The Recursive Record Semantics of Objects Revisited

(Extended Abstract)[⋆]

Gérard Boudol

INRIA Sophia Antipolis
BP 93 – 06902 Sophia Antipolis Cedex, France

Abstract. In a call-by-value language, representing objects as recursive records requires using an unsafe fixpoint. We design, for a core language including extensible records, a type system which rules out unsafe recursion and still supports the reconstruction of a principal type. We illustrate by means of various examples the expressive power of this language with respect to object-oriented programming constructs.

1 Introduction

During the past fifteen years there has been very active research about the formalization of object-oriented programming concepts. One of the main purposes of this research was to design operational models of objects supporting rich type systems, so that one could benefit both from the flexibility of the object-oriented style, and from the safety properties guaranteed by typing. Let us be more precise here: our goal is to have an expressive language – as far as object-oriented constructs are concerned – with a type discipline à la ML [12,24], i.e. implicit typing with reconstruction of a principal type, ruling out run-time errors. This goal has proven difficult to achieve, and most of the many proposals that were put forward fall short of achieving it – with the exception of OCaml [21,27], that we will discuss later.

While the meaning of "typing à la ML" should be clear, it is perhaps less easy to see what is meant by "object-oriented". We do not claim to answer to this question here. Let us just say that, in our view, objects encapsulate a state and react to messages, i.e. method invocations, by udating their state and sending messages, possibly to themselves. Moreover, in our view, object-orientation also involves inheritance, which includes – but should not be limited to, as we shall see – the ability to add and redefine methods. With this informal notion of object-orientation in mind, let us review some of the proposals we alluded to.

An elegant proposal was made by Wand [31], based on his *row variables* [30], consisting of a class-based model, where classes are functions from instance variables and a self parameter to extensible records of methods, and objects are

⋆ Work partially supported by the RNRT Project Marvel, and by the CTI "Objets Migrants: Modélisation et Vérification", France Télécom R&D.

fixpoints of instantiated classes, that is, recursive records. In this model invoking the method of an object amounts to selecting the corresponding component of the record representing the object. An operation of record extension is used to provide a simple model of inheritance, à la SMALLTALK. Unfortunately, although its elegance and simplicity make it very appealing, Wand's model is not expressive enough. More specifically, it does not support state changes in objects: one may override a method in an inherited class, but one apparently cannot modify the state of the object during its life-time (see for instance [1] Section 6.7.2). This is because in creating the object, the self parameter is bound too early.

Wand's model is an instance of what is known as the *recursive record semantics* for objects (see [16]), initiated by Cardelli [8]. Based on this idea that an object is the fixpoint of an instantiated class, Cook proposed a more elaborate model [11], where updating the state of an object is possible, by creating new objects, instances of the same class. This model is operationally quite expressive, but the type theory that it uses is also quite elaborate, and does not fulfil our desires, of reconstruction of a principal type. The same remark actually applies to all the object models that use higher-order types, e.g. [1,6,13,15,25].

In another approach, due to Kamin [20] and known as the *self-application semantics*, an object is a record of pre-methods, that are functions of the object itself. The object is bound to self only when a method is invoked, by applying the pre-method to the object. In this way, the state of the object may dynamically be updated. In this model, which looks indeed operationally satisfactory, an object is not quite a record, since from a typing point of view, we must know that the first parameters (that is, self) of all its pre-methods have the same type. In other words, one must have in this approach specific constructs for objects and object types, depending on the type of self, thus different from record types. This has been developed in *object calculi*, most notably by Fisher and Mitchell [14,15,16] (who call it "the axiomatic approach") and Abadi and Cardelli [1], but as we already noticed, in calculi that support a rich form of inheritance, and in particular object extension, like [15], the type theory which is used is quite elaborate, and does not support implicit typing.

Object calculi claim to fix the principles for objects, thus providing simple formal models, but they actually take design decisions, about inheritance in particular – a concept which is still a matter of debate in the object-oriented programming community (see [29] for example). As a matter of fact, many of the proposals for an object model, including OCAML, follow this approach of designing a specific calculus, e.g. [1,3,6,15,27]. However, there could be some benefits from deriving object-oriented concepts from more basic principles: first, their typing could be derived within simple, unquestionable typing systems. Second, they could be better integrated in a standard computational model, in which one could formalize and compare various approaches to objects, and get more flexible object models. Furthermore, we would not have to develop specific theories for reasoning about them.

In this paper we pursue Wand's approach, aiming at encoding object-oriented concepts by means of extensible records. One may observe that the update operation of object calculi [1,15] is actually overloaded: it serves both in inheritance, to override methods, and in the dynamic behaviour of an object, to update the state. As we have seen, the first usage is not problematic in Wand's model, whereas the second is. Then an obvious idea is to abandon the "functional update" approach in favor of a rather more natural *imperative update* approach, as in [1, 3,13,27]. This means that we are in a language with *references* (following ML's terminology), where a call-by-value strategy is assumed for evaluation. Now a new problem arises: to build objects as recursive records one must have the ability to build recursive non-functional values, and this, in principle, is not supported in a typed call-by-value language. More specifically, we would like to use ($\mathsf{let\,rec}\ x = N\ \mathsf{in}\ M$), where N may be of a record type. This is evaluated by first computing a value for N, returning a cyclic binding to this value for x, and then computing M. Notice that side effects and creation of new references arising from the evaluation of N are completed before a cyclic binding is returned. This is what we need to install the state of an object before returning its (recursive) record of methods. The resulting object model is similar to what is known as the "cyclic record" encoding, see [1], Sections 18.2.4 and 18.3.4.

As remarked by Rémy [26], a recursive record semantics of objects works fine with the $\mathsf{let\,rec}$ construct, except that this construct is *unsafe*. Indeed, some langages, like SCHEME or OCAML, provide us with this feature, but, except for defining recursive functions, its semantics is implementation-dependent. More precisely, in computing ($\mathsf{let\,rec}\ x = N\ \mathsf{in}\ M$), it could happen that evaluating N we have to call the value of x, which is not yet computed, thus getting stuck at this point. One must then have means to prevent such a run-time error in order to design a "safe" object model from recursive records. We must point out that, although this problem of determining restrictions on recursive definitions to ensure that they define something is not at all a new one, no obvious solution to our specific problem emerges from the literature, since ($\mathsf{let\,rec}\ x = (Gx)\ \mathsf{in}\ M$) must sometimes be accepted, where G reduces to a "generator" $\lambda\mathsf{self}M$ [10].

The main contribution of this paper is a solution to this problem: first, we extend the core "Reference ML" language, as considered by Wright and Felleisen [32], with operations on records, similar to the ones of Cardelli and Mitchell [9]. We then provide a type system for this language, refining the simple types by assigning a boolean "degree of harmlessness" to the argument of a function, considering types of the form $\theta^d \to \tau$, and to variables in the typing context. Typically, a variable occurring within a value is harmless, hence may have degree 1, and applying a function of type $\theta^0 \to \tau$ means that the argument might be put in a dangerous position. The "harmlessness degree" is used to determine whether a variable may or may not be safely recursively bound. We show that the evaluation of a typable term either diverges or returns a value, thus avoiding to get stuck in run-time errors, and, adapting a result by Jategaonkar and Mitchell [19], we show that a principal type may be computed for any typable expression. Although our goal here is not to design and encode an object-oriented layer in the language, we examine various examples, to assess the usefulness of the approach, and to illustrate the expressive power of the model.

Note. The proofs of the technical results are not included in this extended abstract. They are to be found in the full version of the paper, at the author's web page (`http://www-sop.inria/mimosa/personnel/Gerard.Boudol.html`).

$$
\begin{array}{rcll}
M,\ N\ldots & ::= & & \textit{expressions}\\
& & V \mid (MN) \mid (\mathsf{let}\ D\ \mathsf{in}\ M) & \quad\textit{core constructs}\\
& \mid & \langle M,\ell = N\rangle \mid (M.\ell) \mid (M\backslash\ell) & \quad\textit{record operations}\\
V,\ W\ldots & ::= & x \mid \mathsf{ref} \mid \mathsf{!} \mid \mathsf{set} \mid (\mathsf{set}\,V) & \textit{values}\\
& \mid & \lambda xM \mid () \mid R & \\
R & ::= & x \mid \langle\rangle \mid \langle R,\ell = V\rangle & \textit{record values}\\
D & ::= & x = N \mid \mathsf{rec}\,x = N & \textit{declarations}
\end{array}
$$

Fig. 1. Syntax

2 The Calculus

Assuming that a set $\mathcal{X}$ of variables, ranged over by x, y, $z\ldots$, and a set $\mathcal{L}$ of labels are given, the syntax of our core language is given in Figure 1, where $x\in\mathcal{X}$ and $\ell\in\mathcal{L}$. It contains the "Reference ML" calculus of [32] – defining the call-by-value fixpoint combinator Y as $(\mathsf{let}\,\mathsf{rec}\ y = \lambda f.f(\lambda x.(yf)x)\ \mathsf{in}\ y)$, and denoting $:=$ by set. Free (fv) and bound (bv) variables are defined as usual, and we denote by $\{x\mapsto N\}M$ the capture-free substitution.

Regarding records, we use the operations of [9], denoting by $\langle M,\ell = N\rangle$ the record M extended with a new field, labelled ℓ, with value N. As in [9], this will only be well-typed if M does not exhibit an ℓ field, whereas the restriction operation, still denoted $(M\backslash\ell)$ and consisting of removing the ℓ field from M, will only be well-typed here if M does contain an ℓ field. The overriding operation is denoted $\langle M,\ell\leftarrow N\rangle$; this is an abbreviation for $\langle(M\backslash\ell),\ell = N\rangle$. We shall write $\langle\ell_1 = M_1,\ldots,\ell_n = M_n\rangle$ for the record $\langle\cdots\langle\langle\rangle,\ell_1 = M_1\rangle\ldots,\ell_n = M_n\rangle$.

Now we specify the semantics of our language, defining an *evaluation relation* $M\rightarrow M'$, that we also call *local* (or functional) *reduction*, which can be performed in *evaluation contexts*. The axioms and rules are given in Figure 3. To describe the semantics of the imperative constructs, given by the rules for *global reduction* in Figure 4, we enrich the language with a denumerable set $\mathcal{N}$ of *names*, or *locations* u, v, $w\ldots$, distinct from the variables and the labels. These names are also values. A *configuration* is a pair $(S \mid M)$ of an expression M and a *store*, that is a mapping from locations to values. We use the following syntax for stores:

$$S\ ::=\ \varepsilon \mid u := V;S$$

$$\begin{array}{rl}\mathbf{E} ::= & [] \mid (\mathbf{E}N) \mid (V\mathbf{E}) \mid (\mathsf{let}\ x = \mathbf{E}\ \mathsf{in}\ M) \mid (\mathsf{let\,rec}\ x = \mathbf{E}\ \mathsf{in}\ M) \\ \mid & \langle \mathbf{E}, \ell = N\rangle \mid \langle R, \ell = \mathbf{E}\rangle \mid (\mathbf{E}.\ell) \mid (\mathbf{E}\backslash\ell)\end{array}$$

Fig. 2. Evaluation Contexts

$$\begin{array}{rll}(\lambda x M V) \to & \{x\mapsto V\}M & \\ (\mathsf{let}\ x = V\ \mathsf{in}\ M) \to & \{x\mapsto V\}M & \\ (\mathsf{let\,rec}\ x = V\ \mathsf{in}\ M) \to & \{x\mapsto (\mathsf{let\,rec}\ x = V\ \mathsf{in}\ V)\}M & \\ (\langle R, \ell = V\rangle.\ell) \to & V & \\ (\langle R, \ell = V\rangle.\ell') \to & (R.\ell') & \ell' \neq \ell \\ (\langle R, \ell = V\rangle\backslash\ell) \to & R & \\ (\langle R, \ell = V\rangle\backslash\ell') \to & \langle (R\backslash\ell'), \ell = V\rangle & \ell' \neq \ell \\ M \to M' \ \Rightarrow & \mathbf{E}[M] \to \mathbf{E}[M'] & \end{array}$$

Fig. 3. Local Reduction

$$\begin{array}{rll}M \to M' \Rightarrow & (S \mid M) \to (S \mid M') & \\ (S \mid \mathbf{E}[(\mathsf{ref}\,V)]) \to & (u := V; S \mid \mathbf{E}[u]) & u \notin \mathsf{dom}(S) \\ (S \mid \mathbf{E}[(!u)]) \to & (S \mid \mathbf{E}[V]) & S(u) = V \\ (S \mid \mathbf{E}[((\mathsf{set}\,u)V)]) \to & ([u := V]S \mid \mathbf{E}[()]) & \end{array}$$

Fig. 4. Global Reduction

The value $S(u)$ of a name in the store, and the partial operation $[u := V]S$ of updating the store, are defined in the obvious way. In the rules for global reduction, we have omitted the side condition that $\mathsf{fv}(V) \cap \mathsf{capt}(\mathbf{E}) = \emptyset$, where $\mathsf{capt}(\mathbf{E})$ is the set of variables that are bound in $\mathbf{E}$ by a $\mathsf{let\,rec}$ binder introducing a sub-context. Let us see an example – which will be the standard object-oriented example of a "point". Assuming that some arithmetical operations are given, we define a "class" of unidimensional points as follows:

$$\begin{array}{l}\mathsf{let\ point} = \lambda x \lambda \mathsf{self}\langle \mathsf{pos} = \mathsf{ref}\,x, \\ \qquad\qquad \mathsf{move} = \lambda y((\mathsf{set\ self.pos})(!\mathsf{self.pos} + y))\rangle\ \mathsf{in}\ldots\end{array}$$

Within the scope of this definition, we may define a point object, instance of that class, by intantiating the position parameter x to some initial value, and building a recursive record of methods. Let us define the fixpoint operator fix as follows:

$$\mathsf{fix} =_{\mathrm{def}} \lambda f(\mathsf{let\,rec}\ x = fx\ \mathsf{in}\ x)$$

Then we have for instance, if we let $V = \lambda y((\mathsf{set}\ x.\mathsf{pos})(\mathbf{!}x.\mathsf{pos} + y))$ and $R = \langle \mathsf{pos} = u, \mathsf{move} = V\rangle$:

$$\begin{aligned}(\varepsilon \mid \mathsf{fix}(\mathsf{point}\,0)) &\xrightarrow{*} (\varepsilon \mid (\mathsf{let\,rec}\ x = \langle \mathsf{pos} = \mathsf{ref}\,0, \mathsf{move} = V\rangle\ \mathsf{in}\ x))\\ &\rightarrow (u := 0;\varepsilon \mid (\mathsf{let\,rec}\ x = R\ \mathsf{in}\ x))\\ &\xrightarrow{*} (u := 0;\varepsilon \mid \langle \mathsf{pos} = u\,,\ \mathsf{move} = \{x \mapsto O\}V\rangle\end{aligned}$$

where $O = (\mathsf{let\,rec}\ x = R\ \mathsf{in}\ R)$. One can see that there are two parts in this evaluated object: a state part, which records the (mutable) position of the object, and the (recursive, immutable) record of methods. Moreover, the state can only be accessed using the methods. Now imagine that we want to enhance the point class with a clear method that resets the position to the origin. Then we introduce a new class inheriting from point:

$$\mathsf{let}\ \mathsf{point}' = \lambda x\lambda\mathsf{self}\langle(\mathsf{point}\,x)\mathsf{self}\,,\ \mathsf{clear} = ((\mathsf{set}\ \mathsf{self.pos})0)\rangle\ \mathsf{in}\ldots$$

However, we cannot create an object instance of that class. More precisely, the type system will reject an expression like $\mathsf{fix}(\mathsf{point}'\,0)$, and rightly so. Indeed, if we try to compute this expression, we get stuck in $(u := 0;\varepsilon \mid (\mathsf{let\,rec}\ x = \mathbf{E}[x]\ \mathsf{in}\ x)$ where $\mathbf{E} = \langle \mathsf{pos} = u\,,\ \mathsf{move} = V\,,\ \mathsf{clear} = ((\mathsf{set}\ [].\mathsf{pos})0)\rangle$. In the clear method, the self parameter ought to be protected from being evaluated, by defining this method as a "thunk" $\mathsf{clear} = \lambda y((\mathsf{set}\ \mathsf{self.pos})0)$. This is the main technical point of the paper: to create objects instance of some class, we must be able to sometimes accept, sometimes reject terms of the form $(\mathsf{let\,rec}\ x = (Gx)\ \mathsf{in}\ N)$, in particular when $G \xrightarrow{*} \lambda\mathsf{self}M$, depending on whether the function (with side effects) G is "protective" towards its argument or not.

In order to establish a type safety result, we need to analyse the possible behaviour of expressions under evaluation: computing an expression may end on a value, or may go forever, but there are other possibilities – in particular an expression may "go wrong" [24] (or "be faulty", following the terminology of [32]). Our analysis is slightly non-standard here, since we have to deal with open terms. Besides the faulty expressions, we distinguish what we call "global redexes" and "head expressions", where a variable appears in a position where it has to be evaluated, and where something has to be done with its value.

Definition 0.1. *A term M is a* global redex *if M is $\mathbf{E}[(\mathsf{ref}\,V)]$, or $\mathbf{E}[(\mathbf{!}u)]$, or else $\mathbf{E}[((\mathsf{set}\,u)V)]$ for some value V and location u, with $\mathsf{fv}(V) \cap \mathsf{capt}(\mathbf{E}) = \emptyset$.*

Definition 0.2. *A term M is a* head expression *if $M = \mathbf{H}[x]$ with $x \notin \mathsf{capt}(\mathbf{H})$, where the $\mathbf{H}$ contexts are given as follows:*

$$\mathbf{H} ::= \mathbf{E}[([]V)] \mid \mathbf{E}[(\mathbf{!}[])] \mid \mathbf{E}[(\mathsf{set}[])] \mid \mathbf{E}[([].\ell)] \mid \mathbf{E}[([]\backslash\ell)]$$

Definition 0.3. *A term M is* faulty *if it contains a sub-expression of one of the following forms:*

(i) (VN), *where V is either a location, or $()$, or a record value;*

(ii) $(\mathsf{let\,rec}\ x = \mathbf{H}[x]\ \mathsf{in}\ M)$ *with $x \notin \mathsf{capt}(\mathbf{H})$*

(iii) $(\mathsf{let\,rec}\ x = \mathbf{E}[N]\ \mathsf{in}\ M)$ *where N is either $(\mathsf{ref}\,V)$ or $((\mathsf{set}\,u)V)$ with $x \in \mathsf{fv}(V)$;*

(iv) $(!V)$ *or* $(\mathsf{set}\,V)$ *where* V *is neither a variable nor a location;*
(v) $\langle V, \ell = N\rangle$ *where* V *is not a record value;*
(vi) $(V.\ell)$ *or* $(V\backslash\ell)$, *where* V *is neither a variable, nor a non-empty record-value.*

Then our first result is:

Proposition 0.4. *For any expression* M*, either* M *reduces, i.e.* $M \to M'$ *for some* M'*, or* M *is a head expression, or a faulty expression, or a global redex, or a value.*

Corollary 0.5. *For any closed configuration* $(S \mid M)$ *such that any location occurring in* M *is in* $\mathsf{dom}(S)$*, either its evaluation does not terminate, or it ends with* $(S' \mid N)$ *where* N *is either faulty or a value.*

3 The Type System

The aim in using a type system is to prevent run-time errors – and also to provide some interesting information about expressions. Then we have to design such a system in a way that rules out faulty expressions. The only case which is non-standard is the one of $\mathsf{let\ rec}$ expressions, that is (ii) and (iii) of Definition 0.3. To exclude unsafe recursion, we will use "decorated types", where the decorations, also called "harmlessness degrees" are boolean values 0 or 1 (with $0 \leqslant 1$), to which we must add, in order to obtain principal types, degree variables p, $q\ldots$ We denote by d, $e\ldots \in \mathcal{D}$ these *degrees*, either constant or variable.

Following Milner [24], we use a polymorphic let construct. This is crucial for defining classes that may be inherited in various ways, and instantiated into objects. Then we will use *type schemes*. As in [19], we do not allow the same label to occur several times in a given record type – but our treatment of row variables is quite different from the one of [19]. Therefore, in quantifying on a row variable, we must take into account the context in which it occurs, by means of the set L of labels that it must not contain. We call such a finite set of labels an *annotation*[1], and we have a simple "annotating" system to ensure that (record) types are well-formed. Given a set $\mathcal{T\!yV\!ar}$ of type variables, the syntax of types and type schemes is:

$$\begin{array}{rcl} \tau,\ \theta\ldots & ::= & \mathsf{unit} \mid t \mid (\theta^d \to \tau) \mid \tau\,\mathsf{ref} \mid \rho \\ \rho & ::= & t \mid \lozenge \mid \langle \rho, \ell : \tau\rangle \\ \sigma,\ \varsigma\ldots & ::= & \tau \mid (\forall C.\sigma) \end{array}$$

where t is any type variable, d is any degree, and $C = t_1 :: L_1, \ldots, t_n :: L_n$. The annotation of type variables is not the only constraint we have to take into account in the type system: we also have to deal with constraints on degrees, that take the form of a set of inequalities $p \leqslant a$ where p is a degree variable and a is a *degree expression*, built from degrees by using the meet operation $\sqcap$. We denote by a, b, $c\ldots \in \mathcal{DE}xp$ these expressions. In order to give a simple form

[1] we borrow this terminology from Fisher's thesis [14].

to typing rules, we group the two kinds of constraints into a single component, called a *constraint*, still denoted by C. Notice that the constraints on degrees are obviously satisfiable, e.g. assigning uniformly 0 to the degree variables. For lack of space, we omit the constraint system, by which one can prove that a type is well-formed and does not posses some given labels, in notation $C \vdash \sigma :: L$, and that a degree inequality is a consequence of the given constraint, denoted $C \vdash a \leqslant b$ (the corresponding inference systems are quite trivial).

As usual, we need the notion of an instance of a type scheme, obtained by substituting not only types for type variables, but also degrees for degree variables. Then a type and degree substitution S is a mapping from type variables to types, and from degree variables to degrees (not degree expressions), which is the identity, except for a finite set $\mathsf{dom}(\mathsf{S})$ of variables. Moreover, we need to ensure that applying a substitution to a type scheme, which we denote $\mathsf{S}(\sigma)$, results in a well-formed type. Given two constraints C_0 and C_1, we then define $\mathcal{S}ub(C_0, C_1)$ as follows:

$$\mathsf{S} \in \mathcal{S}ub(C_0, C_1) \quad \Leftrightarrow_{\mathrm{def}} \quad \mathsf{dom}(\mathsf{S}) \subseteq \mathsf{dom}(C_0) \ \&\ C_1 \vdash \mathsf{S}(C_0)$$

where $\mathsf{S}(C) = \{\, \mathsf{S}(t) :: L \mid t :: L \in C \,\} \cup \{\, \mathsf{S}(p) \leqslant \mathsf{S}(a) \mid p \leqslant a \in C \,\}$. Then for instance the standard relation of being a *generic instance* (see [12]) is relative to some constraint, and is written $C \vdash \sigma \preceq \varsigma$.

The typing judgements have the form $C\,;\Gamma \vdash M : \tau$, where C is a constraint, τ is a type[2], and Γ is a *typing context*. This maps a finite set $\mathsf{dom}(\Gamma)$ of variables not only to type schemes, but also to degree expressions. The idea is that with a variable x we associate an assumption about the fact that it will or will not occur in a dangerous position, typically $\mathbf{H}[x]$ (some other cases are mentionned in the point (iii) of Definition 0.3). This assumption is the *harmlessness degree*, or simply the degree of the variable in the context – 0 standing for "dangerous". We also need to type locations, and therefore a context is a mapping from a finite set of variables to pairs (σ, a), written σ^a, and from a finite set of locations to types. We shall write $\Gamma_{\mathsf{typ}}(x) = \sigma$ and $\Gamma_{\mathsf{deg}}(x) = a$ if $\Gamma(x) = \sigma^a$, and similarly $\Gamma_{\mathsf{typ}}(u) = \Gamma(u)$. To simplify the presentation of the system, we omit the obvious side conditions by which the types and contexts introduced in the axioms are well-formed with respect to the constraint C. We use the following predicate and operations on the typing contexts:

(i) $C \vdash \Delta \leqslant \Gamma$ if and only if $\Gamma_{\mathsf{typ}} = \Delta_{\mathsf{typ}}$ and $C \vdash \Gamma_{\mathsf{deg}}(x) \leqslant \Delta_{\mathsf{deg}}(x)$ for all x;

(ii) let δ be a mapping from variables to degrees. Then we define the context Γ^δ as follows: $(\Gamma^\delta)_{\mathsf{typ}} = \Gamma_{\mathsf{typ}}$ and $(\Gamma^\delta)_{\mathsf{deg}}(x) = \delta(x)$. We let $\Gamma^{\lceil\delta\rceil}$ denote $\Gamma^{\gamma\sqcap\delta}$, where $\gamma = \Gamma_{\mathsf{deg}}$.

We mainly use these last notations when δ is $\lambda x(\mathsf{if}\ x \in \mathsf{fv}(M)\ \mathsf{then}\ a\ \mathsf{else}\ 1)$, which is abbreviated into a_M. We also abusively write 1 for $\lambda x.1$, and similarly for 0.

Now let us comment on some of the rules that are presented in Figure 5. The first one is a "degree weakening" rule, stating that "optimistic" assumptions,

[2] to simplify the presentation we do not include the usual rules of instantiation and generalization (see [12]), but they would easily be shown to be admissible if judgements $C\,;\Gamma \vdash M : \sigma$ were allowed, and therefore we will use them in the examples.

$$\frac{C\,;\Gamma\vdash M:\tau\ ,\ C\vdash\Delta\leqslant\Gamma}{C\,;\Delta\vdash M:\tau}\qquad\frac{C\vdash\sigma\succeq\tau}{C\,;x:\sigma^1\,,\,\Gamma\vdash x:\tau}\qquad\frac{}{C\,;u:\tau\,,\,\Gamma\vdash u:\tau\,\mathsf{ref}}$$

$$\frac{C\,;x:\theta^d\,,\,\Gamma\vdash M:\tau}{C\,;\Gamma^1\vdash\lambda xM:(\theta^d\to\tau)}\qquad\frac{C\,;\Gamma\vdash M:\theta^d\to\tau\ ,\ C\,;\Gamma\vdash N:\theta}{C\,;\Gamma^{\lceil 0_M\sqcap d_N\rceil}\vdash(MN):\tau}$$

$$\frac{C'\,,\,C\,;\Gamma\vdash N:\theta\ ,\ C\,;x:(\forall C'.\theta)^a\,,\,\Gamma\vdash M:\tau}{C\,;\Gamma^{\lceil a_N\rceil}\vdash(\mathsf{let}\ x=N\ \mathsf{in}\ M):\tau}\ (*)$$

$$\frac{C'\,,\,C\,;x:\theta^1\,,\,\Gamma\vdash N:\theta\ ,\ C\,;x:(\forall C'.\theta)^a\,,\,\Gamma\vdash M:\tau}{C\,;\Gamma^{\lceil a_N\rceil}\vdash(\mathsf{let\,rec}\ x=N\ \mathsf{in}\ M):\tau}\ (*)$$

$$\frac{}{C\,;\Gamma\vdash\mathsf{ref}:\tau^0\to\tau\,\mathsf{ref}}\qquad\frac{}{C\,;\Gamma\vdash\,!:(\tau\,\mathsf{ref})^0\to\tau}$$

$$\frac{}{C\,;\Gamma\vdash\mathsf{set}:(\tau\,\mathsf{ref})^0\to\tau^0\to\mathsf{unit}}\qquad\frac{}{C\,;\Gamma\vdash():\mathsf{unit}}$$

$$\frac{}{C\,;\Gamma\vdash\langle\rangle:\langle\rangle}\qquad\frac{C\,;\Gamma\vdash M:\rho\ ,\ C\,;\Gamma\vdash N:\tau\ ,\ C\vdash\rho::\{\ell\}}{C\,;\Gamma\vdash\langle M,\ell=N\rangle:\langle\rho,\ell:\tau\rangle}$$

$$\frac{C\,;\Gamma\vdash M:\langle\rho,\ell:\tau\rangle}{C\,;\Gamma^{0_M}\vdash(M.\ell):\tau}\qquad\frac{C\,;\Gamma\vdash M:\langle\rho,\ell:\tau\rangle}{C\,;\Gamma^{0_M}\vdash(M\backslash\ell):\rho}$$

$$\frac{}{C\,;\Gamma\vdash\varepsilon}\qquad\frac{C\,;u:\tau\,,\,\Gamma\vdash V:\tau\ ,\ C\,;u:\tau\,,\,\Gamma\vdash S}{C\,;u:\tau\,,\,\Gamma\vdash u:=V;S}\qquad\frac{C\,;\Gamma\vdash S\quad ,\quad C\,;\Gamma\vdash M:\tau}{C\,;\Gamma\vdash(S\mid M):\tau}$$

$(*)$ where $t\in\mathsf{dom}(C')\ \Rightarrow\ t\notin C,\Gamma$ and C' is empty if N is neither a value nor $(\mathsf{let\,rec}\ x=V\ \mathsf{in}\ V)$.

Fig. 5. The Type System

assigning for instance degree 1 to some variables, can always be safely downgraded. Notice that a variable in isolation is harmless: indeed the evaluation of $(\mathsf{let\,rec}\ x=x\ \mathsf{in}\ x)$ diverges, hence does not result in a run-time error. In the rule for abstraction of x, we assume that the degree of x does not contain the $\sqcap$ operation, but this is not a restriction, since we may always add a fresh constraint $p\leqslant a$ and use the weakening rule. The rule for abstraction promotes the typing context to a definitely harmless one (Γ^1), since all the variables occurring in the abstraction value are protected from being evaluated by the λ (notice that this holds for record values too). Conversely, the variables occurring in the function part of an application are potentially dangerous, like for instance x in $(\lambda y.xy)V$. Then they are all downgraded to having the degree 0. Regarding the argument, we must be more careful: applying a function of type $\theta^1 \rightarrow \tau$ that does not put its argument in danger, like λxx for instance, we may keep the degree of its free variable as it is. More generally, applying a function of type $\theta^d \rightarrow \tau$ places the argument in a position where the variables have a degree which is, at best, d or the degree they have in the argument. This is where we use the $\sqcap$ operation. For instance, we have, if $C = t :: \emptyset, t' : :: \emptyset, p \leqslant 0, q \leqslant r$:

$$\dfrac{\dfrac{\dfrac{\vdots}{C\,;\, f : (t^r \rightarrow t')^0\,,\, x : t^{r \sqcap 1} \vdash fx : t'}}{C\,;\, f : (t^r \rightarrow t')^p\,,\, x : t^r \vdash fx : t'}\; p \leqslant 0,\ q \leqslant r \sqcap 1}{C;\vdash \lambda fx.fx : (t^r \rightarrow t')^p \rightarrow t^q \rightarrow t'}$$

To see why we need the meet operation, the reader may try to type $f(gx)$, where the degree of x depends on the nature of both f and g. The rule for the let rec construct is the only one involving a real – i.e. possibly unsatisfiable – constraint on degrees, namely $1 \leqslant a$. It is exemplified by the following typing of the fixpoint combinator, where $\Gamma = f : (t^1 \rightarrow t)^1, x : t^1$ and $\Delta = f : (t^1 \rightarrow t)^0, x : t^1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{t :: \emptyset\,;\, \Gamma \vdash f : (t^1 \rightarrow t)} \quad \dfrac{}{t :: \emptyset\,;\, \Gamma \vdash x : t}}{t :: \emptyset\,;\, \Delta \vdash fx : t} \qquad \dfrac{\dfrac{}{t :: \emptyset\,;\, \Gamma \vdash x : t}}{t :: \emptyset\,;\, \Delta \vdash x : t}}{t :: \emptyset\,;\, f : (t^1 \rightarrow t)^0 \vdash (\mathsf{let\,rec}\ x = fx\ \mathsf{in}\ x) : t}}{t :: \emptyset; \vdash \mathsf{fix} : (t^1 \rightarrow t)^0 \rightarrow t}$$

Notice that, as in ML, (let rec $f = \lambda xN$ in M) is always allowed, provided that M and N have appropriate typings. The functional core of the language concentrates all the subtelties of the use of degrees – the rest of the type system is quite trivial, and in particular there is not much choice in the typing of the record constructs.

There is a general rule governing typing, which is that all the variables that occur in an expression placed in a head position, that is in the hole in an **H** context, are regarded as potentially dangerous, as we have seen with the application construct. This explains the resulting downgraded typing context in the rules for selection and restriction. The clauses (ii) and (iii) of Definition 0.3 also indicate that referencing, de-referencing and assignment are not protective operations. Considering these explanations, one should not be surprised that the following holds:

LEMMA 0.6. *The faulty expressions are not typable.*

Then we have the standard "type preservation" property, which relies, as usual, on a property relating typing and substitution:

Proposition (Type Preservation) 0.7. *If $C\,;\Gamma \vdash M : \tau$ and $M \xrightarrow{*} N$ then $C\,;\Gamma \vdash N : \tau$.*

Proving a similar property for global reduction, and then combining these results with the Corollary 0.5 and the Lemma 0.6, we get:

Theorem (Type Safety) 0.8. *For any typable closed configuration $(S \mid M)$ of type τ such that any location occurring in M is in $\mathsf{dom}(S)$, either its evaluation does not terminate, or it ends with $(S' \mid V)$ where V is a value of type τ.*

This is our first main result. The second one is that if an expression M is typable, then it has a computable principal type, of which any other type of M is an instance:

Theorem (Principal Type) 0.9. *There is an algorithm that, given Γ_{typ} and M, fails if M is not typable in the context Γ^{γ} for some γ, and otherwise returns a type τ, a degree assignment δ and a constraint C such that $C\,;\Gamma^{\delta} \vdash M : \tau$, and if $C'\,;\Gamma^{\gamma} \vdash M : \tau'$ then $\tau' = \mathsf{S}(\tau)$ and $C' \vdash \Gamma^{\gamma} \leqslant \mathsf{S}(\Gamma^{\delta})$ for some substitution $\mathsf{S} \in \mathcal{S}ub(C, C')$.*

To perform type reconstruction we have, as usual, to solve equations on types, by means of unification. Using a "strict" record extension operation (and similarly a "strict" restriction), rather than the one originally used by Wand [30], which combines extension with overriding, allows one to solve these equations, up to annotation constraints, in a simple way, as shown by Jategaonkar and Mitchell [19]. As a matter of fact, we also have to solve some degree equalities $d = e$ arising from $\theta_0^d \to \tau_0 = \theta_1^e \to \tau_1$, but these are quite trivial (if d and e are both constants, then the equality is trivially true or false, and if one is a variable, the equality can be treated as a substitution). To construct a type for the $\mathsf{let\,rec}$ construct, we have to solve equations $a = 1$ (or $1 \leqslant a$), but these are also easy to solve, since $(a \sqcap b) = 1$ is equivalent to $a = 1$ and $b = 1$. The only case of failure arising from degrees is when we have to solve $0 = 1$, that is when a potentially dangerous recursion is detected.

4 Some Examples

In this section we illustrate the expressive power of our calculus, as regards object-orientation, both from an operational and from a typing point of view. Let us first see how to type the "point class" previously defined – assuming that $+$ is of type $\mathsf{int}^0 \to \mathsf{int}^0 \to \mathsf{int}$. We recall that this class is given by

$$\begin{aligned}\mathsf{point} = \lambda x \lambda \mathsf{self}\langle &\mathsf{pos} = \mathsf{ref}\, x,\\ &\mathsf{move} = \lambda y((\mathsf{set\ self.pos})(!\mathsf{self.pos} + y))\rangle\end{aligned}$$

In the record of methods of the point class, the x parameter may have any type, but it is placed in a dangerous position, being an argument of ref. Then it has type t^0, where t is a type variable. Regarding the self parameter, we see from its use in the move method that it must have a record type, containing a field pos, of type $\mathsf{int\,ref}$. Moreover, self only occurs within a value, and therefore it has type $\langle s, \mathsf{pos} : \mathsf{int\,ref}\rangle^p$, with no constraint on p, where the row variable s must not contain the pos field. Then, abbreviating ($\forall t :: \emptyset.\sigma$) into ($\forall t.\sigma$), the (polymorphic) type of point is:

$$\mathsf{point} : \forall t.\forall s :: \{\mathsf{pos}\}.t^0 \to \langle s, \mathsf{pos} : \mathsf{int\,ref}\rangle^p \to \langle \mathsf{pos} : t\,\mathsf{ref}\,,\, \mathsf{move} : \mathsf{int}^0 \to \mathsf{unit}\rangle$$

This type may be used in ($\mathsf{let\ point} = P\ \mathsf{in}\ \cdots$) since the class P is a value, being a function. Following Wand [31], for a class $A = \lambda x_1 \ldots x_n \lambda z.M$ with instance variables $x_1, \ldots , x_n$ and self parameter z, where M is a record expression, we may denote by $\mathsf{new}\,A(N_1, \ldots , N_n)$ the expression $\mathsf{fix}(AN_1 \cdots N_n)$, which creates an object instance of the class A. Notice that in typing A, the type of the self parameter z has a priori no relation with the type of the body M of the class: we only have to arrange that z has appropriate type with respect to its uses in M (see the point example). Now to create an object of class A, we have to solve some constraints on the type of the self parameter, since, as we have seen, the fixpoint fix has (principal) type $\mathsf{fix} : \forall t.(t^1 \to t)^0 \to t$. Then for instance, assuming that 0 has type int, to type the expression $\mathsf{new\,point}(0)$ we have to solve the equation $\langle s, \mathsf{pos} : \mathsf{int\,ref}\rangle = \langle \mathsf{pos} : t\,\mathsf{ref}\,,\, \mathsf{move} : \mathsf{int}^0 \to \mathsf{unit}\rangle$, and the equation $p = 1$. In particular, we have to instantiate s into $\langle \mathsf{move} : \mathsf{int}^0 \to \mathsf{unit}\rangle$ (which obviously satisfies the constraint of not containing a pos field). In the context of previous declarations for fix and point, this gives us the expected type for a point object, that is $\mathsf{new\,point}(0) : \mathsf{Point}$ where

$$\mathsf{Point} = \langle \mathsf{pos} : \mathsf{int\,ref}\,,\, \mathsf{move} : \mathsf{int}^0 \to \mathsf{unit}\rangle$$

As one can see, one can create an object instance of a class only if that class is "protective" towards its self argument. That is, once instantiated with initial values for the instance variables, it must have a type of the form $\theta^1 \to \tau$ (moreover θ – the type of self – and τ – the type of the record of methods – should be unifiable). This is not possible with a clear method with body ((set self.pos)0), for instance, since here self is doomed to have degree 0. One might have the discipline that, if the self parameter occurs free in the body of a method of a class, then this body is a value, but as we shall see, there are other uses of self.

Now let us see how to type a simple inheritance situation, again following Wand's model [31]: to inherit from a class A, a class B is defined as $B = \lambda y_1 \ldots y_k \lambda z\langle (AN_1 \cdots N_n)z \cdots \rangle$ where $\langle (AN_1 \cdots N_n)z \cdots \rangle$ typically consists in extending and modifying the record of methods of class A. This may also be defined using a super variable, representing the current object as a member of the superclass, as in $B = \lambda y_1 \ldots y_k \lambda z(\mathsf{let\ super} = (AN_1 \cdots N_n)z\ \mathsf{in}\ \langle \mathsf{super} \cdots \rangle)$. For example, we define a class of "resetable" points, inheriting from point, as follows

$$\mathsf{rPoint} = \lambda x \lambda \mathsf{self}\langle (\mathsf{point}\,x)\mathsf{self}\,,\, \mathsf{reset} = \lambda y((\mathsf{set\ self.pos})y)\rangle$$

The x and self parameters have the same type here as in point, and the type of the reset method is $\mathsf{int}^0 \to \mathsf{unit}$. Now, as usual, we want more "flashy" points, exhibiting a color, that we can change by painting the object. Since we want this extension to be applicable to a variety of points – or other objects –, it is natural to define a function, taking as argument a class, or more accurately an instantiated class g – that is, a generator [10], which is a function of self only –, and returning an extended class:

$$\mathsf{coloring} = \lambda g \lambda c \lambda \mathsf{self}\langle (g\,\mathsf{self})\,,\, \mathsf{color} = \mathsf{ref}\,c,\, \mathsf{paint} = \lambda y((\mathsf{set\ self.color})y)\rangle$$

The reader will easily check that to type the body of this function, the best is to assume that self has type $\langle s, \mathsf{color}: t'\ \mathsf{ref}\rangle^p$ where p depends on the function g. Then g must be applicable to any record that contains, or may be extended to contain a color field of type $\tau\ \mathsf{ref}$. We let the reader check how to type coloring, with g of type $\langle s, \mathsf{color}: t'\ \mathsf{ref}\rangle^p \rightarrow t''$, as well as the derived classes

$$\mathsf{cPoint} = \lambda x(\mathsf{coloring}(\mathsf{point}\, x))$$
$$\mathsf{cRPoint} = \lambda x(\mathsf{coloring}(\mathsf{rPoint}\, x))$$

Notice that since coloring requires that self has a color field, the point and rPoint classes are used here with an instantiated type for self, namely replacing s by $\langle s', \mathsf{color}: t'\ \mathsf{ref}\rangle$. The polymorphism offered by Wand's row variables is crucial here for the inheritance mechanism to work properly, where one usually employs some form of subtyping (see [7,16]). The coloring function may be regarded as an example of what has been called a *mixin*, that is a class definition parameterized over its superclass (see [5]), which bears some similarity with the parameterized classes of EIFFEL [23] and the "virtual classes" of BETA [22]. This notion of a mixin has recently received some attention, see for instance [2,4,17].

Continuing with the same example, one may have wished, in defining a colored "resetable" point, to modify the reset method so that not only the position, but also the color may be reset. Then we may define this by means of a "wrapper" [10], that is a function of super and self parameters:

$$W = \lambda\mathsf{super}\lambda\mathsf{self}\langle\mathsf{super}\,,\ \mathsf{reset} \leftarrow \lambda y\lambda d(\mathsf{super.reset}\, y)\ ;\ (\mathsf{super.paint}\, d)\rangle$$
$$\mathsf{CRPoint} = \lambda x\lambda c\lambda\mathsf{self}(\mathsf{let\ super} = ((\mathsf{cRPoint}\, xc)\,\mathsf{self})\ \mathsf{in}\ (W\,\mathsf{super})\mathsf{self})$$

It is interesting to notice that here the reset method is redefined to have a type which is unrelated to the one it has in the superclass. This is a kind of inheritance which is usually not supported in object-oriented calculi and languages (except, obviously, untyped languages like SMALLTALK). This is not problematic here since no other method of the superclass was using reset.

The fact that classes and objects are "first class citizens" in our core language allows one not only to pass them as arguments and return them as results, as in the "mixin" or "wrapper" facilities, but also to imagine new schemes of "inheritance" – or more generally code reuse, which in our calculus manifests itself through the use of the let construct. For instance, one may build a class as an instance of another class, by fixing the initial value of some instance parameters, like in the specialization of the point class into the class $\mathsf{oPoint} = \lambda\mathsf{self}.(\mathsf{point}\,0)\mathsf{self}$ of the points initially at the origin. One may also decide to dynamically introduce a class of unicolored points, by disallowing the use of the paint method:

$$\mathsf{uCPoint} = \lambda x\lambda c\lambda\mathsf{self}((\mathsf{cPoint}\, xc)\mathsf{self}\backslash\mathsf{paint})$$

We can create an object of that class, since the type of self in $(\mathsf{cPoint}\, xc)\mathsf{self}$ is not required to contain the paint method. Similarly, one can restrict a method to be private to an object; for instance, if we let $o = \mathsf{new}\,\mathsf{point}(0)\backslash\mathsf{pos}$ then this object behaves as a point, except that it does not accept pos messages. We could even do that with a method $\mathsf{hidePos} = \lambda y(\mathsf{self}\backslash\mathsf{pos}\backslash\mathsf{hidePos})$. Such a facility does not seem to be supported by the self-application semantics of objects. Some similar examples of excluding methods, like for instance building a class of stacks from a class of dequeues, were given long ago by Snyder [28]. Clearly, introducing such reuse mechanisms would reinforce the fact that "inheritance is not subtyping" [11]. In the full version of the paper we give some further examples, and discuss related work.

5 Conclusion

In this paper we have adapted and extended Wand's typed model of classes and objects [31] to an imperative setting, where the state of an object is a set of mutable values. Our main achievement is the design of a type system which only accepts safe let rec declarations, while retaining the ability to construct a principal type for a typable term. We believe that our type system does not impose any new restriction on the underlying language, where recursion is limited to ($\mathsf{let\,rec}\ x = N\ \mathsf{in}\ M$) where N is a value: it should not be difficult to show that a term of this language is typable, without using degrees, if and only if it is typable, with the "same" type, in our system, thus showing that our typing is a conservative extension of the usual one, if we forget about degrees. Type reconstruction in our system is based upon solving very simple equations on degree expressions, as we have indicated, and therefore this should not complicate the standard algorithm. This issue has to be investigated from a pragmatic point of view, to see whether our solution is practically useful. This is left for further work, but we hope we have at least suggested that our calculus is very expressive, especially regarding object-oriented programming constructs. We could then use it as a guideline to design a type safe object-oriented layer.

References

[1] M. Abadi, L. Cardelli, *A Theory of Objects,* Springer-Verlag (1996).

[2] D. Ancona, E. Zucca, *A theory of mixin modules: basic and derived operators,* Math. Struct. in Comput. Sci. Vol. 8 (1998) 401-446.

[3] V. Bono, A. Patel, V. Shmatikov, J. Mitchell, *A core calculus of classes and objects,* MFPS'99, Electronic Notes in Comput. Sci. Vol. 20 (1999).

[4] V. Bono, A. Patel, V. Shmatikov, J. Mitchell, *A core calculus of classes and mixins,* ECOOP'99, Lecture Notes in Comput. Sci. 1628 (1999) 43-66.

[5] G. Bracha, W. Cook, *Mixin-based inheritance,* ECOOP/OOPSLA'90 (1990) 303-311.

[6] K. Bruce, *Safe type checking in a statically-typed object-oriented programming language,* POPL'93 (1993) 285-298.

[7] K. Bruce, L. Petersen, A. Fiech, *Subtyping is not a good "match" for object-oriented languages,* ECOOP'97, Lecture Notes in Comput. Sci. 1241 (1997) 104-127.

[8] L. Cardelli, *A semantics of multiple inheritance,* Semantics of Data Types, Lecture Notes in Comput. Sci. 173 (1984) 51-67. Also published in Information and Computation, Vol. 76 (1988).

[9] L. Cardelli, J.C. Mitchell, *Operations on records,* in [18], 295-350.

[10] W. Cook, J. Palsberg, *A denotational semantics of inheritance and its correctness,* OOPSLA'89, ACM SIGPLAN Notices Vol. 24 No. 10 (1989) 433-443.

[11] W. Cook, W. Hill, P. Canning, *Inheritance is not subtyping,* in [18], 497-517.

[12] L. Damas, R. Milner, *Principal type-schemes for functional programs,* POPL'82 (1982) 207-212.

[13] J. Eifrig, S. Smith, V. Trifonov, A. Zwarico, *An interpretation of typed OOP in a langage with state,* LISP and Symbolic Computation Vol. 8 (1995) 357-397.

[14] K. Fisher, *Types Systems for Object-Oriented Programming Languages,* PhD Thesis, Stanford University (1996).

[15] K. Fisher, F. Honsell, J. Mitchell, *A lambda calculus of objects and method specialization,* LICS'93 (1993) 26-38.

[16] K. Fisher, J. Mitchell, *The development of type systems for object-oriented languages,* Theory and Practice of Object Systems Vol. 1, No. 3 (1996) 189-220.

[17] M. Flatt, S. Krishnamurthi, M. Felleisen, *Classes and Mixins,* POPL'98 (1998) 171-183.

[18] C. Gunter, J. Mitchell (Eds.), *Theoretical Aspects of Object-Oriented Programming,* The MIT Press (1994).

[19] L. A. Jategaonkar, J. Mitchell, *Type inference with extended pattern matching and subtypes,* Fundamenta Informaticae Vol. 19 (1993) 127-166.

[20] S. Kamin, *Inheritance in* Smalltalk-*80: a denotational definition,* POPL'88 (1988) 80-87.

[21] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, J. Vouillon, *The Objective Caml System, release* 3.00, Documentation and user's manual, available at `http://caml.inria.fr` (2000).

[22] O. L. Madsen, B. Møller Pedersen, *Virtual Classes: A powerful mechanism in object-oriented programming,* OOPSLA'89, ACM SIGPLAN Notices Vol. 24 No. 10 (1989) 397-406.

[23] B. Meyer, *Genericity versus inheritance,* OOPSLA'86, ACM SIGPLAN Notices Vol. 21 No. 11 (1986) 391-405.

[24] R. Milner, *A theory of type polymorphism in programming,* J. of Computer and System Sciences Vol. 17 (1978) 348-375.

[25] B. C. Pierce, D. Turner, *Simple type-theoretic foundations for object-oriented programming,* J. of Functional Programming Vol. 4 No. 2 (1994) 207-247.

[26] D. Rémy, *Programming with ML-ART: an extension to ML with abstract and record types,* TACS'94, Lecture Notes in Comput. Sci. 789 (1994) 321-346.

[27] D. Rémy, J. Vouillon, *Objective ML: an effective object-oriented extension of ML,* Theory and practice of Objects Systems, Vol. 4, No. 1 (1998) 27-50.

[28] A. Snyder, *Encapsulation and inheritance in object-oriented programming languages,* OOPSLA'86, ACM SIGPLAN Notices Vol. 21 No. 11 (1986) 38-45.

[29] A. Taivalsaari, *On the notion of inheritance,* ACM Computing Surveys Vol. 28 No. 3 (1996) 438-479.

[30] M. Wand, *Complete type inference for simple objects,* LICS'87 (1987) 37-44.

[31] M. Wand, *Type inference for objects with instance variables and inheritance,* in [18], 97-120.

[32] A. Wright, M. Felleisen, *A syntactic approach to type soundness,* Information and Computation Vol. 115 No. 1 (1994) 38-94.