Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON
n° 8512

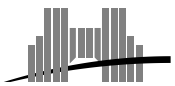
A Framework for Defining Object-Calculi

Extended Abstract

Frédéric Lang
Pierre Lescanne
Luigi Liquori

Décembre 1998

Research Report N° RR 1998-51



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

A Framework for Defining Object-Calculi

Extended Abstract

Frédéric Lang
Pierre Lescanne
Luigi Liquori

Décembre 1998

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 object-based calculi according to their main implementation techniques of inheritance, namely *delegation* and *embedding*. This distinction comes easily from the choice of the rules we make.
Our framework is founded on two previous works, namely the *Lambda Calculus of Objects* of Fischer, Honsell, and Mitchell for the object aspects and the $\lambda\sigma_w^a$ of Benaissa, Lang, 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 substitutions extended with addresses to speak about memory management.
The framework is presented as a set of *modules*, each of which captures a peculiar aspect of object-calculi (functional *vs.* imperative, delegation *vs.* embedding, and any combination of them).
Our framework satisfies some crucial properties, namely *confluence* on the functional fragment (the final result does not depend on the sequence of computations *i.e.*, on the evaluation strategy), *operational soundness* (our calculi yield the same results on the same data as in the Lambda Calculus of Objects), *subject reduction* (programs preserve types), *type soundness* (a typed program can not go wrong as invoking an unknown method on an object).

Keywords: Functional and imperative object-based languages, operational semantics, implementation issues, memory management, type system, delegation *vs.* embedding.

Résumé

Dans cet article, nous définissons un système général pour la fondation d'une sémantique opérationnelle (à pas réduit) des langages basés sur les objets, en insistant sur des problèmes propres aux langages fonctionnels ou impératifs. Dans ce cadre, nous pouvons classifier très naturellement les calculs basés sur les objets selon leur principale technique d'implantation de l'héritage, notamment *délégation* et *emboîtement*. Cette distinction est faite simplement dans le choix des règles.

Notre système se fonde essentiellement sur deux travaux précédents, le Lambda Calcul des Objets de Fisher, Honsell, et Mitchell pour les aspects objets, et le calcul $\lambda\sigma_w^a$ de Benaissa, Lang, Lescanne, et Rose pour la description de la sémantique opérationnelle et du partage. Le premier est la formalisation d'un petit langage basé sur la délégation qui contient à la fois le lambda calcul et des primitives sur les objets permettant de les créer, de les modifier, et de leur envoyer des messages, tandis que le second est conccu pour donner une description générique des langages fonctionnels et est basé sur un calcul de substitution explicite étendu avec des adresses pour modéliser la gestion de la mémoire.

Le système est présenté sous la forme d'un ensemble de *modules*, chacun décrivant un aspect particulier des calculs à objets (fonctionnel ou impératif, délégation ou emboîtement, et toute combinaison de ceux-ci).

Notre système satisfait des propriétés cruciales comme la *confluence* sur le fragment fonctionnel (le résultat final ne dépend pas de la séquence des calculs, c'est-à-dire de la stratégie d'évaluation), la *correction opérationnelle* (nos calculs retournent le même résultat pour les mêmes données que le Lambda Calcul des Objets), l'*auto réduction* (les programmes préservent les types), et la *sûreté du typage* (un programme typé ne peut pas mener à une erreur d'exécution comme invoquer une méthode inconnue sur un objet).

Mots-clés: Langages basés sur les objets, fonctionnels et impératifs, sémantique opérationnelle, problèmes d'implantation, gestion de la mémoire, système de types, délégation ou emboîtement.

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 [LY96], but unfortunately its informal character does not fulfil 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 delegation languages and embedding languages; this comes naturally from the choice of the rules. 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 peculiar aspect of object-calculi (functional *vs.* imperative, delegation *vs.* embedding, and any combination of them). Genericity comes also from a total *independence from the strategy*. Indeed the modules are sets of rules which describe small steps in the transformations of the objects, when the strategies describe how these rules are invoked giving the general evolution of the whole program. Usually in the description of an operational semantics, strategies and small steps are tightly coupled whereas in our approach they are disconnected. $\lambda\mathcal{O}bj^{+a}$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. *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 objects-based languages the concepts of *addresses* and *simultaneous rewriting*, which differ from the classical *match and replace* technique of rewriting and which are intended to capture the imperative features of object-based languages.

$\lambda\mathcal{O}bj^{+a}$ has also a strong mathematical content which allows us to prove formally properties as theorems, like *confluence on the functional fragment* (the final result does not depend on the sequence of computations *i.e.*, on the evaluation strategy), *operational soundness* (our calculi yield the same results on the same data as in the Lambda Calculus of Objects), *subject reduction* (programs preserve types), *type soundness* (a typed program can not go wrong as invoking an unknown method on an object).

$\lambda\mathcal{O}bj^{+a}$ is founded on two previous works, namely the Lambda Calculus of objects of Fisher, Honsell, and Mitchell [FHM94, GHL98] for the object aspects, and $\lambda\sigma_w^a$ of Benaissa, Lang, Lescanne and Rose [BLLR99]. 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 a calculus of explicit substitutions which gives a generic description of functional language implementations. $\lambda\sigma_w^a$ itself was founded on the notion of address defined by Rose in his thesis [Ros96], and derived from the

weak lambda calculus of explicit substitution $\lambda\sigma_w$ [CHL96].

The paper is structured as follows. Section 2 quickly presents the main lines of the formal approaches to object-based languages, when Section 3 addresses mostly the implementation aspects. Section 4 introduces the Lambda Calculus of Objects, and λx_w which is a variant of $\lambda\sigma_w$. Section 5 is the real core of the paper as it presents $\lambda\mathcal{O}bj^{+a}$ through its four modules L, C, F and I. Section 6 gives some examples motivating our framework, Section 7 details the notion of simultaneous rewriting, and Section 8 talks about strategies. Section 9 is a presentation of a type system for $\lambda\mathcal{O}bj^{+a}$. Finally, Section 10 gives some formal properties of $\lambda\mathcal{O}bj^{+a}$.

2 Object-based Calculi

The last few years have addressed the study of object-oriented languages and their type systems [AC96, FHM94]. The main goal of this research was to build *safe and flexible* type systems that analyze the program text before execution.

In addition (and not in contrast) with the traditional class-based view, where *classes* are seen as the primitive notion to build object instances, the last few 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 [US87], Obliq [Car95], Kevo [Tai92], Cecil [Cha93], O-{1,2,3} [AC96]) or simply 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 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 "state". However, those papers are not a simple exercise of style, since, as well stated in [AC96, BF98] it may happen that a type system designed for a functional calculus can be "well fitted" for an imperative one.

Among the theoretical proposals for defining an object-based language, two of them have spurred an intense research.

The Object Calculus of Abadi and Cardelli is a calculus of typed objects. The objects have *fixed size* in order to give account to a standard notion of subtyping. The operations allowed on objects are method invocations and method updates. The calculus is computationally complete since the lambda calculus can be encoded via suitable objects. The calculus has both functional and imperative version, the latter being obtained by simply modifying the dynamic semantics of the former. Many type systems are proposed for this calculus. Some of them take into account the so called *mytype specialization* of the inherited methods. Issues of protection are also elegantly solved via "variance annotations" inside

object-types. Finally, classes can be implemented using the well-known *record-of-premethods* approach (*i.e.*, a class is an object which has a `new` method that "installs" the premethods of the class).

The Lambda Calculus of Objects of Fisher, Honsell, and Mitchell is an untyped functional lambda calculus enriched with object primitives. In this calculus, objects are untyped and a new object can be created by modifying and/or extending an existing object (called a *prototype*). 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. Bono and Fisher in [BF98] have designed an imperative version of $\lambda\mathcal{O}bj$ featuring an encapsulation mechanism obtained via abstract data types. Classes can also be implemented in $\lambda\mathcal{O}bj$: in a simplified view, a class `A` has a `new` method that first creates an instance `b` of the superclass `B` of `A` and then adds (or updates) this instance with all the methods declared in `A`.

3 Implementation of Object-based Calculi

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.*, fully functional and imperative ones) there are two different techniques of implementation of inheritance, namely the *embedding-based* and the *delegation-based* ones, studied in this section. Since we are interested in modeling operational semantics for extensible objects, we have chosen the Lambda Calculus of Objects as a target calculus; likewise we could have chosen the Extended Object Calculus of [Liq97], without loosing the full aims of the paper.

The following schematic example will be useful to understand how inheritance can be implemented using the embedding-based and the delegation-based techniques. Again for the sake of simplicity we will not raise issues like privacy or encapsulation (considering, in fact, that methods and fields belong to the same abstraction level).

Example 1 (A small program using a "schematic" prototype). 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
{p.set(a,b,c):={(((self.x:=self.x*a).y:=self.y*b).onoff:=c)};
 p.switch():+{self.onoff:=not(self.onoff)}}
```

where `:=` denotes an override and `:+` denotes an extension.

In the following we discuss the two models of implementation of inheritance and we highlight the differences between an imperative versus a functional model of object-calculi. Before we start, we explain (rather informally) the semantics of the `clone` operator.

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 that show possible implementations of Example 1.

3.2 Functional Object-calculi

As known, functional calculi lack a notion of *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 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 can either override (indicated in the example as `:=`) or extend (indicated as `:+`) 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). 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. By looking at Figure 1, one sees how Example 1 can be implemented using a delegation-based technique.

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 (for sake of simplicity we omitted in the picture the intermediate identities of the sub-objects of `pixel`, and `p`).

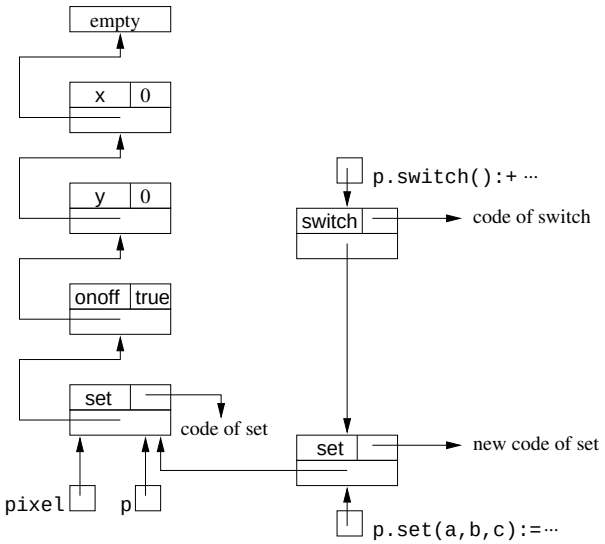


Figure1. Functional Delegation-based Inheritance

3.3 Imperative Object-calculi

Imperative object-calculi have been shown to be fundamental in describing implementation class-based languages like Smalltalk and Java. They are also essential as foundation of 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 when one deals with object extension, because the extension must maintain the object-identity of the object being extended.

Delegation-based Inheritance The semantics of the update operation is subtle because of side effects. Figure 3 shows the implementation of Example 1. Observe how the override of the `set` method and the addition of the `switch` method change the object structure of `p` without changing its object-identity.

Embedding-based Inheritance As above, (see Figure 4), an update modifies the object-structure while keeping its object-identity.

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

In this section we give a gentle introduction to calculi that have inspired our framework.

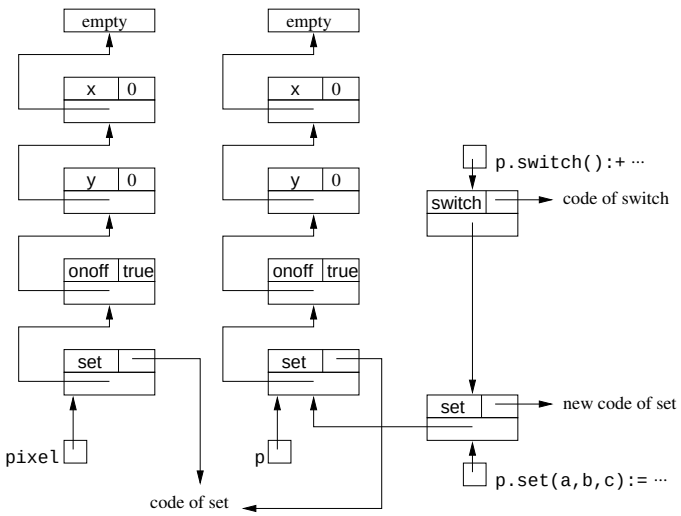


Figure2. Functional Embedding-based Inheritance

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

The calculus $\lambda\mathcal{O}bj^{+}$ [GHL98] 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 *meta substitutions* (denoted by $\{M/x\}$), 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 $\lambda\mathcal{O}bj$ [FHM94] 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 $Sel(M, m, M)$, where the arguments of Sel have the following intuitive meaning (in reverse order):

- (3^{rd}-arg) is the receiver (or recipient) of the message;
- (2^{nd}-arg) is the message we want to send to the receiver of the message;
- (1^{st}-arg) 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.

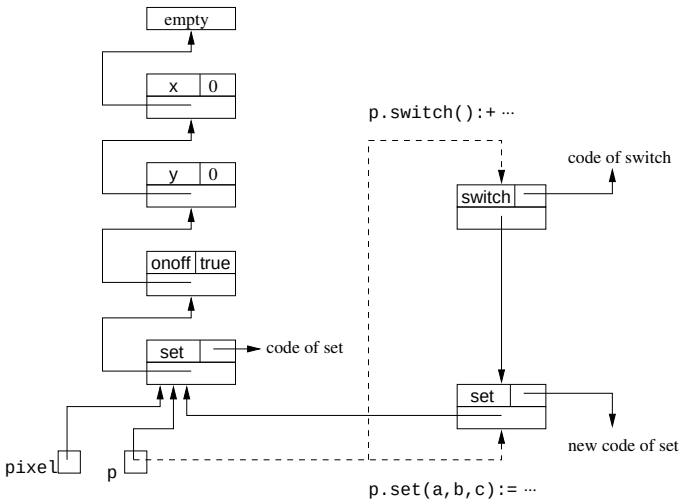


Figure3. Imperative Delegation-based Inheritance

Example 2 (An object with "self-inflicted extension"). Consider the object `self_ext` defined as follows:

$$\texttt{self_ext} \triangleq \langle\langle\rangle \leftarrow \texttt{add_n} = \lambda\texttt{self}.\langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle\rangle$$

If we send the message `add_n` to `self_ext`, then we get the following computation:

$$\begin{aligned}\texttt{self_ext} \Leftarrow \texttt{add_n} &\rightarrow Sel(\texttt{self_ext}, \texttt{add_n}, \texttt{self_ext}) \\ &\rightarrow (\lambda\texttt{self}.\langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle)\ \texttt{self_ext} \\ &\rightarrow \langle\texttt{self_ext} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle\end{aligned}$$

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 of Explicit Substitution λx_w

We introduce a calculus of explicit substitution called λx_w, which is the starting point of our calculi of objects. It is a weak lambda calculus[1] extended with environments. By "weak", we mean a lambda calculus in which reductions may

[1] For sake of readability, in this paper we present a lambda calculus with explicit names, unlike the original work [CHL96] which defines $\lambda\sigma_w$, a lambda calculus with de Bruijn index.

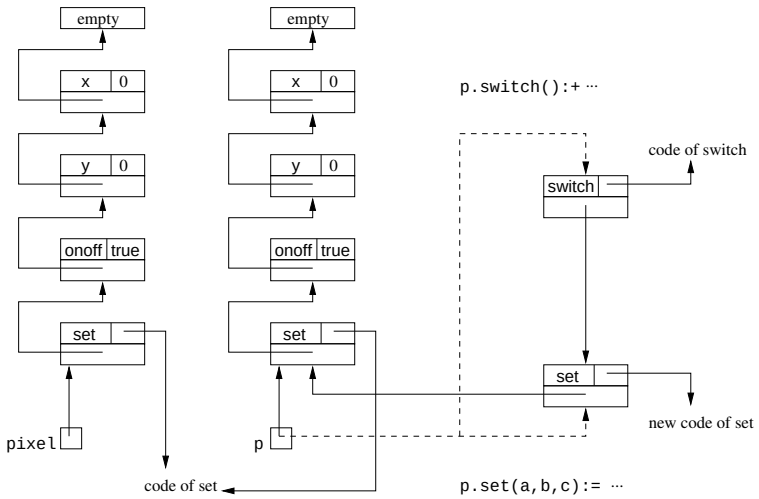


Figure4. Imperative Embedding-based Inheritance

not occur under abstractions. This restriction is standard in all programming languages. The syntax and rules of this calculus are given on Figure 6. A lambda term is either an abstraction, an application, a variable, or a constant.

To be seen as a program *i.e.*, to enable a computation, a lambda term M must be associated with an environment s (also known as an *explicit substitution*) to form what we call a *closure* $M[s]$, where s is the list of bindings of the variables free in M. Therefore $M[s]$ is closed, hence its name of closure. A closure is called weak lambda term. To reduce terms, environments have to be distributed in applications (App) until reaching a function or a variable. Hence, weak lambda terms (denoted by W) may also be applications of weak lambda terms.

When a function is reached, one has a redex $(\lambda x.M)[s]\,W$ and we can apply the rule (B). This redex is reduced locally *i.e.*, W is not propagated to the occurrences of x, but the environment is just enlarged with the new pair W/x. When a variable is reached (Var), it is simply replaced by the weak lambda term it refers to. For readability, we write a substitution containing W as first value associated to x while reading the substitution from left to right as $\ldots W/x \ldots$. In other words, if there exists another binding W'/x of the same variable name in the same substitution, then it is on the right of W/x. The result of a reduction is a *weak normal form i.e.*, a term of the form $(\lambda x.M)[s]$ or $c[s]$.

Example 3. The evaluation of $(\lambda x.\lambda y.x)\,\mathsf{true}$ in $\lambda \mathrm{x}_w$ is

$$((\lambda x.\lambda y.x)\,\mathsf{true})[\mathrm{id}] \xrightarrow[(\mathrm{App})]{} (\lambda x.\lambda y.x)[\mathrm{id}]\,\mathsf{true}[\mathrm{id}] \xrightarrow[(\mathrm{B})]{} (\lambda y.x)[\mathsf{true}[\mathrm{id}]/x; \mathrm{id}].$$

An extension of $\lambda \mathrm{x}_w$ with addresses to handle sharing in implementations (similar to $\lambda\sigma_w^a$) is part of our framework as module L, presented in Subsection 5.1.

Syntax.

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

Operational Semantics.

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

Figure5. The Lambda Calculus of Objects with Self-inflicted Extension $\lambda\mathcal{O}bj^{+}$

5 The Framework $\lambda\mathcal{O}bj^{+a}$

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 $\lambda\mathcal{O}bj^{+a}$, and the union of modules L, C and F (denoted by $\mathsf{L}+\mathsf{C}+\mathsf{F}$) is called *the functional fragment of* $\lambda\mathcal{O}bj^{+a}$.

We use three levels of expressions. The first level is the static level, *i.e.*, the *code* of programs. The second and third levels are dynamic levels: the level of *evaluation contexts*, and the level of *internal structure of objects* (or simply *object-structure*). These three levels are described on Figure 7.

The static level (terms written M and N) gives all the constructs the programmer may need to write her (his) program: pure lambda terms, 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 [GHL98], these operators can be understood as extension as well as override operators, since an override is handled as a particular case of extension. One has also two 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; $\mathsf{refresh}(x)$ is a kind of *dual* to $\mathsf{shallow}(x)$ in the sense that it makes an "hard copy" of the object-structure of x, and reassigns this structure to x. Therefore, the object-identity of x is not affected.

Evaluation contexts (terms written U and V) model states of abstract machines. An evaluation context contains the temporary structure needed to com-

Terms.

$$M, N ::= \lambda x.M \mid MN \mid x \mid c \qquad \text{(Lambda Calculus)}$$
$$W ::= M[s] \mid WW \qquad \text{(Weak Lambda Calculus)}$$
$$s ::= W/x; s \mid \mathrm{id} \qquad \text{(Substitution)}$$

Beta-reduction.

$$(\lambda x.M)[s]\, W \to M[W/x; s] \qquad \text{(B)}$$

Substitution elimination.

$$(MN)[s] \to M[s]\, N[s] \qquad \text{(App)}$$
$$x[\ldots W/x \ldots] \to W \qquad \text{(Var)}$$

Figure6. The Weak Lambda Calculus of Explicit Substitutions $\lambda \mathrm{x}_w$.

pute the result of an operation. It always denote a term closed by the distribution of the environment. There is an evaluation context corresponding to each construct of the language. Evaluation contexts are given *addresses*, denoted by $a, b, \ldots$, taken from an infinite and denumerable set $\mathbf{A}$. Addresses were initially introduced for λx in [Ros96] (and later on for $\lambda\sigma_w$ in [Ben97, BLLR99]) to model sharing, particularly in functional languages whose underlying evaluation strategy is *call-by-need* (the so-called *lazy languages*). In this paper, addresses are convenient to model the implementation of objects as well. Intuitively, E^a models a reference to the term E at address a. Therefore, we use subjacent notions of *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 Section 7) means that, if a subterm E at address a is reduced to a term U, then all the subterms at the same address a are reduced in the same step to U. In other words, the simultaneous rewriting is the rewrite relation preserving admissibility.

Example 4. The term $((V\, U)^b\, U)^c$ where $U \equiv ((\lambda x.x)[\mathrm{id}]^d\, \mathsf{true}[\mathrm{id}]^e)^f$ and V is any evaluation context, may reduce in one step by rule (B) of Figure 8 to

$$((V\, x[\mathsf{true}[\mathrm{id}]^e/x; \mathrm{id}]^f)^b\, x[\mathsf{true}[\mathrm{id}]^e/x; \mathrm{id}]^f)^c$$

but not to, *e.g.*,

$$((V\, \underline{x[\mathsf{true}[\mathrm{id}]^e/x; \mathrm{id}]^f})^b\, \underline{((\lambda x.x)[\mathrm{id}]^d\, \mathsf{true}[\mathrm{id}]^e)^f})^c$$

since the two distinct underlined subterms have a same address, namely f.

Code

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

where x ranges over variables, c ranges over literal constants and m ranges over method names.

Evaluation Contexts

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

where everywhere a, b, c range over an infinite set $\mathbf{A}$ 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

$$
\begin{array}{rll}
s ::= & U/x; s \mid \mathrm{id} & \text{(Substitution)}
\end{array}
$$

Evaluation contexts $U, V, \ldots$ and Object-structures O, O' may also be written $E^a, F^b, \ldots$ when convenient.

Figure7. The Syntax of $\lambda\mathcal{O}bj^{+a}$.

"Fresh" addresses are often provided to evaluation contexts, while distributing the environment. A fresh address is an address unused in a global term. Intuitively, the address of an evaluation context is the address where the result of the computation will be stored. A *closure* $M[s]^a$ is analogous to a closure in $\lambda \mathrm{x}_w$, but it is given an address, and the terms in s are also addressed terms. $\lceil O \rceil^a$ represents an object whose *internal* object-structure is O and whose object-identity is a. In other words, the address a is the (an) *entry point* of the object-structure O. $(O \hookleftarrow m)^a$ is the evaluation context associated to a method-lookup *i.e.*, the scanning of the object-structure to find the method m. $\hookleftarrow$ is an auxiliary operator, reminiscent to the selection operator Sel of $\lambda\mathcal{O}bj^{+}$, invoked when one sends a message to an object. $\bullet^a$ is a *back pointer* [Ros96], its rôle will be explained in Subsection 5.4 when we will deal with the cyclic aspects of objects *i.e.*, the possibility to create "loops in the store". $\bullet^a$ is the only term which 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 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 term 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 $\bullet^a$.

$$\begin{aligned}
\big((\lambda x.M)[s]^b\ U\big)^a &\to M[U/x; s]^a && && \text{(B)}\\
(M\,N)[s]^a &\to (M[s]^b\, N[s]^c)^a && b, c \text{ fresh} && \text{(App)}\\
x[\ldots U/x\ldots]^a &\to U && && \text{(VarG)}\\
x[\ldots E^b/x\ldots]^a &\to E^a && \text{where } E \equiv (\lambda y.M)[s] \text{ or } c[s] && \text{(VarE)}
\end{aligned}$$

Figure8. The Module L.

5.1 The Module L

The module L is the calculus $\lambda \mathrm{x}_w$, presented in Subsection 4.2, to which addresses have been added. It is presented on Figure 8. It is almost the calculus $\lambda\sigma^a_w$ of Benaissa, Lang, Lescanne, and Rose [BLLR99], but with variable names

instead of de Bruijn index. Addresses allow to give account to *sharing* in implementations of functional programming languages, particularly lazy (or call by need) languages.

The rules of L are almost the same as the rules of λx_w, but they handle address management and sharing. The main idea is that if an evaluation context (*i.e.*, an addressed term) is located at address a, then its reducts will also be located at address a. Therefore, the term is only reduced once since all its occurrences share the result.

(B) does not need more explanation than the ones given in Section 4.2. Note only how the result of the reduction of the evaluation context located at address a is "stored" at address a. (App) creates two new evaluation contexts located at new addresses b and c. Note that, since it duplicates the environment s, it creates a sharing among the addresses which s contains. This will be the case for every other rule which duplicates an environment. Here again, the right hand side of the reduction is stored at the same address as the left hand side.

The main subtlety in this module is in rules (VarG, VarE). Indeed, we have two rules corresponding to the single rule (Var) of λx_w. The reason is due to the choice that can be made on the address where to store the right hand side: either the address of the main evaluation context (VarE) or the address of the evaluation context bound to x (VarG). Actually, both choices are valid, and the main difference is in the amount of sharing provided by each solution. A very detailed discussion on this point can be found in the above citations.

The side condition on (VarE) was absent in the original presentation. However, it is crucial because of side effects, due to adopting an imperative semantics for objects. We want two occurrences of a variable to represent a "unique" object-identity. In a calculus of functions and addresses (like the original $\lambda\sigma_w^a$) this restriction disappears. Anyway, this is not too strong a restriction as efficient implementations of functional languages (call by value or call by need) use the same slightly restricted version of the rule.

5.2 The Common Object Module C

The Common Object module is shown on Figure 9. It handles object instantiation and message sending. *Object instantiation* is characterized by (NO) 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 five remaining rules, namely (SP) which propagates the environment into the receiver of the message, (SA) which performs the self-application, (SG, SE, NL) which perform the method-lookup. Note that the message sending operator $\Leftarrow$ is *pre-computed* so that the object it is applied to is given an object-structure and an object-identity (SP). When this is the case (SA), the object is looked up and the result applied to the object itself. Note that we find in (SG, SE) a choice similar to the *graph-based vs. environment-based access* in functional languages, characterized by rules (VarG, VarE). This is not surprising, since now there is two ways to access functions: through environments and through objects (the latter are called methods).

Instantiation

$$\langle\rangle[s]^a \to \lceil\langle\rangle^b\rceil^a \qquad b \text{ fresh} \qquad \text{(NO)}$$

Message Sending

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

$$\left(\lceil O\rceil^b \Leftarrow m\right)^a \to \left((O \hookleftarrow m)^c \lceil O\rceil^b\right)^a \qquad c \text{ fresh} \qquad \text{(SA)}$$

$$(\langle O \leftarrow m = V\rangle^b \hookleftarrow m)^a \to V \qquad \text{(SG)}$$

$$(\langle O \leftarrow m = E^c\rangle^b \hookleftarrow m)^a \to E^a \qquad \text{(SE)}$$

$$(\langle O \leftarrow n = V\rangle^b \hookleftarrow m)^a \to (O \hookleftarrow m)^a \qquad \text{(NL)}$$

Figure9. The Common Object Module C.

Functional Update

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

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

Figure10. The Functional Object Module F.

5.3 The Functional Object Module F

The operational semantics of the Functional Object module is given by two rules (Figure 10). (FP) pre-computes the functional update, installing the evaluation context needed to actually proceed. (FC) describes the actual update of an object of identity b. The update is not made in place and no side effect is performed, but the result is a new object (with a new object-identity a). This is why we call this operator "functional".

5.4 The Imperative Object Module I

The Imperative Object module, shown on Figure 11, contains rules for the imperative update and cloning primitives. Imperative update is formalized in a way close to the functional update. (IP) and (IC) are much like (FP) and (FC), but they differ in address management. 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

Imperative Update

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

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

Cloning Primitives

$$\mathsf{shallow}(x)[\ldots \lceil O \rceil^b / x \ldots]^a \to \lceil O \rceil^a \qquad \text{(SC)}$$

$$\mathsf{refresh}(x)[\ldots \lceil O \rceil^b / x \ldots]^a \to \lceil \mathsf{copy}(O)^c \rceil^b \qquad c \text{ fresh} \qquad \text{(RE)}$$

$$\mathsf{copy}(\langle\rangle^b)^a \to \langle\rangle^a \qquad \text{(CE)}$$

$$\mathsf{copy}(\langle O \leftarrow m = V \rangle^b)^a \to \langle \mathsf{copy}(O)^c \leftarrow m = V \rangle^a \qquad c \text{ fresh} \qquad \text{(CO)}$$

Figure11. The Imperative Object Module I.

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 side effect is performed.

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. This implies to redefine (see Section 7) the concept of simultaneous rewriting in order to include this feature.

The primitives for cloning are $\mathsf{shallow}(x)$ and $\mathsf{refresh}(x)$; $\mathsf{shallow}(x)$ creates an object-identity for an object sharing the same object-structure as x, whereas $\mathsf{refresh}(x)$ gives a new (but identical to the old one) internal structure to the object x. The operational semantics of these primitives is also given on Figure 11. $\mathsf{refresh}(x)$ calls an auxiliary operation named $\mathsf{copy}(O)$ (RE) to recursively perform a copy of the linked list (CE, CO). Note however that methods are never copied.

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

An Example of Derivation

We give an example of a reduction in $\lambda\mathcal{O}bj^{+a}$.

Example 5. Let `self_ext` be the term defined in Example 2 *i.e.*,

$$\texttt{self_ext} \triangleq \langle\langle\rangle \leftarrow \texttt{add_n} = \underbrace{\lambda \texttt{self}.\langle \texttt{self} \leftarrow \texttt{n} = \lambda \texttt{s}.1 \rangle}_{N} \rangle$$

The reduction of $(\texttt{self_ext} \Leftarrow \texttt{add_n})$ in $\lambda\mathcal{O}bj^{+a}$ is as follows:

$$
\begin{aligned}
&(\texttt{self_ext} \Leftarrow \texttt{add_n})[\mathrm{id}]^1 \\
&\quad \xrightarrow{*} (\langle\langle\rangle[\mathrm{id}]^4 \leftarrow \texttt{add_n} = N[\mathrm{id}]^3\rangle^2 \Leftarrow \texttt{add_n})^1 && (1)\\
&\quad \to (\langle\lceil\langle\rangle^5\rceil^4 \leftarrow \texttt{add_n} = N[\mathrm{id}]^3\rangle^2 \Leftarrow \texttt{add_n})^1 && (2)\\
&\quad \to (\lceil \underbrace{\langle\langle\rangle^5 \leftarrow \texttt{add_n} = N[\mathrm{id}]^3\rangle^6}_{O} \rceil^2 \Leftarrow \texttt{add_n})^1 && (3)\\
&\quad \to ((O \hookleftarrow \texttt{add_n})^7 \lceil O\rceil^2)^1 && (4)\\
&\quad \to ((\lambda\texttt{self}.\langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle)[\mathrm{id}]^3 \lceil O\rceil^2)^1 && (5)\\
&\quad \to \langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle[\lceil O\rceil^2/\texttt{self}; \mathrm{id}]^1 && (6)\\
&\quad \xrightarrow{*} \langle\lceil O\rceil^2 \leftarrow \texttt{n} = \lambda\texttt{s}.1[\lceil O\rceil^2/\texttt{self}; \mathrm{id}]^8\rangle^1 && (7)\\
&\quad \to \lceil\langle O \leftarrow \texttt{n} = \lambda\texttt{s}.1[\lceil O\rceil^2/\texttt{self}; \mathrm{id}]^8\rangle^9\rceil^1 && (8)
\end{aligned}
$$

In (1), two steps are performed to distribute the environment inside the extension (SP, FP). In (2), the empty object is given an object-structure and an object identity (NO). In (3), this new object is functionally extended (FC), hence it shares the structure of the former object but has a new object-identity. In (4), and (5), two steps are performed to look up method `add_n` (SA, SG). Here, we have chosen to apply (SG), but we could have chosen (SE) as well, hence given a fresh address to the closure representing the method `add_n` (actually the closure which is to be applied to the self object, not the one inside the object). (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 is bound to (VarG). Here, we cannot apply (VarE) because of its side condition (otherwise it would have changed the identity of `self`). (8) is simply a (FC) *i.e.*, the proceeding of a functional extension.

Functional *vs.* Imperative

We show how 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 as illustrated by the following example.

Example 6. Let the functional object $\langle M \leftarrow m = N\rangle$, obtained by inheriting the properties of the prototype M. This object can be encoded by a $\mathsf{shallow}(x)$ followed by an imperative method update as follows:

$$(\lambda x.\langle \mathsf{shallow}(x) \leftarrow\!: m = N\rangle)\, M$$

This proves the fact that $\mathsf{F} \subseteq \mathsf{I}$.

Cloning

It is possible, using the Imperative Object module, to define a clone operation. The clone defined in Figures 2 and 4 is defined as follows:

$$\begin{aligned}\mathsf{clone}(x) &\triangleq (\mathsf{refresh} \circ \mathsf{shallow})(x)\\ &\triangleq (\lambda y.\mathsf{refresh}(y))\,\mathsf{shallow}(x)\end{aligned}$$

Its intuitive semantics is illustrated by Figure 12.

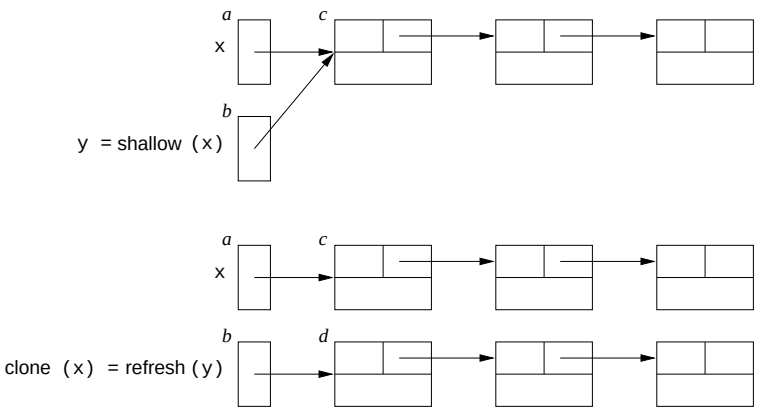


Figure12. The $\mathsf{clone}(x)$ Operator.

The clone defined in Figures 1 and 3 is defined as follows:

$$\mathsf{clone}(x) \triangleq \mathsf{shallow}(x)$$

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. Other interesting operators can be defined by combining the different features of $\lambda\mathcal{O}bj^{+a}$.

7 The Simultaneous Rewriting

Simultaneous rewriting [Ros96] 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 rewritten together, as shown in Example 4.

Vertical Admissibility, *i.e.*, for all $\bullet^a$ occuring in a term U, there exists a subterm E^a of U such that $\bullet^a$ is a proper subterm of E^a.

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

1. Match a subterm of A at address say a with a left hand side of a rule and replace all the subterms of A at address a with the corresponding right hand side (except $\bullet^a$) and create the new fresh addresses (if required). This way, one gets a term B.
2. Find in B all the subterms at an address b which occur inside another subterm at the same address b, and replace them by $\bullet^b$.
3. Find $\bullet^b$ not in the context of an E^b, and replace them by their corresponding E^b as found in A.

Points 2 and 3 must be repeated until a fix point is reached.

Example 7. The term $\langle\lceil\langle\rangle^1\rceil^2 \leftarrow\colon \mathtt{m} = (\lambda\mathtt{self.x})[\lceil\langle\rangle^1\rceil^2/\mathtt{x};\mathrm{id}]^3\rangle^4$ does not reduce to

$$\lceil\langle\langle\rangle^1 \leftarrow \mathtt{m} = (\lambda\mathtt{self.x})[\lceil\langle\rangle^1\rceil^2/\mathtt{x};\mathrm{id}]^3\rangle^4\rceil^2$$

(a non admissible term) but instead to

$$\langle\lceil\langle\rangle^1 \leftarrow \mathtt{m} = (\lambda\mathtt{self.x})[\bullet^2/\mathtt{x};\mathrm{id}]^3\rangle^4\rceil^2$$

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 (two semantically distinct subterms have a same address).

Example 8. The term $\langle\lceil\langle\langle\rangle^1 \leftarrow \mathtt{m} = M[\bullet^4/\mathtt{x};\mathrm{id}]^2\rangle^3\rceil^4 \leftarrow \mathtt{n} = N[\mathrm{id}]^5\rangle^6$ does not reduce to

$$\lceil\langle\langle\langle\rangle^1 \leftarrow \mathtt{m} = M[\bullet^4/\mathtt{x};\mathrm{id}]^2\rangle^3 \leftarrow \mathtt{n} = N[\mathrm{id}]^5\rangle^7\rceil^6$$

but instead to

$$\lceil\langle\langle\langle\rangle^1 \leftarrow \mathtt{m} = M[\lceil\bullet^3\rceil^4/\mathtt{x};\mathrm{id}]^2\rangle^3 \leftarrow \mathtt{n} = N[\mathrm{id}]^5\rangle^7\rceil^6$$

In this last term, the back pointer $\bullet^4$ has been *unfolded* following the definition of simultaneous rewriting *i.e.*, replaced by the term it refers to, namely $\lceil\bullet^3\rceil^4$ (3 is still in the context of the subterm, and therefore $\bullet^3$ is not unfolded). This unfolding is due to the removal of the surrounding address 4, which otherwise could lead to a loss of information on the shape of the term associated to the address 4.

8 Strategies

The rules of $\lambda\mathcal{O}bj^{+a}$ express the operational semantics of $\lambda\mathcal{O}bj^{+}$, independently of any evaluation strategy. Thus, in the reminder of the paper, we give strong results valid on any implementation which uses $\lambda\mathcal{O}bj^{+a}$ as its underlying model *i.e.*, independent of any strategy. However, if one wants to deal with implementations, one should address the strategy issue. For this reason, although strategies are not the main subject of this paper, we wish to give some intuition of how they can be defined in $\lambda\mathcal{O}bj^{+a}$.

Intuitively, defining a strategy is to say, given an evaluation context, where (at which address) is the subterm to reduce next. Therefore, a strategy is a binary relation between an evaluation context and an address. For convenience (or because of a professional bias!) we decided to describe it by an inference system; each inference rule is subject to conditions on its application.

The reader will find in Appendix A all the strategy rules. Let us have a look to only two characteristic rules to understand what a formal strategy is, namely:

$$\frac{}{M[s]^a \rhd a}\ (\text{Scl}) \qquad \frac{s \rhd b}{M[s]^a \rhd b}\ (\text{Sub})$$

The rules say where to reduce next in the closure $M[s]^a$. There are two answers: either one reduces the closure itself (located at address a), or one reduces in s (therefore at another address, denoted by b). (Scl) describes the first solution and (Sub) describes the second. Thus, a strategy will be associated with rule conditions like *enable* (Sub) *and disable* (Scl), or vice-versa, or something more involved like *enable* (Sub) *and disable* (Scl) *if M is a variable, otherwise disable* (Sub) *and enable* (Scl). Hence, a strategy is simply a set of inference rules with conditions. It is possible to define deterministic strategies (given a term, only one inference is possible) or non deterministic strategies (several inferences are possible). Strategies are discussed in more details in [BRL96].

9 A Type System for $\lambda\mathcal{O}bj^{+a}$

In this section, we present a type system for $\lambda\mathcal{O}bj^{+a}$. See Appendix B for the full set of rules. This type system is inspired from [GHL98] to which we refer for a precise discussion of issues related to subtyping.

9.1 Types

The type expressions are described as follows:

$$\sigma, \tau ::= \iota \mid t \mid \sigma \to \tau \mid \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle \mid \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle \mid \sigma \twoheadleftarrow m$$

$$R, R' ::= \varepsilon \mid R, m : \sigma$$

$$\kappa ::= T \mid Rgd$$

Object-types have the form $\mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle$ or $\mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle$; they are called `pro-` or `obj`-types, respectively. As in [FHM94], we may consider object-types as a form of recursively-defined types. The intended meaning of an object-type:

$$\mathtt{pro}\, t.\langle\!\langle m_1 : \sigma_1 \ldots m_h : \sigma_h \lhd m_{h+1} : \sigma_{h+1} \ldots m_k : \sigma_k \rangle\!\rangle, \quad \text{with } k \geq 0$$

is the following:

- $m_1, \ldots, m_h$ are the methods that can be invoked; we say that these methods belong to the *interface* part of the object-type.
- $m_{h+1}, \ldots, m_k$ are the methods that cannot be invoked; they are *reserved*, *i.e.*, they belong to the *reservation* part of the object-type. We can extend an object M with a new method m having type σ only if it is possible to assign to M an object-type of the form $\mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle$.

The intended meaning of an `obj`-type is the same as a `pro`-type except that an object assigned to an `obj`-type can be *covariantly subsumed and extended only with the methods contained in its reservation-part*. Moreover, an `obj`-type has "kind" *Rgd*. The operator $\twoheadleftarrow$ is used to add new methods to an object-type; essentially it is the "type counterpart" of the operators $\leftarrow$ and $\leftarrow:$.

9.2 Contexts and Judgments

The contexts have the following shape:

$$\Gamma ::= \varepsilon \mid \Gamma, x : \sigma \mid \Gamma, t \lessdot\!\!\# \sigma \mid \Gamma, a : \sigma$$

The only difference between these contexts and the ones of [GHL98] lies in that we add also declarations of addresses. Our type assignment system uses judgments of the following shapes:

(1) $\quad \Gamma \vdash ok, \quad \Gamma \vdash \sigma : T, \quad \Gamma \vdash \sigma : Rgd, \quad \Gamma \vdash M : \sigma, \quad \Gamma \vdash \sigma \lessdot\!\!\# \tau, \quad \Gamma \vdash \sigma \xrightarrow{type} \tau$

(2) $\quad \Gamma \vdash U : \sigma, \quad \Gamma \vdash O : \sigma \quad \Gamma \vdash a : \sigma$

The intended meaning of the judgments in the set (1) is the same as in [GHL98]. In particular, the judgment $\Gamma \vdash \sigma \lessdot\!\!\# \tau$ (read σ *matches* τ) means that σ is the type of a possible update of an object having type τ. As in the original spirit of matching [Bru94], this judgment formalizes the ability to "inherit" method types. The judgment $\Gamma \vdash \sigma \xrightarrow{type} \tau$ (read σ *reduces to* τ) expresses a limited form of "type-conversion" which amounts to simplify occurrences of $\twoheadleftarrow$.

The remaining judgments in the set (2) are peculiar to the framework: they give a type to the evaluation contexts, object-structures, and addresses respectively (addresses are treated as special variables). We present the most interesting rules.

9.3 Type Rules for Object Expressions

The type rules relative to judgments in the set (1) contain the object expressions of $\lambda\mathcal{O}bj^{+a}$; they are mostly the ones of [GHL98]. One important remark is that both functional and imperative object update are treated by the same rule (*Object*); this is essentially due to the fact that our extension always "attaches" a new method in front of its object-structure, keeping or not its object-identity, whether or not we use imperative traits. Moreover, observe that the rules (*Refresh*) and (*Shallow*) applies only when the variable x, occurring in the premise, refers to an object (*i.e.*, is declared in the context with an object-type).

9.4 Type Rules for Evaluation Contexts and Object-structures

For the set (2) we present the most interesting rules. The rules for the object-structures are self-explanatory and need no comments.

The (*Eval−Closure*) rule

$$\frac{\begin{array}{l}\Gamma \equiv \Gamma_1, \ldots, \Gamma_{n+1} \\ \Gamma' \equiv \Gamma_1, \ldots, x_i : \tau_i, \Gamma_{i+1}, x_{i+1} : \tau_{i+1}, \ldots, \Gamma_{n+1} \\ \Gamma' \vdash M : \sigma \quad \Gamma \vdash a : \sigma \quad \Gamma \vdash U_i : \tau_i \quad \forall\, i = 1 \ldots n\end{array}}{\Gamma \vdash M[U_1/x_1; \ldots; U_n/x_n; \mathrm{id}]^a : \sigma} \; (Eval{-}Closure)$$

assigns a type σ to a closure. This requires the code and the address (where the closure is memorized) to be of the same type σ, and the types of the free variables of M to be of the same type as the corresponding evaluation contexts occurring in the substitution. Note that the assumptions needed to give a type to the free variables of M are not used in the conclusion of the judgment, since a closure has no free variables at all.

The (*Eval−Send*) and (*Eval−Object*) rules are essentially inspired from the corresponding type rules for object expressions.

The (*Eval−Identity*) rule

$$\frac{\Gamma \vdash O : \sigma \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash \sigma <\!\!\# \tau}{\Gamma \vdash \lceil O \rceil^a : \sigma}$$

assigns a type to an evaluation context associated with an object; if the object-structure has type σ, its address has a type τ and σ matches τ, then $\lceil O \rceil^a$ has type σ. Due to imperative traits, it follows that the object-structure and its address may have different "matched" types.

The (*Eval−Lookup*) rule

$$\frac{\begin{array}{ll}\Gamma \vdash O : \tau & \Gamma \vdash \tau <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle \\ \Gamma \vdash \rho <\!\!\# \tau & \Gamma \vdash a : (t \to \sigma)[\rho/t]\end{array}}{\Gamma \vdash (O \leftarrow m)^a : (t \to \sigma)[\rho/t]}$$

is complex. It assigns a type to the body of a method m attached to an object O. Since a method is nothing but a function whose first formal parameter refers always to "self", it follows that the type of the conclusion must be an arrow-type. The operational semantics of the operator $\hookleftarrow$ is destructive *i.e.*, it destroys the object-structure of the receiver of message m. Therefore, the type ρ represents the type of the receiver of the message m.

The $(Eval-Back-Pointer)$ rule

$$\frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash \sigma <\!\# \tau}{\Gamma \vdash \bullet^a : \sigma}$$

says in its premise that terms at address a have type τ such that σ matches τ. Again the matching judgment $\sigma <\!\# \tau$ is required by imperative traits.

10 Properties of $\lambda\mathcal{O}bj^{+a}$

In this section we show some properties of $\lambda\mathcal{O}bj^{+a}$. First of all, we give three functions (called *decompilation functions*) which translate the expressions of $\lambda\mathcal{O}bj^{+a}$ into those of $\lambda\mathcal{O}bj^{+}$. The functions $\mathcal{C}[\![_]\!]$, $\mathcal{E}[\![_]\!]$ and $\mathcal{O}[\![_]\!]$ are mutually recursive and are defined in Figure 13.

The following theorem expresses the confluence of the functional fragment of our framework.

Theorem 1 (Confluence of Functional Fragment).
$\mathsf{L} + \mathsf{C} + \mathsf{F}$ *is confluent modulo address erasure.*

Proof. The proof lies on the fact that the system is orthogonal (left linear and with no critical pair), hence strongly confluent (each confluent diagram can be closed in one or zero step. Actually, there are just two exceptions to the absence of critical pair, namely (VarG, VarE) and (SG, SE) which overlap. It has been shown [BLLR99] that (VarG, VarE) make the rewrite system strongly confluent modulo address erasure anyway, and this result can be easily generalized to (SG, SE). Moreover, the fact that we use simultaneous rewriting does not change the result on strong confluence of the rewrite system. □

Of course this result does not hold for the whole $\lambda\mathcal{O}bj^{+a}$, since as well known, imperative aspects break the confluence.

The following theorem expresses the soundness of the functional part of our system *w.r.t.* $\lambda\mathcal{O}bj^{+}$ *i.e.*, the fact that to any reduction in $\mathsf{L} + \mathsf{C} + \mathsf{F}$ corresponds an equivalent reduction in $\lambda\mathcal{O}bj^{+}$, and the fact that any $\lambda\mathcal{O}bj^{+}$-normal form can be computed via the system $\mathsf{L} + \mathsf{C} + \mathsf{F}$ (*i.e.*, without the imperative module I).

Theorem 2 (Soundness and Weak Completeness).

1. *(Soundness) If* $U \stackrel{*}{\Rightarrow} V$ *in* $\mathsf{L} + \mathsf{C} + \mathsf{F}$ *then* $\mathcal{E}[\![U]\!] \stackrel{*}{\Rightarrow} \mathcal{E}[\![V]\!]$ *in* $\lambda\mathcal{O}bj^{+}$.

Code

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &= x \\
\mathcal{C}[\![\mathsf{shallow}(x)]\!] &= x \\
\mathcal{C}[\![\mathsf{refresh}(x)]\!] &= x \\
\mathcal{C}[\![\lambda x.M]\!] &= \lambda x.\mathcal{C}[\![M]\!] \\
\mathcal{C}[\![M\,N]\!] &= \mathcal{C}[\![M]\!]\,\mathcal{C}[\![N]\!] \\
\mathcal{C}[\![c]\!] &= c \\
\mathcal{C}[\![\langle\rangle]\!] &= \langle\rangle \\
\mathcal{C}[\![\langle M \leftarrow m = N\rangle]\!] &= \langle \mathcal{C}[\![M]\!] \leftarrow m = \mathcal{C}[\![N]\!]\rangle \\
\mathcal{C}[\![\langle M \leftarrow\!: \; m = N\rangle]\!] &= \langle \mathcal{C}[\![M]\!] \leftarrow m = \mathcal{C}[\![N]\!]\rangle \\
\mathcal{C}[\![M \Leftarrow m]\!] &= \mathcal{C}[\![M]\!] \Leftarrow m
\end{aligned}
$$

Evaluation contexts

$$
\begin{aligned}
\mathcal{E}[\![M[U/x;s]^a]\!] &= \mathcal{E}[\![M\{\mathcal{E}[\![U]\!]/x\}[s]^a]\!] \\
\mathcal{E}[\![M[\mathrm{id}]^a]\!] &= \mathcal{C}[\![M]\!] \\
\mathcal{E}[\![(UV)^a]\!] &= \begin{cases} Sel(\mathcal{O}[\![O]\!], m, \mathcal{E}[\![V]\!]) & \text{if } U \equiv (O \hookleftarrow m)^b \\ \mathcal{E}[\![U]\!]\,\mathcal{E}[\![V]\!] & \text{otherwise} \end{cases} \\
\mathcal{E}[\![(U \Leftarrow m)^a]\!] &= \mathcal{E}[\![U]\!] \Leftarrow m \\
\mathcal{E}[\![\langle U \leftarrow m = V\rangle^a]\!] &= \langle \mathcal{E}[\![U]\!] \leftarrow m = \mathcal{E}[\![V]\!]\rangle \\
\mathcal{E}[\![\langle U \leftarrow\!: \; m = V\rangle^a]\!] &= \langle \mathcal{E}[\![U]\!] \leftarrow m = \mathcal{E}[\![V]\!]\rangle \\
\mathcal{E}[\![\lceil O \rceil^a]\!] &= \mathcal{O}[\![O]\!]
\end{aligned}
$$

Object-structures

$$
\begin{aligned}
\mathcal{O}[\![\langle\rangle^a]\!] &= \langle\rangle \\
\mathcal{O}[\![\langle O \leftarrow m = V\rangle^a]\!] &= \langle \mathcal{O}[\![O]\!] \leftarrow m = \mathcal{E}[\![V]\!]\rangle \\
\mathcal{O}[\![\mathsf{copy}(O)^a]\!] &= \mathcal{O}[\![O]\!]
\end{aligned}
$$

Figure13. Translation from $\lambda\mathcal{O}bj^{+\,a}$ to $\lambda\mathcal{O}bj^{+}$ (decompilation).

2. *(Weak Completeness) If* $M \twoheadrightarrow_w N$ *(read "weakly reduces") in* $\lambda\mathcal{O}bj^+$ *and* N *is a normal form, then there exists* V *such that* $M[\mathrm{id}]^a \twoheadrightarrow V$ *in* $\mathsf{L}+\mathsf{C}+\mathsf{F}$ *and* $\mathcal{E}[\![V]\!] \equiv N$.

Proof.

1. It is clear that every rule of $\mathsf{L}+\mathsf{C}+\mathsf{F}$ translates either to a rule of $\lambda\mathcal{O}bj^+$, or to an identity, except (SG, SE, NL) which must occur in the left hand side of an application in order to catch the semantics of Sel. However, it is clear from the rules that this is always the case. Addresses only generalize these rules in the sense that a reduction can occur simultaneously in different parts of a term. However, it has been shown in [BLLR99] that this is sound and this result may easily be generalized to our calculus. □
2. By weak reduction we mean that reductions never occur under a lambda abstraction. It is clear from the definition of the translation function that if there is no redex in V then there may not be any redex in $\mathcal{E}[\![V]\!]$. Moreover, every rule of $\lambda\mathcal{O}bj^+$ has an implementation in $\mathsf{L}+\mathsf{C}+\mathsf{F}$. Then, the result is a direct consequence of the soundness (Theorem 2) and of the confluence (Theorem 1). □

In the following lemma, we assume $\pi * \Pi$ be either $x : \sigma$, or $a : \sigma$, or $t \lessdot\!\!\# \sigma$, and $A * B$ be any provable statement in our system.

Lemma 3 (Weakening).
If $\Gamma, \Gamma' \vdash A * B$ *and* $\Gamma, \pi * \Pi, \Gamma' \vdash ok$, *then* $\Gamma, \pi * \Pi, \Gamma' \vdash A * B$.

The subject reduction theorem can be stated as follows.

Theorem 4 (Subject Reduction for $\mathsf{L}+\mathsf{C}+\mathsf{F}$).
If $\Gamma \vdash U : \sigma$ *and* $U \to V$, *then there exists* Γ', *such that* $\Gamma' \vdash V : \sigma$ *and* $\Gamma \subseteq \Gamma'$.

Proof. Consider the simultaneous rewriting of the calculus $\mathsf{L}+\mathsf{C}+\mathsf{F}$ as first order when the reduction occurs in an empty context. We prove that the type is preserved by each case of the reduction rules. Most of the rules are immediate or follow by induction hypothesis and an application of Lemma 3 in order to give a type to fresh addresses of the term occurring in the right hand side of the reduction. Note that we do not have to resort to a standard substitution lemma for rule (B) (unlike the lambda calculus), since the closure occurring in the right hand side of the rule does not have free variables.

Then the thesis follows by observing that a simultaneous rewriting step occurring in any context can be simulated as a sequence of the above standard rewriting steps. □

Definition 5. Define the set of wrong terms as follows:

$$\mathtt{wrong} ::= (\langle\rangle^b \hookleftarrow m)^a \;\mid\; (c[s]^b \Leftarrow m)^a \;\mid\; ((\lambda x.M)[s]^b \Leftarrow m)^a$$

By inspecting the typing rules, one can see that `wrong` can not be typed, hence the type soundness for $\mathsf{L}+\mathsf{C}+\mathsf{F}$ follows as a corollary of the subject reduction theorem *i.e.*,

Corollary 6 (Type Soundness for $\mathsf{L}+\mathsf{C}+\mathsf{F}$).
If $\Gamma \vdash U : \tau$, then U does not reduce to $C[\mathtt{wrong}]$ in $\mathsf{L}+\mathsf{C}+\mathsf{F}$, where $C[\,]$ is a generic context in $\lambda\mathcal{O}bj^{+a}$ i.e., a term with an "hole" inside it.

Conjecture 7 (Type Soundness for $\lambda\mathcal{O}bj^{+a}$).
If $\Gamma \vdash U : \tau$, then U does not reduce to $C[\mathtt{wrong}]$ in $\lambda\mathcal{O}bj^{+a}$.

Hint. The presence of imperative traits induces some complications in the proof. Therefore, a simultaneous rewriting *cannot* be simulated as a simple sequence of first order rewriting steps. We feel that those technical problems will be fixed soon, but at the time of the submission, we do not have proven all the cases.

11 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 and distribution of objects across networks. Other useful extensions of this calculus should be studied, such as providing a real imperative override of fields *i.e.*, a real *field look up and replacement*. To this aim, a distinction has to be done between fields (and may be more generally procedures, functions that do not have a self-reference) and methods, since it is known that overriding methods this way is not type sound [BF98] in presence of object 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

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Ben97] 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.

[BF98] 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.

[BLLR99] Z.-E.-A. Benaissa, F. Lang, P. Lescanne, and K.H. Rose. Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. *The Journal of Functional and Logic Programming*, 1999. Accepted, extended version of [BRL96].

[BRL96] 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.

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

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Cha93] C. Chambers. The Cecil language specification, and rationale. Technical Report 93-03-05, University of Washington, Dept. of Computer Science and Engineering, 1993.

[CHL96] 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.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[GHL98] 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.

[Liq97] L. Liquori. An Extended Theory of Primitive Objects: First Order System. In *Proc. of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 146–169. Springer-Verlag, 1997.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Company, 1996.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, Universitetsparken 1, DK-2100 København Ø, 1996. DIKU report 96/1.

[Tai92] A. Tailvalsaari. Kevo, a prototype-based object-oriented language based on concatenation and modules operations. Technical report, University of Victoria, 1992.

[US87] D. Ungar and B. Smith, R. Self: The power of simplicity. In *Proc. of OOPSLA*, pages 227–241. The ACM Press, 1987.

A The Strategy Rules

Closures

$$\frac{}{M[s]^a \rhd a}\ (\text{Scl}) \qquad \frac{s \rhd b}{M[s]^a \rhd b}\ (\text{Sub}) \qquad \frac{U \rhd a}{U/x \cdot s \rhd a}\ (\text{Hd}) \qquad \frac{s \rhd a}{U/x \cdot s \rhd a}\ (\text{Tl})$$

Applications

$$\frac{}{(UV)^a \rhd a}\ (\text{Sap}) \qquad \frac{U \rhd b}{(UV)^a \rhd b}\ (\text{Lap}) \qquad \frac{V \rhd b}{(UV)^a \rhd b}\ (\text{Rap})$$

Message Sending

$$\frac{}{(U \Leftarrow m)^a \rhd a}\ (\text{Ssn}) \qquad \frac{U \rhd b}{(U \Leftarrow m)^a \rhd b}\ (\text{Lsn})$$

Functional Update

$$\frac{}{\langle U \leftarrow m = V\rangle^a \rhd a}\ (\text{Sfe}) \quad \frac{U \rhd b}{\langle U \leftarrow m = V\rangle^a \rhd b}\ (\text{Lfe}) \quad \frac{V \rhd b}{\langle U \leftarrow m = V\rangle^a \rhd b}\ (\text{Rfe})$$

Imperative Update

$$\frac{}{\langle U \leftarrow\!: \ m = V\rangle^a \rhd a}\ (\text{Sie}) \quad \frac{U \rhd b}{\langle U \leftarrow\!: \ m = V\rangle^a \rhd b}\ (\text{Lie}) \quad \frac{V \rhd b}{\langle U \leftarrow\!: \ m = V\rangle^a \rhd b}\ (\text{Rie})$$

Objects

$$\frac{O \rhd b}{\lceil O \rceil^a \rhd b}\ (\text{Obj}) \qquad \frac{O \rhd b}{\langle O \leftarrow m = V\rangle^a \rhd b}\ (\text{Oob}) \qquad \frac{V \rhd b}{\langle O \leftarrow m = V\rangle^a \rhd b}\ (\text{Mob})$$

Look up

$$\frac{}{(O \hookleftarrow m)^a \rhd a}\ (\text{Slk}) \qquad \frac{O \rhd b}{(O \hookleftarrow m)^a \rhd b}\ (\text{Llk})$$

Copy

$$\frac{}{\mathsf{copy}(O)^a \rhd a}\ (\text{Cop}) \qquad \frac{O \rhd b}{\mathsf{copy}(O)^a \rhd b}\ (\text{Nor})$$

B The Type System

Rules for Well-formed Contexts

$$\frac{}{\varepsilon \vdash ok}\ (Cont{-}\varepsilon)$$

$$\frac{\Gamma \vdash \sigma : T \quad x \notin dom(\Gamma)}{\Gamma, x : \sigma \vdash ok}\ (Cont{-}x)$$

$$\frac{\Gamma \vdash \sigma : T \quad a \notin dom(\Gamma)}{\Gamma, a : \sigma \vdash ok}\ (Cont{-}a)$$

$$\frac{\Gamma \vdash \sigma : T \qquad t \notin dom(\Gamma)}{\Gamma, t \lessdot\!\!\# \sigma \vdash ok} \quad (Cont\text{–}t)$$

Rules for Well-formed Types

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \iota : T} \quad (Type\text{–}Const)$$

$$\frac{\Gamma, t \lessdot\!\!\# \sigma, \Gamma' \vdash ok}{\Gamma, t \lessdot\!\!\# \sigma, \Gamma' \vdash t : T} \quad (Type\text{–}Var)$$

$$\frac{\Gamma \vdash \sigma : T \qquad \Gamma \vdash \tau : T}{\Gamma \vdash \sigma \to \tau : T} \quad (Type\text{–}Arrow)$$

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle \lhd \rangle\!\rangle : T} \quad (Type\text{–}Pro\text{–}Empty)$$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle \lhd R, R' \rangle\!\rangle : T}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T} \quad (Type\text{–}Pro\text{–}Left)$$

$$\frac{\Gamma, t \lessdot\!\!\# \mathtt{pro}\, t.\langle\!\langle \lhd R \rangle\!\rangle \vdash \sigma : T \qquad m \notin \mathcal{M}(R)}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle \lhd R, m : \sigma \rangle\!\rangle : T} \quad (Type\text{–}Pro\text{–}Right)$$

$$\frac{\Gamma \vdash \tau \lessdot\!\!\# \mathtt{pro}\, t.\langle\!\langle \lhd R, m : \sigma \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow m : T} \quad (Type\text{–}Extend\text{–}Pro)$$

$$\frac{\Gamma \vdash \tau \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle \lhd R, m : \sigma \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow m : T} \quad (Type\text{–}Extend\text{–}Obj)$$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T \qquad t \text{ covariant in } R, R'}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T} \quad (Type\text{–}Obj)$$

Rules for Well-formed Rigid Types

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \iota : Rgd} \quad (Rgd\text{–}Const)$$

$$\frac{\Gamma \vdash \sigma : Rgd}{\Gamma, t \lessdot\!\!\# \sigma, \Gamma' \vdash t : Rgd} \quad (Rgd\text{–}Var)$$

$$\frac{\Gamma \vdash \sigma : Rgd \qquad \Gamma \vdash \tau : Rgd}{\Gamma \vdash \sigma \to \tau : Rgd} \quad (Rgd\text{–}Arrow)$$

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : Rgd} \quad (Rgd{-}Obj)$$

$$\frac{\Gamma \vdash \tau : Rgd \qquad \Gamma \vdash \tau \twoheadleftarrow m : T}{\Gamma \vdash \tau \twoheadleftarrow m : Rgd} \quad (Rgd{-}Extend)$$

Rules for Matching

$$\frac{\Gamma, t \lessdot\!\# \sigma, \Gamma' \vdash ok}{\Gamma, t \lessdot\!\# \sigma, \Gamma' \vdash t \lessdot\!\# \sigma} \quad (Match{-}Var)$$

$$\frac{\Gamma \vdash \sigma \lessdot\!\# \tau \qquad \Gamma \vdash \tau \lessdot\!\# \rho}{\Gamma \vdash \sigma \lessdot\!\# \rho} \quad (Match{-}Trans)$$

$$\frac{\Gamma \vdash \sigma : T}{\Gamma \vdash \sigma \lessdot\!\# \sigma} \quad (Match{-}Refl)$$

$$\frac{\Gamma \vdash \sigma' \lessdot\!\# \sigma \qquad \Gamma \vdash \tau \lessdot\!\# \tau'}{\Gamma \vdash \sigma \to \tau \lessdot\!\# \sigma' \to \tau'} \quad (Match{-}Arrow)$$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T \qquad \Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle : T}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle \lessdot\!\# \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle} \quad (Match{-}Book{-}Pro)$$

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T \qquad \Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle : T}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle \lessdot\!\# \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle} \quad (Match{-}Book{-}Obj)$$

$$\frac{\Gamma \vdash \tau \twoheadleftarrow m : T}{\Gamma \vdash \tau \twoheadleftarrow m \lessdot\!\# \tau} \quad (Match{-}Inherit)$$

$$\frac{\Gamma \vdash \tau \lessdot\!\# \tau' \qquad \Gamma \vdash \tau' \twoheadleftarrow m : T}{\Gamma \vdash \tau \twoheadleftarrow m \lessdot\!\# \tau' \twoheadleftarrow m} \quad (Match{-}Extend)$$

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle \lessdot\!\# \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle} \quad (Promote)$$

$$\frac{\Gamma \vdash \sigma \xrightarrow{type} \tau}{\Gamma \vdash \sigma \lessdot\!\# \tau} \quad (Match{-}Red{-}Left)$$

$$\frac{\Gamma \vdash \tau \xrightarrow{type} \sigma}{\Gamma \vdash \sigma \lessdot\!\# \tau} \quad (Match{-}Red{-}Right)$$

Rules for Type-Reduction

$$\frac{\Gamma \vdash \sigma \xrightarrow{type} \sigma' \qquad \Gamma \vdash \tau : T}{\Gamma \vdash \sigma \to \tau \xrightarrow{type} \sigma' \to \tau} \quad (Red{-}Arrow{-}Left)$$

$$\frac{\Gamma \vdash \tau \xrightarrow{type} \tau' \qquad \Gamma \vdash \sigma : T}{\Gamma \vdash \sigma \to \tau \xrightarrow{type} \sigma \to \tau'} \quad (Red{-}Arrow{-}Right)$$

$$\frac{\Gamma \vdash \tau \twoheadleftarrow m : T \qquad \Gamma \vdash \tau \xrightarrow{type} \tau'}{\Gamma \vdash \tau \twoheadleftarrow m \xrightarrow{type} \tau' \twoheadleftarrow m} \quad (Red{-}Inherit)$$

$$\frac{\Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow m \xrightarrow{type} \tau} \quad (Red{-}Over{-}Pro)$$

$$\frac{\Gamma \vdash \tau <\!\# \,\mathtt{obj}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow m \xrightarrow{type} \tau} \quad (Red{-}Over{-}Obj)$$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle : T}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle \twoheadleftarrow m \xrightarrow{type} \mathtt{pro}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle} \quad (Red{-}Ext{-}Pro)$$

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle : T}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle \twoheadleftarrow m \xrightarrow{type} \mathtt{obj}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle} \quad (Red{-}Ext{-}Obj)$$

$$\frac{\Gamma, t <\!\# \,\mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle \vdash \sigma \xrightarrow{type} \sigma'}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle \xrightarrow{type} \mathtt{pro}\, t.\langle\!\langle R, m : \sigma' \lhd R' \rangle\!\rangle} \quad (Red{-}Meth{-}Pro)$$

$$\frac{\Gamma, t <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle \vdash \sigma \xrightarrow{type} \sigma'}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle \xrightarrow{type} \mathtt{obj}\, t.\langle\!\langle R, m : \sigma' \lhd R' \rangle\!\rangle} \quad (Red{-}Meth{-}Obj)$$

Rules for Lambda Terms

$$\frac{\Gamma \vdash ok}{\Gamma \vdash c : \iota} \quad (Const)$$

$$\frac{\Gamma, x : \sigma, \Gamma' \vdash ok}{\Gamma, x : \sigma, \Gamma' \vdash x : \sigma} \quad (Var)$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \quad (Abs)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau} \quad (Appl)$$

Rules for Object Expressions

In the following, we let $\leftarrow\!*$ be either the functional $\leftarrow$ or the imperative $\leftarrow\!:$ operators.

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \langle\,\rangle : \mathtt{pro}\, t.\langle\!\langle \lhd \rangle\!\rangle} \quad (Empty)$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash \tau <\!\!\# \,\mathtt{obj}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle}{\Gamma \vdash M \Leftarrow m : \sigma[\tau/t]} \quad (Send)$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash \tau <\!\!\# \,\mathtt{obj}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle \qquad \Gamma, t <\!\!\# \tau \twoheadleftarrow m \vdash N : t \to \sigma}{\Gamma \vdash \langle M \leftarrow\!* m = N \rangle : \tau \twoheadleftarrow m} \quad (Object)$$

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash \tau <\!\!\# \,\mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle}{\Gamma \vdash \mathsf{refresh}(x) : \tau} \quad (Refresh)$$

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash \tau <\!\!\# \,\mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle}{\Gamma \vdash \mathsf{shallow}(x) : \tau} \quad (Shallow)$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma <\!\!\# \tau \qquad \Gamma \vdash \tau : Rgd}{\Gamma \vdash M : \tau} \quad (Subsume)$$

Rules for Evaluation Contexts

$$\frac{\begin{array}{l}\Gamma \equiv \Gamma_1, \ldots, \Gamma_{n+1} \\ \Gamma' \equiv \Gamma_1, \ldots, x_i : \tau_i, \Gamma_{i+1}, x_{i+1} : \tau_{i+1} \ldots, \Gamma_{n+1} \\ \Gamma' \vdash M : \sigma \quad \Gamma \vdash a : \sigma \quad \Gamma \vdash U_i : \tau_i \quad \forall\, i = 1 \ldots n\end{array}}{\Gamma \vdash M[U_1/x_1; \ldots; U_n/x_n; \mathrm{id}]^a : \sigma} \quad (Eval{-}Closure)$$

$$\frac{\Gamma \vdash U : \sigma \to \tau \qquad \Gamma \vdash V : \tau \qquad \Gamma \vdash a : \tau}{\Gamma \vdash (UV)^a : \tau} \quad (Eval{-}Appl)$$

$$\frac{\Gamma \vdash U : \tau \qquad \Gamma \vdash \tau <\!\!\# \,\mathtt{pro}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle \qquad \Gamma \vdash a : \sigma[\tau/t]}{\Gamma \vdash (U \Leftarrow m)^a : \sigma[\tau/t]} \quad (Eval{-}Send)$$

$$\frac{\Gamma \vdash U : \tau \qquad \Gamma \vdash \tau \lessdot\!\!\# \,\mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle \quad \Gamma, t \lessdot\!\!\# \tau \twoheadleftarrow m \vdash V : t \to \sigma \qquad \Gamma \vdash a : \tau \twoheadleftarrow m}{\Gamma \vdash \langle U \leftarrow\!\!* \, m = V \rangle^a : \tau \twoheadleftarrow m} \quad (Eval\text{-}Object)$$

$$\frac{\Gamma \vdash O : \tau \qquad \Gamma \vdash \tau \lessdot\!\!\# \,\mathtt{pro}\, t.\langle\!\langle R, m : \sigma \lhd R' \rangle\!\rangle \quad \Gamma \vdash \rho \lessdot\!\!\# \tau \qquad \Gamma \vdash a : (\sigma \to \tau)[\rho/t]}{\Gamma \vdash (O \hookleftarrow m)^a : (\sigma \to \tau)[\rho/t]} \quad (Eval\text{-}Lookup)$$

$$\frac{\Gamma \vdash O : \sigma \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash \sigma \lessdot\!\!\# \tau}{\Gamma \vdash \lceil O \rceil^a : \sigma} \quad (Eval\text{-}Identity)$$

$$\frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash \sigma \lessdot\!\!\# \tau}{\Gamma \vdash \bullet^a : \sigma} \quad (Eval\text{-}Back\text{-}Pointer)$$

Rules for Object-Structures

$$\frac{\Gamma \vdash a : \mathtt{pro}\, t.\langle\!\langle \lhd \rangle\!\rangle}{\Gamma \vdash \langle \rangle^a : \mathtt{pro}\, t.\langle\!\langle \lhd \rangle\!\rangle} \quad (Structure\text{-}Empty)$$

$$\frac{\Gamma \vdash O : \tau \qquad \Gamma \vdash \tau \lessdot\!\!\# \,\mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle \quad \Gamma, t \lessdot\!\!\# \tau \twoheadleftarrow m \vdash V : t \to \sigma \qquad \Gamma \vdash a : \tau \twoheadleftarrow m}{\Gamma \vdash \langle O \leftarrow m = V \rangle^a : \tau \twoheadleftarrow m} \quad (Structure\text{-}Object)$$

$$\frac{\Gamma \vdash O : \sigma \qquad \Gamma \vdash a : \sigma}{\Gamma \vdash \mathsf{copy}(O)^a : \sigma} \quad (Structure\text{-}Copy)$$

General Rules for Code, Evaluation Contexts and Object-Structures
In the following, we let A be either M, or U, or O, or a.

$$\frac{\Gamma \vdash A : \sigma \qquad \Gamma \vdash \sigma \xrightarrow{type} \tau}{\Gamma \vdash A : \tau} \quad (Red\text{-}Left)$$

$$\frac{\Gamma \vdash A : \sigma \qquad \Gamma \vdash \tau \xrightarrow{type} \sigma}{\Gamma \vdash A : \tau} \quad (Red\text{-}Right)$$

$$\frac{\Gamma \vdash A : \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle \qquad \Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle : T}{\Gamma \vdash A : \mathtt{pro}\, t.\langle\!\langle R \lhd R', m : \sigma \rangle\!\rangle} \quad (Pre\text{-}Extend)$$

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