A Framework for Defining Object-Calculi

Extended Abstract

Frédéric Lang, Pierre Lescanne, and Luigi Liquori

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme
46, Allée d'Italie, F–69364 Lyon Cedex 07, FRANCE
E-mail: {flang,plescann,lliquori}@ens-lyon.fr

Abstract. In this paper, we give a general framework for the foundation of an operational (small step) semantics of object-based languages with an emphasis on functional and imperative issues. The framework allows classifying very naturally many object-based calculi according to their main implementation techniques of inheritance, namely *delegation* and *embedding*, and their particular strategies. This distinction comes easily from a choice in the rules. Our framework is founded on two previous works: $\lambda\mathcal{O}bj^{+}$, a version of the *Lambda Calculus of Objects* of Fischer, Honsell, and Mitchell, for the object aspects, and $\lambda\sigma_w^a$ of Benaissa, Lescanne, and Rose, for the description of the operational semantics and sharing. The former is the formalization of a small delegation-based language which contains both lambda calculus and object primitives to create, update, and send messages to objects, while the latter is designed to provide a generic description of functional language implementations and is based on a calculus of explicit substitution extended with *addresses* to deal with memory management. The framework is presented as a set of *modules*, each of which captures a particular aspect of object-calculi (functional *vs.* imperative, delegation *vs.* embedding, and any combination of them). Above all, it introduces and illustrates a new promising approach to formally reason about the operational semantics of languages with (possibly) mutable states.

Keywords. Design of functional and imperative object-oriented languages, operational semantics, implementation issues, memory management.

1 Introduction

An (operational) semantics for a programming language is aimed to help the programmer and the designer of a compiler to better understand her (his) work and possibly to prove mathematically that what she (he) does is correct. For instance, the designers of Java proposed a description of an operational semantics of the Java Virtual Machine [16], but unfortunately its informal character does not fulfill the above aim. In this paper, we set the foundation for a formal description of the operational semantics (small step) of object-based languages. One main characteristic of our framework, called $\lambda\mathcal{O}bj^{+a}$, is that it induces an easy *classification* of the object-based languages and their semantics, making a clear distinction between functional and imperative languages. Moreover, the present formal system is *generic*, which means that it presents many semantics in one framework which can be instantiated to conform to specific wishes. For this, it proposes a set of several *modules*, each of which captures a particular aspect of object-calculi (functional *vs.* imperative[1], delegation *vs.* embedding, and any combination of them). Genericity comes also from a total *independence from the strategy*, the latter being sometimes crucial when dealing with imperative semantics.

The framework $\lambda\mathcal{O}bj^{+a}$ describes both static and dynamic aspects of object-oriented languages. *Static* aspects are the concepts related to the program, namely its syntax, including variable scoping, and above all its type system. The type system (not presented in this paper for obvious lack of space) avoids the unfortunate run-time error `message-not-understood`, obtained when one sends a message, say `m`, to an object which has no `m` in its protocol. *Dynamic* aspects are related to its behavior at run-time *i.e.*, its operational semantics, also known as the implementation choices. In addition, this paper introduces in the world of the formal operational semantics of object-based languages, the concepts of *addresses* and *simultaneous rewriting*, which differ from the classical *match and replace* technique of rewriting.

"Road Map". Section 2 sets the context of the framework $\lambda\mathcal{O}bj^{+a}$. Section 3 addresses mostly the implementation aspects of object-based languages. Section 4 introduces ancestors of $\lambda\mathcal{O}bj^{+a}$, namely $\lambda\mathcal{O}bj^{+}$, a slightly modified version of the Lambda Calculus of Objects, and $\lambda\sigma_w^a$, a weak lambda-calculus with explicit substitution and addresses. Section 5 is the real core of the paper as it details the notion of simultaneous rewriting, and presents $\lambda\mathcal{O}bj^{+a}$ through its four modules L, C, F, and I. Section 6 gives some examples motivating our framework. Finally, Section 7 compares our framework with some related works.

A richer version of this paper (containing some open problems) can be found in [14].

2 The Context of $\lambda\mathcal{O}bj^{+a}$

The framework $\lambda\mathcal{O}bj^{+a}$ is founded on an *object-based calculus*, enriched with *explicit substitution* and *addresses*. We explain this in the current section.

2.1 Object-based Calculi

The last few years have addressed the foundation of object-oriented languages. The main goal of this research was twofold: to understand the operational behaviour of object-oriented languages and to build *safe and flexible* type systems which analyze the program text before execution. In addition (and not in contrast) to the traditional class-based view, where *classes* are seen as the primitive notion to build object instances, recent years have seen the development of the, so called, *object-based* (or *prototype-based*) languages. Object-based languages can be either viewed as a novel object-oriented style of programming (such as in Self [22], Obliq [6], Kevo [19], Cecil

[1] The terminology "functional" and "imperative" seems to be more or less classical in the scientific object-oriented community. However, we could use "calculi of non-mutable (resp. mutable) objects" as synonymous for functional (resp. imperative) calculi.

[7], O-$\{1,2,3\}$ [1]) or simply as a way to implement the more traditional class-based languages. In object-based languages there is no notion of class: the inheritance takes place at the object level. Objects are built "from scratch" or by inheriting the methods and fields from other objects (sometimes called *prototypes*). Most of the theoretical papers on object-based calculi address the study of *functional object-calculi*; nevertheless, it is well-known that object-oriented programming is inherently "imperative" since it is based on a notion of "mutable state". However, those papers are not a simple exercise of style, since, as stated in [1, 5], it may happen that a type system designed for a functional calculus can be "well fitted" for an imperative one. Among the proposals firmly setting the theoretical foundation of object-oriented languages, two of them have spurred on an intense research.

The Object Calculus of Abadi and Cardelli [1], is a calculus of *typed objects* of *fixed size* in order to give an account of a standard notion of subtyping. The operations allowed on objects are method invocation and method update. The calculus is computationally complete since the lambda calculus can be encoded via suitable objects. The calculus has both functional and imperative versions, the latter being obtained by simply modifying (with the help of a strategy and suitable data structures) the dynamic semantics of the former. Classes can be implemented using the well-known *record-of-premethods* approach: a class `A` is an object which has a method called `new` creating an instance `a` of the class and a set of "premethods" which become real methods when embedded (*i.e.*, installed) into `a`. Class inheritance can be treated by "reusing" the premethods of the superclass.

The Lambda Calculus of Objects $\lambda\mathcal{O}bj$ of Fisher, Honsell, and Mitchell [11] is an untyped lambda calculus enriched with object primitives. Objects are *untyped* and a new object can be created by modifying and/or extending an existing prototype object. The result is a new object which inherits all the methods and fields of the prototype. This calculus is also (trivially) computationally complete, since the lambda calculus is built in the calculus itself. Classes can also be implemented in $\lambda\mathcal{O}bj$: in a simplified view, a class `A` has a method `new` which first creates an instance `b` of the superclass `B` of `A` and then adds (or updates) this instance with all the fields/methods declared in `A`. In [5], an imperative version of $\lambda\mathcal{O}bj$ featuring an encapsulation mechanism obtained via abstract data types, was introduced. In [9], a modified version of $\lambda\mathcal{O}bj$, called $\lambda\mathcal{O}bj^+$ (see Subsection 4.1), was introduced together with a more powerful type system.

2.2 Explicit Substitution Calculi and Addresses

Explicit Substitution Calculi (see for instance [2, 15]) were invented in order to give a finer description of operational semantics of the functional programming languages. Roughly speaking, an explicit substitution calculus fully includes the *meta substitution* operation as part of the syntax, adding suitable rewriting rules to deal with it. These calculi give a good description of implementations by modeling the concept of *closure*, but do not give an account of the *sharing* needed in lazy functional languages implementations. In [8], a *weak* lambda-calculus of explicit substitution, called $\lambda\sigma_w$, was introduced; here *weak* means a calculus in which one can not compute inside abstractions. In [4], an extension of $\lambda\sigma_w$, called $\lambda\sigma_w^a$, was presented (see Subsection 4.2); this calculus added the notion of *address* and *simultaneous rewriting*, introduced by Rose in his thesis [18]. Addresses are global annotations on terms, which allow to handle sharing of arguments.

2.3 The Framework

The framework $\lambda\mathcal{O}bj^{+a}$ is founded on $\lambda\mathcal{O}bj^{+}$ for the object aspects, to which we add addresses and explicit substitution, following the lines of $\lambda\sigma_w^a$. The reason why we are interested in addresses in the context of an object-based calculus is not only their ability to express sharing of arguments, as in lazy languages, but much more because objects are typical structures which need to be shared in a memory, independently of the chosen strategy.

The framework $\lambda\mathcal{O}bj^{+a}$ deals also with graphs. As a positive consequence of having addresses, the technique of "stack and store" used to design static and dynamic features of imperative languages [10, 20, 24, 1, 5] is substituted in our framework by a technique of graphs (directed and possibly cyclic) which can undergo *destructive updates* through mutation of objects. This makes our framework more abstract, as it involves no particular structure for computing. Moreover it provides a small step semantics of object mutation. This is in fact a generalization of Wadsworth's graph reduction technique of implementation of functional languages [23, 21], which, by essence, forbids destructive updates. Moreover, our graphs are represented as special terms, called *addressed terms*, exploiting the idea of *simultaneous rewriting*, already mentioned in [18, 4], and slightly generalized in this paper (see [13])

The framework $\lambda\mathcal{O}bj^{+a}$ is much more than a simple calculus. One of the consequences of this abstraction is that $\lambda\mathcal{O}bj^{+a}$ allows to define many calculi. A specific calculus is therefore a combination of *modules plus a suitable strategy*. Hence, what makes our approach original are the following features: genericity, independence of the strategy, and capture of both dynamic and static aspects of a given language. Thanks to these features, our framework handles in a unified way and with a large flexibility, functional and imperative object-oriented languages, using both embedding- and delegation-based inheritance techniques, and many different evaluation strategies.

3 Implementation of Object-based Languages

While issues related to the soundness of the various type systems of object-calculi are widely studied in the literature, a few papers address how to build formally a general framework to study and implement inheritance in the setting of object-based calculi. Among the two main categories of object-based calculi (*i.e.*, functional and imperative ones, or with non-mutable and with mutable objects) there are two different techniques of implementation of inheritance, namely the *embedding-based* and the *delegation-based* ones, studied in this section.

The following schematic example will be useful to understand how inheritance can be implemented using the embedding-based and the delegation-based techniques.

Example 1. Consider the following (untyped) definition of a "pixel" prototype.

```
object pixel is
  x:=0; y:=0; onoff:=true;
  set(a,b,c){((self.x:=a).y:=b).onoff:=c}
end
```

Consider the following piece of code.

```
let p=clone(pixel) in
 let q=p.set(a,b,c):={((self.x:=self.x*a).y:=self.y*b).onoff:=c}
  in let r=q.switch():={self.onoff:=not(self.onoff)}
```

where `:=` denotes both a method override and an object extension.

In the following we discuss the two models of implementation of inheritance and we highlight the differences between functional versus imperative models of object-calculi. Before we start, we explain (rather informally) the semantics of the `clone` operator, present in many real object-oriented programming languages.

3.1 The `clone` Operator

The semantics of the `clone` operator changes depending on the delegation-based or embedding-based technique of inheritance, and is *orthogonal* to the functional or imperative features of the framework. In delegation-based inheritance, a `clone` operation produces a "shallow" copy of the prototype *i.e.*, another object-identity which shares the *same* object-structure as the prototype itself. On the contrary, in embedding-based inheritance, a `clone` operation produces a "hard copy" of the prototype, with a proper object-identity and a proper object-structure obtained by "shallowing" and "refreshing" the object-structure of the prototype. This difference will be clear in the next subsections which show possible implementations of the program of Example 1.

3.2 Functional Object-calculi

As known, functional calculi lack a notion of *mutable state*. Although people feel that object-calculi have only little sense in functional setting, we will show in this paper that they are worth studying and that it may be possible to include an object calculus in a pure functional language, like Haskell [17], with much of the interesting features of objects.

Delegation-based Inheritance. The main notion is this of object since there are no classes. Some objects are taken as *prototypical* in order to build other objects. An "update" operation (indicated in the example as `:=`) can either override or extend an object with some fields or methods. A functional update always produces another object, which owns a proper "object-identity" (*i.e.*, a memory location containing a reference to the object-structure, represented as a small square in figures). The result of an update is a "new" object, with a proper object-identity, which shares all the methods of the prototype except the one affected by the update operation. A `clone` operator builds another object with a proper object-identity which shares the structure of the prototype. By looking at Figure 1, one sees how Example 1 can be implemented using a delegation-based technique.

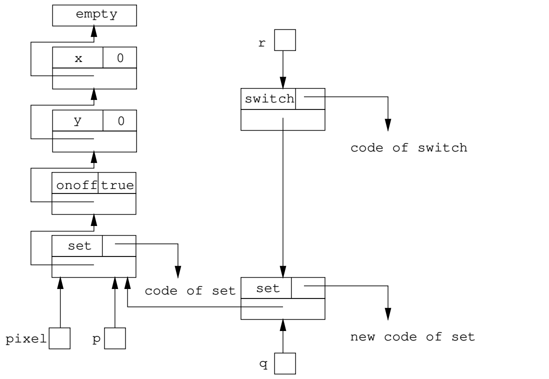

Fig. 1. Functional Delegation-based Inheritance.

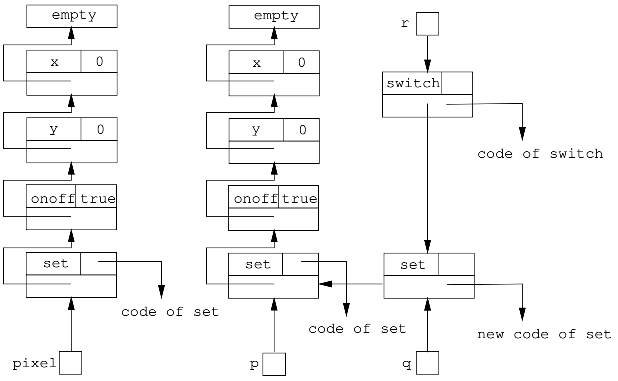

Fig. 2. Functional Embedding-based Inheritance.

Embedding-based Inheritance. In embedding-based inheritance a new object is built by a "hard copy" of the prototype; in fact, `clone` really builds another object with a proper object-identity and a proper copy of the object-structure of the prototype. By looking at Figure 2 one can see how Example 1 can be implemented using an embedding-based technique.

3.3 Imperative Object-calculi

Imperative object-calculi have been shown to be fundamental in describing implementations of class-based languages like Smalltalk and Java. They are also essential as foundations of object-based programming languages like Obliq and Self. The main goal when one tries to define the semantics of an imperative object-based language is to say how an object can be modified while maintaining its object-identity. Particular attention must be paid to this when dealing with object extension. The semantics

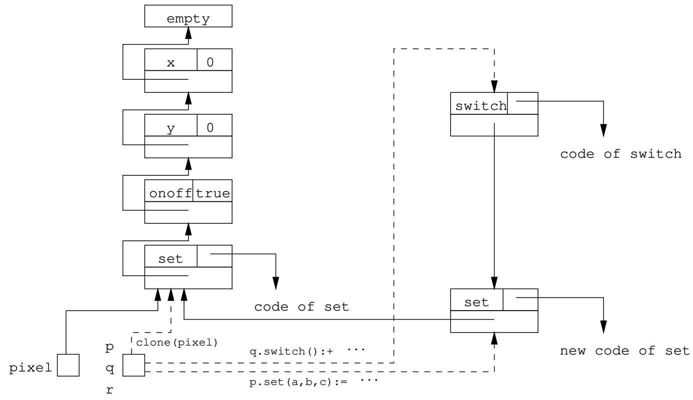

Fig. 3. Imperative Delegation-based Inheritance.

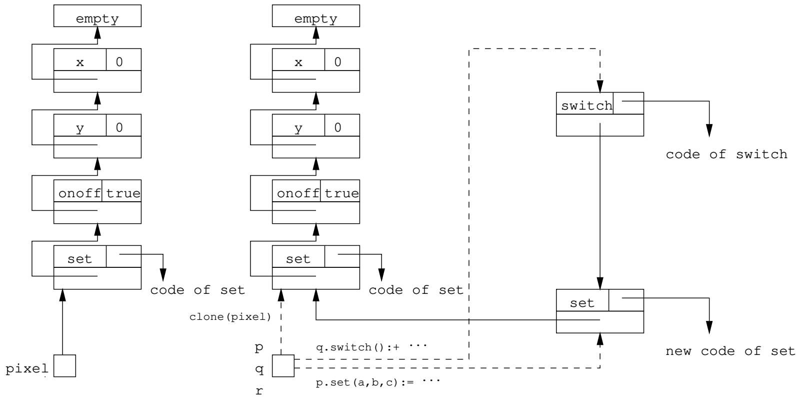

Fig. 4. Imperative Embedding-based Inheritance.

of the imperative update operation is subtle because of side effects. Figure 3 shows the implementation of Example 1 with delegation-based inheritance, and Figure 4 with embedding-based inheritance. Dashed lines represent pointers due to the evaluation of some expression indicated as annotation. Each dashed line cancels the others *i.e.*, there is only one dashed line at each moment. In both cases, observe how the override of the `set` method and the addition of the `switch` method change the object structure of `p` (later on also called `q`, `r`) without changing its object-identity.

4 Some Ancestors of $\lambda\mathcal{O}bj^{+a}$

In this section we give a gentle introduction to calculi which inspired our framework, namely $\lambda\mathcal{O}bj^{+}$ and $\lambda\sigma_w^a$.

Syntax.

$$\begin{aligned} M, N ::= {} & \lambda x.M \mid MN \mid x \mid c && \text{(Lambda Calculus)} \\ & \mid \langle\, \rangle \mid \langle M \leftarrow m = N \rangle \mid M \Leftarrow m && \text{(Objects)} \\ & \mid \mathit{Sel}(M, m, N) && \text{(Auxiliary)} \end{aligned}$$

Operational Semantics.

$$\begin{aligned} (\lambda x.M)\, N &\to M\{N/x\} && \text{(Beta)} \\ M \Leftarrow m &\to \mathit{Sel}(M, m, M) && \text{(Select)} \\ \mathit{Sel}(\langle M \leftarrow m = N \rangle, m, P) &\to N P && \text{(Success)} \\ \mathit{Sel}(\langle M \leftarrow n = N \rangle, m, P) &\to \mathit{Sel}(M, m, P) \quad m \neq n && \text{(Next)} \end{aligned}$$

Fig. 5. The Lambda Calculus of Objects with Self-inflicted Extension $\lambda\mathcal{O}bj^+$.

4.1 The Lambda Calculus of Objects with Self-Extension $\lambda\mathcal{O}bj^+$

The calculus $\lambda\mathcal{O}bj^+$ [9] is a calculus in the style of $\lambda\mathcal{O}bj$. The type system of $\lambda\mathcal{O}bj^+$ allows to type the, so called, "self-inflicted extensions" *i.e.*, the capability of objects to extend themselves upon receiving a message. The syntax and the operational semantics are defined in Figure 5. Observe that the (Beta) rule is given using the *meta substitution* (denoted by $\{N/x\}$), as opposed to the *explicit substitution* used in $\lambda\mathcal{O}bj^{+a}$. The main difference between the syntax of $\lambda\mathcal{O}bj^+$ and that of $\lambda\mathcal{O}bj$ [11] lies in the use of a single operator $\leftarrow$ for building an object from an existing prototype. If the object M contains m, then $\leftarrow$ denotes an object override, otherwise $\leftarrow$ denotes an object extension. The principal operation on objects is method invocation, whose reduction is defined by the (Select) rule. Sending a message m to an object M containing a method m reduces to $\mathit{Sel}(M, m, M)$. The arguments of Sel in $\mathit{Sel}(M, m, P)$ have the following intuitive meaning (in reverse order):

- P is the receiver (or recipient) of the message;
- m is the message we want to send to the receiver of the message;
- M is (or reduces to) a proper sub-object of the receiver of the message.

By looking at the last two rewrite rules, one may note that the Sel function "scans" the recipient of the message until it finds the definition of the method we want to use. When it finds the body of the method, it applies this body to the recipient of the message.

Example 2 (An object with "self-inflicted extension"). Consider the object `self_ext` defined as follows: $\texttt{self\_ext} \triangleq \langle\langle\, \rangle \leftarrow \texttt{add\_n} = \lambda\texttt{self}.\langle \texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.\texttt{1}\rangle\rangle$. If we send the message `add_n` to `self_ext`, then we get the following computation:

$$\begin{aligned} \texttt{self\_ext} \Leftarrow \texttt{add\_n} &\longrightarrow \mathit{Sel}(\texttt{self\_ext}, \texttt{add\_n}, \texttt{self\_ext}) \\ &\longrightarrow (\lambda\texttt{self}.\langle \texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.\texttt{1}\rangle)\, \texttt{self\_ext} \\ &\longrightarrow \langle \texttt{self\_ext} \leftarrow \texttt{n} = \lambda\texttt{s}.\texttt{1}\rangle, \end{aligned}$$

Terms.

$$
\begin{aligned}
M, N &::= \lambda x.M \mid MN \mid x \mid c && \text{(Lambda Calculus)}\\
U, V, E^a &::= M[s]^a \mid (UV)^a && \text{(Evaluation Contexts)}\\
s &::= U/x; s \mid id && \text{(Substitution)}
\end{aligned}
$$

where a ranges over an infinite set of *addresses*.

Rules.

$$
\begin{aligned}
\big((\lambda x.M)[s]^b\, U\big)^a &\to M[U/x;s]^a && && \text{(B)}\\
(MN)[s]^a &\to (M[s]^b\, N[s]^c)^a && b, c \text{ fresh} && \text{(App)}\\
x[E^b/x;s]^a &\to E^b && && \text{(FVarG)}\\
x[E^b/x;s]^a &\to E^a && && \text{(FVarE)}\\
x[U/y;s]^a &\to x[s]^a && x \not\equiv y && \text{(RVar)}
\end{aligned}
$$

Fig. 6. The Weak Lambda Calculus with Explicit Substitution and Addresses $\lambda\sigma_w^a$.

resulting in the method `n` being added to `self_ext`. On the other hand, if we send the message `add_n` twice to `self_ext` instead, the method `n` is only overridden with the same body; hence we obtain an object which is "operationally equivalent" to the previous one.

4.2 The Weak Lambda Calculus with Explicit Substitution and Addresses $\lambda\sigma_w^a$

We introduce the weak lambda calculus with explicit substitution and addresses $\lambda\sigma_w^a$ [4], where for the sake of simplicity, and in the style of Rose [18], we replace de Bruijn indexes with variable names. By "weak", we mean a lambda calculus in which reductions may not occur under abstractions, as standard in many programming languages. The syntax and the rules of this calculus are given in Figure 6. The explicit substitution gives an account of the concept of closure, while addresses give an account of sharing. Both are essential in efficient implementations of functional languages.

There are three levels of expressions. The first level is static. It gives the syntax of programs *code* (terms written $M, N, \ldots$), and it contains no address. The second and third levels are dynamic. They contain addresses and they are the level of *evaluation contexts*, and the level of *substitutions*. Evaluation contexts (terms written $U, V, \ldots$) model states of abstract machines. An evaluation context contains the temporary structure needed to compute the result of an operation. It denotes a term closed by a list of bindings also called substitution. There is an evaluation context associated with each construct of the language. Addresses (denoted by $a, b, \ldots$) label evaluation contexts. Intuitively, an address a models a reference to a unique term graph which is denoted as a standard term by simply unraveling it. The sharing information is kept through addresses, as superscripts of terms. This leads to two associated notions, namely *admissible terms* and *simultaneous rewriting*. An *admissible term* is a term in which there is not two different subterms at the same address. In the following, we only deal with admissible terms. A *simultaneous rewriting* (see also Subsection 5.1) means that, if a subterm U at address a is reduced to a term V, then all the subterms at the same address a are reduced in the same step to V. In other words, the simultaneous rewriting is a rewriting relation meant to preserve admissibility.

To be seen as a program *i.e.*, to enable a computation, a closed lambda term M must be given a substitution s (also called *environment*), initially the empty substitution *id*, and a location a to form an evaluation context called *addressed closure* $M[s]^a$. The environment s is the list of bindings of the variables free in M. To reduce terms, environments have to be distributed inside applications (App) until reaching a function or a variable. Hence, applications of weak lambda terms are also evaluation contexts. In this step of distributing the environment, "fresh" addresses are provided to evaluation contexts. A fresh address is an address unused in the global term. Intuitively, the address of an evaluation context is the address where the result of the computation will be stored. Since in a *closure* $M[s]^a$, the terms in s are also addressed terms, it follows that the duplication of s in (App) induces duplications of lists of pointers. Not only a duplication does not loose sharing, but it increases it.

When an abstraction is reached by a substitution, one gets a redex $((\lambda x.M)[s]^b\,U)^a$ (provided there is an argument U), hence one can apply the rule (B). This redex is reduced locally *i.e.*, U is not propagated to the occurrences of x, but the environment is just enlarged with the new pair U/x. Moreover, the result of the reduction is put at the same location as this of the redex in the left hand side, namely a. As a matter of fact, the result of the rewriting step is *shared* by all the steps that occur at address a.

When a variable x is reached, and the environment scanned by several steps of rule (RVar), x has eventually to be replaced by the evaluation context it refers to. $\lambda\sigma_w^a$ proposes two rules to do this, namely (FVarG), and (FVarE). The reason is that a choice has to be made on the address where to "store" the right hand side: it may be either the address of the evaluation context bound to x (FVarG), or the address of the main evaluation context (FVarE). In the first case, a redirection of the pointer which refers to the address a is performed toward the address b (where the term E lies), whereas in the latter case a copy of the part of the term E from address b to address a is made. In both cases, the result of the rewriting step is shared.

In the case of a copy, further sharing between the original node and the copied node will not be possible, but this has no influence on efficiency if the copied node denoted a value *i.e.*, a term of the form $(\lambda x.M)[s]^a$ or $c[s]^a$, because there may be no more further reductions on them.

A detailed discussion on this choice of rules can be found in [3, 4].

Example 3. The term $((V\,U)^a\,U)^b$ where $U \equiv ((\lambda x.x)[id]^c\;true[id]^d)^e$ and V is any evaluation context, may reduce in one step by rule (B) of Figure 6 to

$$((V\;x[true[id]^d/x;id]^e)^a\;x[true[id]^d/x;id]^e)^b,$$

but not to *e.g.*, $((V\ \underline{x[\mathit{true}[\mathit{id}]^d/x;\mathit{id}]^e})^a\ \underline{((\lambda x.x)[\mathit{id}]^c\ \mathit{true}[\mathit{id}]^d)^e})^b$ since the two distinct underlined subterms have a same address, namely e.

If we set V to $(\lambda y.\lambda z.y)[\mathit{id}]^f$, then the computation may proceed as follows:

$$
\begin{aligned}
&(((\lambda y.\lambda z.y)[\mathit{id}]^f\ x[\mathit{true}[\mathit{id}]^d/x;\mathit{id}]^e)^a\ x[\mathit{true}[\mathit{id}]^d/x;\mathit{id}]^e)^b \\
&\qquad\qquad \xrightarrow{*} y[x[\mathit{true}[\mathit{id}]^d/x;\mathit{id}]^e/z;x[\mathit{true}[\mathit{id}]^d/x;\mathit{id}]^e/y;\mathit{id}]^b && \text{(B+B)} \\
&\qquad\qquad \to y[x[\mathit{true}[\mathit{id}]^d/x;\mathit{id}]^e/y;\mathit{id}]^b && \text{(RVar)} \\
&\qquad\qquad \to x[\mathit{true}[\mathit{id}]^d/x;\mathit{id}]^e && \text{(FVarG)} \\
&\qquad\qquad \to \mathit{true}[\mathit{id}]^e, && \text{(FVarE)}
\end{aligned}
$$

where we chose to use both (FVarG) and (FVarE) for the sake of illustration.

All along this paper, we use the helpful intuition that an address corresponds to a location in a physical memory. However, we warn the reader that this intuition may be error prone. Access and allocation in a physical memory are expensive and often avoidable. Since a fresh address is given to every new evaluation context, the reader may think that we are not aware of this cost and have in mind an implementation which overuse locations. In fact, addresses capture more than locations. This has been shown in [12] where the states of an environment machine (with code, environment, stacks, and heap) are translated into $\lambda\sigma_w^a$. This translation showed that many addresses are *artificial i.e.*, do not have a physical reality in the heap, but correspond to components of states. It was also shown that the abstraction of sharing with addresses fits well with the environment machine, because it captures the strategy of the machine. The moral is that *everything which could have a physical location, in a particular implementation, has an address in the framework.*

5 The Syntax and the Operational Semantics of $\lambda\mathcal{O}bj^{+a}$

This section presents our framework. It is split into separated modules, namely L for the lambda calculus, C for the common operations on objects, F for the functional object part, and I for the imperative object part. All these modules can be combined, giving the whole $\lambda\mathcal{O}bj^{+a}$. The union of modules L, C, and F can be understood as the *the functional fragment of* $\lambda\mathcal{O}bj^{+a}$. As described in Figure 7, we find in $\lambda\mathcal{O}bj^{+a}$ the same levels as in $\lambda\sigma_w^a$, plus a dynamic level of *internal structures of objects* (or simply *object-structures*).

To the static level, we add some constructs: constructors of objects, method invocations, and explicit duplicators. There are operations to modify objects: the functional update, denoted by $\leftarrow$, and the imperative update, denoted by $\leftarrow:$. An informal semantics of these operators has been given in Section 3. As in [9], these operators can be understood as extension as well as override operators, since an extension is handled as a particular case of an override. One has also two imperative primitives for "copying" objects: $\mathsf{shallow}(x)$ is an operator which gives a new object-identity to the object pointed by x but still shares the same object-structure as the object x itself; $\mathsf{refresh}(x)$

Code

$$
\begin{array}{rll}
M, N ::= & \lambda x.M \mid MN \mid x \mid c & \text{(Lambda Calculus)} \\
\mid & M \Leftarrow m & \text{(Message Sending)} \\
\mid & \langle\,\rangle & \text{(Object Initialization)} \\
\mid & \langle M \leftarrow m = N\rangle \mid \langle M \leftarrow\!:\ m = N\rangle & \text{(Object Updates)} \\
\mid & \mathsf{shallow}(x) \mid \mathsf{refresh}(x) & \text{(Duplication Primitives)}
\end{array}
$$

where x ranges over variables, c ranges over constants and m ranges over methods.

Evaluation Contexts

$$
\begin{array}{rll}
U, V ::= & M[s]^a & \text{(Closure)} \\
\mid & (UV)^a & \text{(Application)} \\
\mid & (U \Leftarrow m)^a & \text{(Message Sending)} \\
\mid & \langle U \leftarrow m = V\rangle^a \mid \langle U \leftarrow\!:\ m = V\rangle^a & \text{(Object Updates)} \\
\mid & \lceil O\rceil^a \mid \bullet^a & \text{(Objects)} \\
\mid & Sel^a(O, m, U) & \text{(Lookup)}
\end{array}
$$

where a ranges over an infinite set of *addresses*.

Object-structures

$$
\begin{array}{rll}
O ::= & \langle\,\rangle^a \mid \langle O \leftarrow m = V\rangle^a \mid \bullet^a & \text{(Internal Objects)} \\
\mid & \mathsf{copy}(O)^a & \text{(Duplicator)}
\end{array}
$$

Environments

$$
s ::= U/x; s \mid id \qquad \text{(Substitution)}
$$

Fig. 7. The Syntax of λObj^{+a}.

is a kind of *dual* to $\mathsf{shallow}(x)$ as it makes a “hard copy” of the object-structure of x, and reassigns this structure to x. Therefore, the object-identity of x is not affected.

Similarly, some constructs are added as evaluation contexts. The evaluation context $\lceil O \rceil^a$ represents an object whose *internal* object-structure is O and whose object-identity is $\lceil\ \rceil^a$. In other words, the address a plays the rôle of an *entry point* of the object-structure O. An expression like $Sel^a(O, m, \lceil O \rceil^b)$ is an evaluation context (at address a). It looks up in the object-structure O of the receiver (represented by an evaluation context $\lceil O \rceil^b$), gets the method body and applies it to the receiver itself. The term $\bullet^a$ is a *back pointer* [18], its rôle is explained in Subsection 5.5 when we deal with the cyclic aspects of objects *i.e.*, the possibility to create “loops in the store”. Only $\bullet^a$ can occur inside a term having the same address a, therefore generalizing our informal notion of admissible term and simultaneous rewriting.

Internal objects O model the object-structures in memory. They are permanent structures which may only be accessed through the address of an object (denoted by a in $\lceil O \rceil^a$), and are never destroyed nor modified (but by the garbage collector, if there is one). Our calculus being inherently delegation-based, objects are implemented as linked lists (of fields/methods). Embedding-based inheritance can however be simulated thanks to the $\mathsf{refresh}(x)$ and $\mathsf{shallow}(x)$ operators. In particular, $\mathsf{refresh}(x)$ is defined in terms of an auxiliary operator called $\mathsf{copy}(O)$ which makes a copy of the object-structure. Again, because of imperative traits, object-structures can contain occurrences of $\bullet^a$.

5.1 The Simultaneous Rewriting

Simultaneous rewriting [18, 3] is a key concept in this paper and we would like to warn the reader not to take it as just a slight variant of the usual term rewriting. Actually, due mostly to imperative features introduced in module I, simultaneous rewriting goes much beyond the classical *match and replace* paradigm of the traditional first order rewriting and must be defined extremely carefully in order to preserve:

Horizontal Admissibility, *i.e.*, all the subterms at the same address should be equal and rewritten together, as shown in Example 3.

Vertical Admissibility, *i.e.*, a term can contain its own address a as the address of one of its proper subterms, only if this subterm is a $\bullet^a$. This ensures that back-pointers for terms at address a are only denoted by the term $\bullet^a$.

Roughly speaking, in order to maintain these requirements the definition proceeds as follows to rewrite a term U into V.

1. Match a subterm of U at address say a with a left hand side of a rule, compute the corresponding right hand side and create the new fresh addresses (if required), then replace all the subterms of U at address a with the obtained right hand side.
2. Replace some subterms by back-pointers (a *fold* operation), or some back-pointers by particular terms (an *unfold* operation), following some specific techniques (see [13]), so that the result is a vertically admissible term.

Instantiation

$$\langle\,\rangle[s]^a \to \lceil\langle\,\rangle^b\rceil^a \qquad b \text{ fresh} \qquad \text{(OI)}$$

Message Sending

$$(M \Leftarrow m)[s]^a \to (M[s]^b \Leftarrow m)^a \qquad b \text{ fresh} \qquad \text{(CP)}$$

$$\left(\lceil O\rceil^b \Leftarrow m\right)^a \to Sel^a\left(O, m, \lceil O\rceil^b\right) \qquad \text{(SE)}$$

$$Sel^a\left(\langle O \leftarrow m = V\rangle^b, m, U\right) \to (V\,U)^a \qquad \text{(SU)}$$

$$Sel^a\left(\langle O \leftarrow n = V\rangle^b, m, U\right) \to Sel^a\left(O, m, U\right) \qquad \text{(NE)}$$

Fig. 8. The Common Object Module C.

5.2 The Module L

The module L is the calculus $\lambda\sigma_w^a$, and needs no comments.

5.3 The Common Object Module C

The Common Object module is shown in Figure 8. It handles object instantiation and message sending. *Object instantiation* is characterized by the rule (OI) where an empty object is given an object-identity. More sophisticated objects may then be obtained by functional or imperative update. *Message sending* is formalized by the four remaining rules. The rule (CP) which propagates a given substitution into the receiver of the message; apply this rule means to "install" the evaluation context needed to actually proceed. The meaning of the remaining rules is quite intuitive: (SE) performs message sending, while (SU), and (NE) perform the method-lookup. We can observe here the similarity with the operational semantics of $\lambda\mathcal{O}bj^+$.

Functional Update

$$\langle M \leftarrow m = N\rangle[s]^a \to \langle M[s]^b \leftarrow m = N[s]^c\rangle^a \qquad b, c \text{ fresh} \qquad \text{(FP)}$$

$$\langle\lceil O\rceil^b \leftarrow m = V\rangle^a \to \lceil\langle O \leftarrow m = V\rangle^c\rceil^a \qquad c \text{ fresh} \qquad \text{(FC)}$$

Fig. 9. The Functional Object Module F.

5.4 The Functional Object Module F

The Functional Object module gives the operational semantics of a calculus of non mutable objects. It contains only two rules (Figure 9). Rule (FP) "pre-computes" the functional update, installing the evaluation context needed to actually proceed. Rule (FC) describes the actual update of an object of identity b. The update is not made in place and no mutation is performed, but the result is a new object (with a different object-identity). This is why we call this operator "functional" or "non mutating".

Imperative Update

$\langle M \leftarrow\colon m = N\rangle[s]^a \to \langle M[s]^b \leftarrow\colon m = N[s]^c\rangle^a$	b, c fresh	(IP)
$\langle \lceil O\rceil^b \leftarrow\colon m = V\rangle^a \to \lceil \langle O \leftarrow m = V\rangle^c\rceil^b$	c fresh	(IC)

Cloning Primitives

$\mathsf{shallow}(x)[U/y; s]^a \to \mathsf{shallow}(x)[s]^a$	$x \not\equiv y$	(VS)
$\mathsf{shallow}(x)[\lceil O\rceil^b/x; s]^a \to \lceil O\rceil^a$		(SC)
$\mathsf{refresh}(x)[U/y; s]^a \to \mathsf{refresh}(x)[s]^a$	$x \not\equiv y$	(RS)
$\mathsf{refresh}(x)[\lceil O\rceil^b/x; s]^a \to \lceil \mathsf{copy}(O)^c\rceil^b$	c fresh	(RE)
$\mathsf{copy}(\langle\,\rangle^b)^a \to \langle\,\rangle^a$		(CE)
$\mathsf{copy}(\langle O \leftarrow m = V\rangle^b)^a \to \langle \mathsf{copy}(O)^c \leftarrow m = V\rangle^a$	c fresh	(CO)

Fig. 10. The Imperative Object Module I.

5.5 The Imperative Object Module I

The Imperative Object module (Figure 10) contains rules for the mutation of objects (imperative update) and cloning primitives. Imperative update is formalized in a way close to the functional update. Rules (IP) and (IC) are much like (FP) and (FC); they differ in address management and they are self-explaining. Indeed let us look at the address b in rule (IC). In the left hand side, b is the identity of an object $\lceil O\rceil$, when in the right hand side it is the identity of the whole object modified by the rule. Since b may be shared from anywhere in the context of evaluation, this modification is observable non locally, hence a mutation is performed.

It is worth to note that the rule (IC) may create cycles and therefore back pointers. Intuitively, when we deal with imperative traits, we can create non admissible terms because of cyclic references. Every reference to $\lceil O\rceil^b$ in V must be replaced by $\bullet^b$ to avoid $\lceil\langle O \leftarrow m = V\rangle\rceil^b$ to contain itself.

The primitives for cloning are $\mathsf{shallow}(x)$ and $\mathsf{refresh}(x)$.

- A $\mathsf{shallow}(x)$ creates an object-identity for an object, but x and $\mathsf{shallow}(x)$ share the same object-structure. The rule (SC) can be seen as the imperative counterpart of the rule (FVarE) of module L in case $E^b \equiv \lceil O \rceil^b$, for a given b.
- A $\mathsf{refresh}(x)$ creates for x a new object-structure isomorphic to the previous one. A $\mathsf{refresh}(x)$ calls, through the rule (RE), an auxiliary operator named copy. A $\mathsf{copy}(O)$ recursively performs a copy of the linked list, via the rules (CE), and (CO).

An intuitive representation of the behaviour of those operators is given in Figure 11.

6 Understanding $\lambda\mathcal{O}bj^{+a}$

6.1 Examples of Terms

Example 4. The term $\langle \lceil \langle\,\rangle^a \rceil^b \leftarrow: \mathtt{m} = (\lambda\mathtt{self.x})[\lceil \langle\,\rangle^a \rceil^b/\mathtt{x}; id]^c\rangle^d$ does not reduce to

$$\lceil \langle \langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda\mathtt{self.x})[\lceil \langle\,\rangle^a \rceil^b/\mathtt{x}; id]^c\rangle^d \rceil^b$$

(which is a non admissible term) but instead to

$$\langle \lceil \langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda\mathtt{self.x})[\bullet^b/\mathtt{x}; id]^c\rangle^d \rceil^b.$$

It is crucial to note that the sense of the two terms is essentially different, since the latter expresses a loop in the store whereas the former does not mean anything consistent, since two semantically distinct subterms have the same address b.

Example 5. The term $\langle \lceil \langle \langle\,\rangle^a \leftarrow \mathtt{m} = M[\bullet^d/\mathtt{x}; id]^b\rangle^c \rceil^d \leftarrow \mathtt{n} = N[id]^e\rangle^f$ does not reduce to

$$\lceil \langle \langle \langle\,\rangle^a \leftarrow \mathtt{m} = M[\bullet^d/\mathtt{x}; id]^b\rangle^c \leftarrow \mathtt{n} = N[id]^e\rangle^g \rceil^f$$

(which is not admissible) but instead to

$$\lceil \langle \langle \langle\,\rangle^a \leftarrow \mathtt{m} = M[\lceil \bullet^c \rceil^d/\mathtt{x}; id]^b\rangle^c \leftarrow \mathtt{n} = N[id]^e\rangle^g \rceil^f.$$

In this last term, the back pointer $\bullet^d$ has been *unfolded* following the definition of simultaneous rewriting *i.e.*, replaced by the term it refers to, namely $\lceil \bullet^c \rceil^d$ (c is still in the context of the subterm, and therefore $\bullet^c$ is not unfolded). This unfolding is due to the removal of the surrounding address d, which otherwise could lead to a loss of information on the shape of the term associated to the address d.

6.2 Examples of Derivations

Example 6. Let $\mathtt{self\_ext}$ be the term defined in Example 2, and N denote the subterm $\lambda\mathtt{self}.\langle\mathtt{self} \leftarrow \mathtt{n} = \lambda\mathtt{s}.1\rangle$.

$$\begin{aligned}
(\mathtt{self_ext} \Leftarrow \mathtt{add_n})[id]^a &\xrightarrow{*} (\langle\langle\,\rangle[id]^d \leftarrow \mathtt{add_n} = N[id]^c\rangle^b \Leftarrow \mathtt{add_n})^a && (1)\\
&\to (\langle \lceil \langle\,\rangle^e \rceil^d \leftarrow \mathtt{add_n} = N[id]^c\rangle^b \Leftarrow \mathtt{add_n})^a && (2)\\
&\to (\lceil \underbrace{\langle\langle\,\rangle^e \leftarrow \mathtt{add_n} = N[id]^c\rangle^f}_{O} \rceil^b \Leftarrow \mathtt{add_n})^a && (3)
\end{aligned}$$

$$\to Sel^a(O, \mathtt{add\_n}, \lceil O \rceil^b) \quad (4)$$
$$\to ((\lambda\mathtt{self}.\langle \mathtt{self} \leftarrow \mathtt{n} = \lambda\mathtt{s}.1\rangle)[id]^c \lceil O \rceil^b)^a \quad (5)$$
$$\to \langle \mathtt{self} \leftarrow \mathtt{n} = \lambda\mathtt{s}.1\rangle[\lceil O \rceil^b/\mathtt{self}; id]^a \quad (6)$$
$$\xrightarrow{*} \langle \lceil O \rceil^b \leftarrow \mathtt{n} = \lambda\mathtt{s}.1[\lceil O \rceil^b/\mathtt{self}; id]^g\rangle^a \quad (7)$$
$$\to \lceil \langle O \leftarrow \mathtt{n} = \lambda\mathtt{s}.1[\lceil O \rceil^b/\mathtt{self}; id]^g\rangle^h \rceil^a \quad (8)$$

In (1,2), two steps are performed to distribute the environment inside the expression by rules (CP) and (FP), then the empty object is given an object-structure and an object identity (OI). In (3), this new object is functionally extended (FC), hence it shares the structure of the former object but has a different object-identity. In (4,5), two steps are performed to look-up the method add_n (rules (NE) and (SU)). Step (6) is an application of (B). In (7), the environment is distributed inside the functional extension (FP), and then self is replaced by the object it refers (FVarG). Step (8) is simply an application of rule (FC) *i.e.*, the proceeding of a functional extension. The final term contains some sharing, as the object-structure denoted by O and rooted at address b occurs twice.

Example 7. We give a similar example, where a functional update is replaced by an imperative one. Let $\mathtt{self\_ext}'$ denote the term $\langle\langle\,\rangle \leftarrow \mathtt{add\_n} = N'\rangle$, where N' is $\lambda\mathtt{self}.\langle \mathtt{self} \leftarrow\!:\ \mathtt{n} = \lambda\mathtt{s}.1\rangle$.

$$(\mathtt{self\_ext}' \Leftarrow \mathtt{add\_n})[id]^a \xrightarrow{*} (\lceil \underbrace{\langle\langle\,\rangle^e \leftarrow \mathtt{add\_n} = N'[id]^c\rangle^f}_{O'} \rceil^b \Leftarrow \mathtt{add\_n})^a \quad (1)$$
$$\xrightarrow{*} \langle \lceil O' \rceil^b \leftarrow\!:\ \mathtt{n} = \lambda\mathtt{s}.1[\lceil O' \rceil^b/\mathtt{self}; id]^g\rangle^a \quad (2)$$
$$\to \lceil \langle O' \leftarrow \mathtt{n} = \lambda\mathtt{s}.1[\bullet^b/\mathtt{self}; id]^g\rangle^h \rceil^b \quad (3)$$

The first steps (1,2) are similar to the first steps (1 to 7) of the previous example. In (3), the imperative extension is performed (IC), and a subterm replaced by $\bullet^b$ to denote the loop to the root, since the object in the environment has the same identity as the object the environment belongs to.

6.3 Functional *vs.* Imperative (Non Mutable *vs.* Mutable)

The functional module F can be simulated by the imperative one I. This can be simply done by combining the $\mathsf{shallow}(x)$ operation with an imperative update. Indeed, a functional object obtained by inheriting the properties of a prototype can be encoded by a $\mathsf{shallow}$ followed by an imperative method update. This proves the fact that $\mathsf{F} \subseteq \mathsf{I}$. The encoding of $\langle M \leftarrow m = N\rangle$ is $(\lambda x.\langle \mathsf{shallow}(x) \leftarrow\!:\ m = N\rangle)\, M$.

6.4 Cloning

It is possible, using the Imperative Object module, to define a clone operation. The clone used in Figures 2 and 4, whose intuitive semantics is illustrated in Figure 11, is defined as follows: $\mathsf{clone}(x) \triangleq (\mathsf{refresh} \circ \mathsf{shallow})(x) \triangleq (\lambda y.\mathsf{refresh}(y))\, \mathsf{shallow}(x)$. The clone used in Figures 1 and 3, instead, is defined as follows: $\mathsf{clone}(x) \triangleq \mathsf{shallow}(x)$.

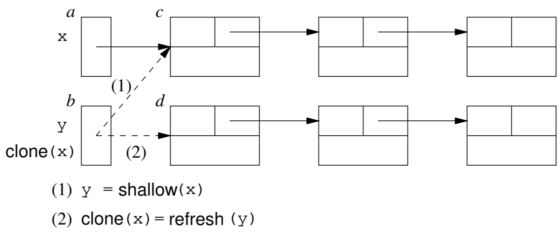

Fig. 11. The embedding-based $\mathsf{clone}(x)$ Operator.

Since $\lambda\mathcal{O}bj^{+}$ is inherently delegation-based, it follows that an embedding-based technique of inheritance can be encoded using the Imperative Object module I. Other interesting operators can be defined by combining the different features of $\lambda\mathcal{O}bj^{+a}$.

7 Related Work

- The framework $\lambda\mathcal{O}bj^{+a}$ generalizes a novel technique to implement programming languages: we call this technique *address-based*. Addresses are attached to every entities of our framework. The graph-based implementation technique à la Wadsworth, as well as others, can be subsumed within our framework. A type system can be defined quite easily by adding in the "context soup" also the type of addresses. As such, a type soundness result can be proved relatively simply, if we compare it with the traditional approaches of stack (a function from variables to results) and store (a function from store locations to "closures"). It is worth to note that the choice of the target calculus (an object based one) is not important; the address-based technique can be used, in principle, to implement other calculi, but it fits well to object-calculi.
- The framework $\lambda\mathcal{O}bj^{+a}$ performs an *imperative object extension*: an imperative (mutable) object is a "functional (non-mutable) internal object-structure" pointed by an object-identity. To extend an imperative object means to functionally extend its object-structure while keeping its object-identity. Note that the same mechanism is used to implement method override.

Among the many imperative object calculi presented in the literature, the closest are the one described in [1] (Chapter 10-11) and [5]. The first is the ς_{imp} calculus of Abadi and Cardelli, while the second is the first imperative version of $\lambda\mathcal{O}bj$ of Fisher Honsell and Mitchell. Both calculi use a stack and store technique to present the semantics of the calculus and to prove type soundness. Both calculi have an imperative override that (in contrast to our approach) *substitutes* the old body of a field/method with the new one. Both calculi adopt a *call-by-value* strategy. In addition, the calculus presented in [5] have a *functional object extension*. The divergence from those calculi is shown in the following table, where s&s stands for stack and store, addr. for address-based, and c.b.v. for call-by-value:

	model	override	extension	self-infliction	strategies
[1]	s&s	imperative	no	no	c.b.v.
[5]	s&s	imperative	functional	no	c.b.v.
$\lambda\mathcal{O}bj^{+a}$	addr.	funct./imp.	funct./imp.	yes	many

Less specifically, a work is in progress [13] to formalize in a general setting all the notions of sharing, cycles, and mutation, mentioned in this paper.

8 Conclusions

We have defined $\lambda\mathcal{O}bj^{+a}$, a framework for object calculi which is intended to give a firm foundation for the operational semantics of object oriented languages. Future works will focus on specific calculi as combination of modules and strategies *e.g.*, the functional fragment with embedding and call-by-need, or the imperative fragment with delegation and call-by-value. It should also be interesting to study specific aspects like typing, strategies (see [14]) and distribution of objects across networks. Other useful extensions of this framework should be studied, such as providing an imperative override of fields (in the style of [1, 5]) *i.e.*, a *field look up and replacement*. To this aim, a distinction has to be made between fields (and may be more generally procedures or functions that do not have a self-reference) and methods. The formalism used to describe $\lambda\mathcal{O}bj^{+a}$ provides the suited tools for such an extension.

Acknowledgement. The authors are grateful to Zine-El-Abidine Benaissa, Furio Honsell, and Kristoffer Høgsbro Rose for their useful comments on this work.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
3. Z.-E.-A. Benaissa. *Les calculs de substitutions explicites comme fondement de l'implantation des langages fonctionnels*. PhD thesis, Université Henri Poincaré Nancy 1, 1997. In french.
4. Z.-E.-A. Benaissa, K.H. Rose, and P. Lescanne. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Proc. of PLILP*, number 1140 in Lecture Notes in Computer Science, pages 393–407. Springer-Verlag, 1996.
5. V. Bono and K. Fisher. An imperative first-order calculus with object extension. In *Proc. of ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 462–497. Springer-Verlag, 1998.
6. L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.
7. C. Chambers. The Cecil language specification, and rationale. Technical Report 93-03-05, University of Washington, Department of Computer Science and Engineering, 1993.
8. P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.
9. P. Di Gianantonio, F. Honsell, and L. Liquori. A lambda calculus of objects with self-inflicted extension. In *Proc. of OOPSLA*, pages 166–178. The ACM Press, 1998.

10. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102, 1992.
11. 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.
12. F. Lang. *Modèles de la β-réduction pour les implantations*. PhD thesis, École Normale Supérieure de Lyon, 1998. In french.
13. F. Lang, D. Dougherty, P. Lescanne, and K. H. Rose. Addressed term rewriting systems. Research Report RR 1999-30, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 1999.
14. F. Lang, P. Lescanne, and L. Liquori. A framework for defining object calculi. Research Report RR 1998-51, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 1998.
15. P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In *Proc. of POPL*, pages 60–69, 1994.
16. T. Lindholm and F. Yellin. *The Java Virtual Machine specification*. Addison-Wesley Publishing Company, 1996.
17. J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton Jones, A. Reid, and P. Wadler. *Haskell 1.4, a non strict purely functional language*, 1997.
18. K. H. Rose. *Operational reduction models for functional programming languages*. PhD thesis, DIKU, København, 1996.
19. A. Tailvalsaari. Kevo, a prototype-based object-oriented language based on concatenation and modules operations. Technical Report LACIR 92-02, University of Victoria, 1992.
20. M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.
21. D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.
22. D. Ungar and B. Smith, R. Self: the power of simplicity. In *Proc. of OOPSLA*, pages 227–241. The ACM Press, 1987.
23. C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus*. PhD thesis, Oxford, 1971.
24. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.