \Downarrow_o \widehat{a} \quad b_j\{\widehat{a}\} \Downarrow_o r \quad (\widehat{a} \equiv \langle \ldots, m_j = \varsigma(x)b_j\{x\}, \ldots\rangle)}{a \circ m_j \Downarrow_o r}$$

$$(\mathrm{Update})\quad \frac{a \Downarrow_o \langle \ell_i = \mathbf{b}_i\{x\}^{i\in I\cup J}\rangle \quad k \in I \cup J}{a \cdot \ell_k \leftarrow \mathbf{b}\{x\} \Downarrow_o \langle \ell_i = \mathbf{b}_i\{x\}^{i\in I\cup J-\{k\}}, \ell_k = \mathbf{b}\{x\}\rangle}$$

$$(\mathrm{Extend})\quad \frac{a \Downarrow_o \langle \ell_i = \mathbf{b}_i\{x\}^{i\in I\cup J}\rangle \quad \ell \notin \{\ell_i\}^{i\in I\cup J}}{a \cdot \ell \longleftarrow\!\!\!+ \mathbf{b}\{x\} \Downarrow_o \langle \ell = \mathbf{b}\{x\}, \ell_i = \mathbf{b}_i\{x\}^{i\in I\cup J}\rangle}$$

A.3 $\succ_o$: Small-Step Operational Semantics

$a \equiv \langle v_i = {c_i}^{i\in I}, m_j = \varsigma(x)b_j\{x\}^{j\in J}\rangle$

$(\mathit{Select_v})\quad a\text{.}v_i \succ_o c_i \qquad i \in I$

$(\mathit{Select_m})\quad a \circ m_j \succ_o b_j\{a\} \qquad j \in J$

$a \equiv \langle \ell_i = \mathbf{b}_i\{x\}^{i\in I\cup J}\rangle$

$(\mathit{Extend})\qquad \ell \notin \{\ell_i\}^{i\in I\cup J}$

$\qquad a \cdot \ell \longleftarrow\!\!\!+ \mathbf{b}\{x\} \succ_o \langle \ell = \mathbf{b}\{x\}, \ell_i = \mathbf{b}_i\{x\}^{i\in I\cup J}\rangle$

$(\mathit{Update})\qquad k \in I \cup J$

$\qquad a \cdot \ell_k \leftarrow \mathbf{b}\{x\} \succ_o \langle \ell_i = \mathbf{b}_i\{x\}^{i\in I\cup J-\{k\}}, \ell_k = \mathbf{b}\{x\}\rangle$

A.4 Typing Rules

Context Formation

$$(\mathrm{Ctx}\ \varnothing)\ \frac{}{\varnothing \vdash *} \qquad (\mathrm{Ctx}\ X)\ \frac{\mathit{\Gamma} \vdash * \quad X \notin \mathit{Dom}(\mathit{\Gamma})}{\mathit{\Gamma}, X \vdash *} \qquad (\mathrm{Ctx\ Match})\ \frac{\mathit{\Gamma} \vdash \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in I}\rangle\!\rangle \quad U \notin \mathit{Dom}(\mathit{\Gamma})}{\mathit{\Gamma}, U \mathrel{<\!\!\!\#} \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in I}\rangle\!\rangle \vdash *}$$

Type formation

$$(\mathrm{Type}\ X)\ \frac{\mathit{\Gamma}', X, \mathit{\Gamma}'' \vdash *}{\mathit{\Gamma}', X, \mathit{\Gamma}'' \vdash X} \qquad (\mathrm{Type\ Match}\ U)\ \frac{\mathit{\Gamma}', U \mathrel{<\!\!\!\#} A, \mathit{\Gamma}'' \vdash *}{\mathit{\Gamma}', U \mathrel{<\!\!\!\#} A, \mathit{\Gamma}'' \vdash U} \qquad (\mathrm{Type}\ \mathtt{pro})\ \frac{\mathit{\Gamma}, X \vdash B_i\{X\}}{\mathit{\Gamma} \vdash \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i\in I}\rangle\!\rangle}$$

Term Formation

(Val x)

$$\frac{\Gamma', x : A, \Gamma'' \vdash *}{\Gamma', x : A, \Gamma'' \vdash x : A}$$

(Val Select)

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <\!\!\# \, \mathtt{pro}(X)\langle\!\langle \ell : B\{X\} \rangle\!\rangle}{\Gamma \vdash a \cdot \ell : B\{A\}}$$

(Val Object: $A \equiv \mathtt{pro}(X)\langle\!\langle v_i : {C_i}^{i \in I}, m_j : B_j\{X\}^{j \in J} \rangle\!\rangle$)

$$\frac{\Gamma \vdash c_i : C_i \qquad \Gamma, U <\!\!\# \, A, x : U \vdash b_j\{x\} : B_j\{U\} \qquad \forall\ i \in I, j \in J}{\Gamma \vdash \langle v_i = {c_i}^{i \in I}, m_j = \varsigma(x) b_j\{x\}^{j \in J} \rangle : A}$$

(Val Field Addition: $A^+ \equiv \mathtt{pro}(X)\langle\!\langle \ell : B\{X\}, \ell_i : B_i\{X\}^{i \in I} \rangle\!\rangle$)

$$\frac{\Gamma \vdash a : \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i \in I} \rangle\!\rangle \quad \Gamma \vdash c : B \quad (\ell \neq \ell_i\ \forall i \in I)}{\Gamma \vdash a{\bullet}\ell \longleftarrow\!\!\!\!+\ c : A^+}$$

(Val Method Addition: $A^+ \equiv \mathtt{pro}(X)\langle\!\langle \ell : B\{X\}, \ell_i : B_i\{X\}^{i \in I} \rangle\!\rangle$)

$$\frac{\Gamma \vdash a : \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i \in I} \rangle\!\rangle \quad \Gamma, U <\!\!\# \, A^+, x : U \vdash b\{x\} : B\{U\} \quad (\ell \neq \ell_i\ \forall i \in I)}{\Gamma \vdash a \circ \ell \longleftarrow\!\!\!\!+\ \varsigma(x) b\{x\} : A^+}$$

(Val Field Update)

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <\!\!\# \, \mathtt{pro}(X)\langle\!\langle v : C \rangle\!\rangle \quad \Gamma \vdash c : C}{\Gamma \vdash a{\bullet}v \leftarrow c : A}$$

(Val Method Update: $A \equiv \mathtt{pro}(X)\langle\!\langle v_i : {C_i}^{i \in I}, m_j : B_j\{X\}^{j \in J} \rangle\!\rangle$)

$$\frac{\Gamma \vdash a : A \quad \Gamma, U <\!\!\# \, A, x : U \vdash b\{x\} : B_k\{U\} \quad k \in J}{\Gamma \vdash a \circ m_k \leftarrow \varsigma(x) b\{x\} : A}$$

Matching

(Match U)

$$\frac{\Gamma', U <\!\!\# \, A, \Gamma'' \vdash *}{\Gamma', U <\!\!\# \, A, \Gamma'' \vdash U <\!\!\# \, A}$$

(Match Refl)

$$\frac{\Gamma', U <\!\!\# \, A, \Gamma'' \vdash U}{\Gamma', U <\!\!\# \, A, \Gamma'' \vdash U <\!\!\# \, U}$$

(Match Trans)

$$\frac{\Gamma \vdash U <\!\!\# \, B \quad \Gamma \vdash B <\!\!\# \, A}{\Gamma \vdash U <\!\!\# \, A}$$

(Match $\mathtt{pro}$)

$$\frac{\Gamma \vdash \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i \in 1..n+k} \rangle\!\rangle}{\Gamma \vdash \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i \in 1..n+k} \rangle\!\rangle <\!\!\# \, \mathtt{pro}(X)\langle\!\langle \ell_i : B_i\{X\}^{i \in 1..n} \rangle\!\rangle}$$