A Linear Logic Calculus of Objects

Michele Bugliesi
Dipartimento di Matematica, Università di Padova
Via Belzoni 7, I-35131 Padova, Italy
e-mail: `michele@math.unipd.it`

Giorgio Delzanno
DISI, Università di Genova
Via Dodecaneso, 35, I-16146 Genova, Italy
e-mail: `giorgio@disi.unige.it`

Luigi Liquori
Dipartimento di Informatica, Università di Torino
C.so Svizzera 185, I-10149 Torino, Italy
e-mail: `liquori@di.unito.it`

Maurizio Martelli
DISI, Università di Genova
Via Dodecaneso, 35, I-16146 Genova, Italy
e-mail: `martelli@disi.unige.it`

Abstract

This paper presents a linear logic programming language, called $O_{\multimap}$, that gives a complete account of an object-oriented calculus with inheritance and override. This language is best understood as a logical counterpart the object and record extensions of functional programming that have recently been proposed in the literature. From these proposals, $O_{\multimap}$ inherits the representation of objects as composite data structures, with attribute and method fields, as well as their interpretation as first-class values. $O_{\multimap}$ also gives a direct logical modeling of the self-application semantics of method invocation that justifies the view of objects as elements of recursive types. As such, the design of $O_{\multimap}$ appears interesting, in perspective, as a basis for developing flexible and powerful type systems for logical object-based languages.

1 Introduction

Object-oriented languages can be classified as either *class-based* or *object-based* according to the underlying object-oriented model. In class-based languages objects are created by instantiating their class. In object-based languages, instead, objects are derived dynamically by modifying or extending existing objects, used as *prototypes*. A derived object may then respond to messages directly, if it contains corresponding methods, or "pass them back", i.e. *delegate* them, to the prototype.

Originated with the seminal work on *Self* [9], the object-based model has subsequently been studied in several papers, among which the record calculus of [3], and then in recent papers on functional object-oriented languages [1, 5]. These latter proposals are centered around two main design choices: (i) objects are first-class values consisting of the set of their methods (attributes are viewed as constant methods), and (ii) methods are functions with a distinguished parameter denoting the host object. The resulting languages are rather effective in that they encompass a very simple and elegant modeling of the object-oriented notion of *self* directly by lambda abstraction, and they provide a well-suited basis for studying a typed foundation for object-oriented languages and systems.

Drawing on these proposals, in this paper we present a linear logic programming language, called $O_{\multimap}$, that gives a complete account of an object-based calculus with inheritance and override.

The logical foundations of $O_{\multimap}$ lie in a fragment of intuitionistic higher-order Linear Logic consisting of the connectives $\multimap$, $\Rightarrow$, $\&$, and $\forall$. The higher-order nature of the syntax allows a direct encoding of objects as first-class values with attribute and method fields: the linear connectives, on the other hand, allow methods to be selected and applied according to the self-application semantics of method invocation. $O_{\multimap}$ also provides primitives for method addition and redefinition, peculiar to the companion functional calculi, and it encompasses the same mechanism of method and attribute inheritance by delegation. Method redefinition relies on a functional form of update whereby overriding a method (or attribute) is realized by computing a modified copy of the object containing that method.

The design of $O_{\multimap}$ shares ideas with previous proposals of linear object-oriented languages [2, 4, 7], but the resulting language exhibits several novel features. As in in these languages, computation in $O_{\multimap}$ is a process of logical deduction in which, however, objects are computed values rather than consumable resources. Accordingly, the result of a program is a set of answer substitutions as it is customary in logic programming. In fact, $O_{\multimap}$ can readily be extended to allow writing program clauses (actually meta-programs with respect to objects) in the usual logic programming style, using predicates which have objects as data.

A further important difference with respect to previous work, is in the choice of the underlying object-oriented model: to our knowledge, this is the first attempt to give a direct logical semantics to the object and record calculi that subscribe to the object-based view of object-orientation.

The rest of the paper is organized as follows. In Section 2 we introduce the logic at the basis of our calculus, and we analyze its proof theoretic properties. In Section 3 we describe how $O_{\multimap}$ can be expressed in this logic, and we exemplify it on a few programming examples. In Section 4 we present an interpreter for $O_{\multimap}$ based on unification, showing that unification is decidable for $O_{\multimap}$ in spite of its higher-order syntax. We conclude in Section 5 discussing related research, and addressing issues relative to the design of a type system for our language.

2 A Logic for a Calculus of Objects

We start with a presentation of a fragment of Intuitionistic Linear Logic that will serve as the logical language underlying the definition of $O_{\multimap}$.

2.1 A Fragment of Intuitionistic Linear Logic

Terms in this language are defined over a denumerable set of variables $\mathcal{V}$ and a signature Σ partitioned into a set of *non-logical* constants, or *parameters*, and a set of *logical constants*, with the latter comprising the symbols $\&, \Rightarrow$, $\multimap$, and the infinite list of quantifiers $\forall_\tau$ for every type τ ($\Rightarrow$ denotes intuitionistic implication, defined as $A \Rightarrow B \equiv (!A) \multimap B$, and indicated by :− when used in the reverse direction). Types, ranged over by τ, are defined by the following productions: $\tau ::= \phi \mid b \mid \tau \to \tau \mid \tau \times \tau$ with ϕ denoting the type of formulas, b any base type, $\tau \to \tau$ function types and $\tau \times \tau$ product types. We assume that every type is non-empty, and that types for the logical constants are assigned as follows: $\&$, $\multimap$ and $\Rightarrow$ have type $\phi \times \phi \to \phi$, whereas $\forall_\tau$ has type $\tau \times \phi \to \phi$ for every type τ.

Terms. The set T_Σ of terms over $\mathcal{V}$ and Σ is the smallest set that satisfies the following conditions:

1. every variable $\mathrm{X} \in \mathcal{V}$ and symbol $h{:}\tau \in \Sigma$ is a term in T_Σ;
2. if $t_1{:}\tau_1, \ldots, t_n{:}\tau_n$ are terms in T_Σ, and $h{:}\tau_1 \times \cdots \times \tau_n \to \tau_{n+1}$ is a parameter from Σ, then $(h\ t_1 \ldots t_n)$ is a term in T_Σ. If $\tau_{n+1}{=}\phi$, then $(h\ t_1 \ldots t_n)$ is an *atomic formula*.
3. If A is an atomic formula or a variable of type ϕ, and A^r is a *rigid* atomic formula (an atomic formula that is not a variable), then any term in the set generated by the following productions is a term in T_Σ:

$$\begin{array}{lcl} E & ::= & A^r \mid G \Rightarrow A^r \mid E \,\&\, E \mid \forall_\tau \mathrm{X}.E \\ G & ::= & A \mid E \multimap A^r \mid G \,\&\, G \mid \forall_\tau \mathrm{X}.G. \end{array}$$

Intuitively, terms generated by the non-terminal symbol E denote program formulas, whereas terms generated by the non-terminal G denote goal formulas. Atomic E-formulas A^r represent *unit* clauses. Existential quantification for goal formulas is omitted as it can be defined in terms of universal quantification within E-formulas, and replaced by the use of “logical” variables within top-level goal formulas. Note that our encoding of quantified formulas as terms differs from the more conventional use of terms with types as in Church’s simple theory of types adopted elsewhere to define higher-order languages (cfr. [4, 8]). Since we don't need abstraction and application for the scope of the present paper, the choice of this simplified encoding is possible, and appealing in that (i) it reduces the notion of equality over Σ-terms to α-convertibility, (ii) it makes unification of Σ-terms a relatively easy matter, and (iii) it guarantees that closed E- and G-formulas remain E- and G-formulas in the language under arbitrary variable instantiation with closed Σ-terms. A complete set of proof rules for the set of E and G formulas can be defined as in Figure 2.1.

$$\frac{}{\Sigma:\Gamma;A \longrightarrow A}\;(identity) \qquad \frac{\Sigma:\Gamma,B;\Delta,B \longrightarrow C}{\Sigma:\Gamma,B;\Delta \longrightarrow C}\;(absorb)$$

$$\frac{\Sigma:\Gamma;\Delta,B_i \longrightarrow G}{\Sigma:\Gamma;\Delta,B_1 \,\&\, B_2 \longrightarrow C}\;(\&\,L) \qquad \frac{\Sigma:\Gamma;\Delta \longrightarrow C_1 \quad \Sigma:\Gamma;\Delta \longrightarrow C_2}{\Sigma:\Gamma;\Delta \longrightarrow C_1 \,\&\, C_2}\;(\&\,R)$$

$$\frac{\Sigma:\Gamma;\emptyset \longrightarrow B \quad \Sigma:\Gamma;\Delta,C \longrightarrow D}{\Sigma:\Gamma;\Delta,B \Rightarrow C \longrightarrow D}\;(\Rightarrow L) \qquad \frac{\Sigma:\Gamma\,;\,\Delta,\,B \longrightarrow C}{\Sigma:\Gamma\,;\,\Delta \longrightarrow B \multimap C}\;(\multimap R)$$

$$\frac{\Sigma:\Gamma;\Delta,[t/\mathrm{x}]\,B \longrightarrow C}{\Sigma:\Gamma;\Delta,\forall_\tau \mathrm{x}.B \longrightarrow C}\;(\forall L) \qquad \frac{\Sigma':\Gamma\,;\,\Delta \longrightarrow [c/\mathrm{x}]\,C}{\Sigma:\Gamma\,;\,\Delta \longrightarrow \forall_\tau \mathrm{x}.C}\;(\forall R)$$

Figure 2.1: The proof system $\mathcal{L}_0$. In $(\forall L)$, t is a term of type τ; in $(\forall R)$, $\Sigma' = c{:}\tau, \Sigma$, with $c{:}\tau$ not in Σ.

The proof rules of system $\mathcal{L}_0$ are derived by restricting the proof system of the *Lolli* language of [6] to our set of formulas. From [6], we also borrow the interpretation of sequents: in $\Sigma{:}\Gamma;\Delta \longrightarrow G$, Γ and Δ are, respectively, a set and a multi-set of E-formulas (the *unbounded* and *bounded* contexts), whereas G is a G-formula, all defined over the signature Σ.

Completeness of uniform (i.e. goal directed) $\mathcal{L}_0$ proofs for our language of E and G formulas (with respect to provability in Linear Logic) derives directly from the results proved in [6]. Owing to the syntactic structure of our formulas, we may in fact restrict to simpler proofs without losing completeness.

Say that an $\mathcal{L}_0$ proof is *simple* if and only if is uniform and for every sequent $\Sigma{:}\Gamma;\Delta \longrightarrow G$ occurring in the proof, Δ is either the singleton or the empty multiset. Then, we have the following:

Proposition 2.1 *Let Γ and Δ be multisets of E-formulas and G a G-formula. If the sequent $\Sigma{:}\Gamma;\Delta \longrightarrow G$ has an $\mathcal{L}_0$-proof then it has a simple proof.*

PROOF. *First observe (i) that every $\mathcal{L}_0$-proof ends up with instances of the (identity) axiom in their leaves, and (ii) that (identity) has a unique formula in the bounded part of the context of its lower sequent. Then, the claim follows by the completeness of $\mathcal{L}_0$ uniform proofs, and noting that uses of the (absorb) rule in an $\mathcal{L}_0$-proof are limited to sequents that have an empty bounded context.* □

Based on this property, we may further specialize the set of proof rules as shown in Figure 2.2.

$$\frac{\Sigma : \Gamma, E \xrightarrow{E} A}{\Sigma : \Gamma, E \longrightarrow A}\ (absorb) \qquad \frac{\Sigma : \Gamma \longrightarrow G}{\Sigma : \Gamma \xrightarrow{E} A}\ (bc) \qquad \frac{}{\Sigma : \Gamma \xrightarrow{A'} A}\ (identity)$$

$$\frac{\Sigma : \Gamma \xrightarrow{E} A}{\Sigma : \Gamma \longrightarrow E \multimap A}\ (\multimap R) \quad \frac{\Sigma' : \Gamma \longrightarrow [c/\mathrm{x}]\, G}{\Sigma : \Gamma \longrightarrow \forall_\tau \mathrm{x}.G}\ (\forall R) \quad \frac{\Sigma : \Gamma \longrightarrow G_1 \quad \Sigma : \Gamma \longrightarrow G_2}{\Sigma : \Gamma \longrightarrow G_1 \,\&\, G_2}\ (\& R)$$

Figure 2.2: The proof system $\mathcal{L}$. In the (bc) rule, A is an atomic formula, and $G \Rightarrow A \in [E]_\Sigma$; in the $(identity)$ rule, A is atomic and $A \in [A']_\Sigma$.

Note that sequents have now the two simplified forms $\Sigma{:}\Gamma \xrightarrow{E} G$ and $\Sigma{:}\Gamma \longrightarrow G$, denoting, respectively, a sequent with E as the unique formula in the bounded part of the context, and a sequent with an empty bounded context. All the formulas occurring in the sequents are assumed to be closed, and the set $[E]_\Sigma$ used in the backchaining rule is the minimal set of formulas that satisfies the following conditions:

$E \in [E]_\Sigma$;
if $E_1 \,\&\, E_2 \in [E]_\Sigma$, then $E_1 \in [E]_\Sigma$ and $E_2 \in [E]_\Sigma$;
if $\forall \mathrm{x}_\tau.E \in [E]_\Sigma$, then $[t/\mathrm{x}]\, E \in [E]_\Sigma$ for every closed Σ-term $t : \tau$.

Completeness of $\mathcal{L}$-proofs with respect to simple $\mathcal{L}_0$-proofs immediately follows since a $\mathcal{L}$-proof is just a simple $\mathcal{L}_0$-proof where sequences of left-rules are collapsed into (bc) steps.

For future reference, we note here that the constraint on the bounded part of the context in $\mathcal{L}$-sequents is central for the encoding of our object calculus in the language of E and G formulas: as we shall see in Section 3, it is precisely this constraint that guarantees that no conflict may arise among methods of different objects upon method invocation. Before giving further details, we next briefly introduce the main features of the object-calculi we wish to represent, taking the *Lambda Calculus of Objects* [5] as representative.

2.2 A Functional Calculus of Objects

The *Lambda Calculus of Objects*, [5], is an untyped lambda calculus enriched with object forms and three primitive operations on objects: *method addition*, to define new methods, *method override*, to redefine methods, and *method call*, to send a message to (i.e., invoke a method on) an object.

Extensionally, an object can be viewed as the set of its methods, and these may be invoked by means of message sends. When an object containing an m method is sent the message m, the result is obtained by applying the body of m to the object itself. The intuitive semantics of method evaluation can then be expressed as follows:

$$\langle \ldots\ m = e\ \ldots \rangle \Leftarrow m \quad \xrightarrow{eval} \quad e\, \langle \ldots\ m = e\ \ldots \rangle,$$

where $\langle m_1 = e_1; \ldots; m_n = e_n\rangle$ denotes a record consisting of the definitions of the methods $m_1, \ldots, m_n$, and $\Leftarrow$ denotes method invocation.

The method body e must be a function, and the first actual parameter of e will always be the receiver of the message associated to the method itself: in this way, the special symbol *self* of object-oriented languages is modeled directly by means of lambda abstraction and application. A further distinguishing feature of this calculus is that objects have no explicit identifiers attached to them. This makes it easy to create new objects without having to generate new identifiers: new objects are created by simply modifying the structure of existing objects by adding new methods or redefining already defined methods. As an example, consider the expression

$$\mathtt{pt} \equiv \langle \mathtt{x} = \lambda\mathtt{self}.3;\ \mathtt{move} = \lambda\mathtt{self}.\lambda\mathtt{d}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{d}\rangle\rangle,$$

representing a one dimensional point with a `move` method, where $\leftarrow$ denotes method override. Evaluating the message `move` with argument `2` yields the following sequence of steps:

$$\begin{array}{rcl}
\mathtt{pt} \Leftarrow \mathtt{move}\,2 & \xrightarrow{eval} & (\lambda\mathtt{self}.\lambda\mathtt{d}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{d}\rangle)\ \mathtt{pt}\,2 \\
& \xrightarrow{eval} & \langle \mathtt{pt} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{pt} \Leftarrow \mathtt{x}) + 2\rangle \\
& \xrightarrow{eval} & \langle \mathtt{pt} \leftarrow \mathtt{x} = \lambda\mathtt{s}.3 + 2\rangle \\
& \xrightarrow{eval} & \langle \mathtt{pt} \leftarrow \mathtt{x} = \lambda\mathtt{s}.5\rangle.
\end{array}$$

The last expression is equivalent to the new object

$$\langle \mathtt{x} = \lambda\mathtt{self}.5;\ \mathtt{move} = \lambda\mathtt{self}.\lambda\mathtt{d}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{d}\rangle,$$

that results from replacing the body of the `x` method in `pt` as specified by the override.

3 A Logic Calculus of Objects

The key idea in defining $O_{\multimap}$ is to distinguish, within the object calculus we have just outlined, the aspects relative to the representation of objects and methods, from those relative to the semantics of the primitive operations of extension, override and method invocation. Intuitively, this distinction is achieved by representing objects a complex terms, with method fields, occurring within G formulas, while encoding the $\xrightarrow{eval}$ relation by means of E formulas placed in the unbounded context of a sequent. As in the language *F&O* of [4], the higher order nature of the syntax allows us to encode both these aspects uniformly in the language.

Objects, Methods and Atomic Formulas. Objects are represented as particular terms: syntactically, an object is either the *empty object* $\langle\rangle$, or a term of the form:

$$obj \quad ::= \quad \langle m_1 :: D_1\ ;\ \ldots\ ;\ m_k :: D_k\rangle.$$

For every $i = 1, \ldots, k$, m_i is a method name acting as the label of the method definition D_i. Objects occur as subterms of atomic formulas which are statements of the form:

$$A \quad ::= \quad obj.\, m\; args \;\mapsto\; res.$$

Here *obj* is an object, m a method name, $args = [\, a_1\, a_2\, \ldots\, a_n \,]$ is the tuple of input arguments of m, and res is the result of m. Intuitively, the above formula states that the result of invoking m at *obj* with input arguments $args$ yields res as a result. The notation $m\; args$ indicates that both m and $args$ are arguments of the operator $\bullet.\;\bullet\;\bullet\!\mapsto\!\bullet$ which is the constructor of atomic formulas. Methods of an object are defined by formulas of the form:

$$\forall \overline{\mathrm{X}}.\; (\mathrm{SELF}.\, m\, \mathrm{T} \mapsto \mathrm{R}) :\!-\; G \qquad \text{or} \qquad \forall \overline{\mathrm{X}}.\; (\mathrm{SELF}.\, m\, \mathrm{T} \mapsto \mathrm{R}).$$

Here, $\overline{\mathrm{X}}$ is a sublist of all the variables occurring in the definition, (i.e. variables may occur free in a method definition), SELF is a term representing the host object for the method, and G is the body of the method. Method bodies are G-formulas built over the following set of atomic formulas:

$O.\,\mathtt{send}\;[\,m\; args\,] \mapsto res$ send message m to object O, returning res;
$O.\,\mathtt{ext}\;\;[\,m\, D\,] \mapsto O_1$ extend O with a D definition for m, returning O_1;
$O.\,\mathtt{over}\;\;[\,m\, D\,] \mapsto O_1$ replace O's definition for m with D, returning O_1.

The symbols `send`, `ext` and `over` are symbols corresponding to the three object-related primitives defined defined by means of the E-formulas that we discuss next. To ease the notation, in the rest of this section we adopt the following standard conventions: top-level typed quantifiers are omitted, and quantified variables are denoted by UPPER-CASE symbols; conjoined E formulas are written as separate clauses, whereas G-formulas are separated by commas; finally, square brackets are omitted from 1-component tuples.

Messages and Method Invocation. Sending a message to an object consists of two separate steps, namely (i) select the definition of the corresponding method on the receiver of the message, and (ii) apply the definition to the receiver itself and the input arguments specified in the message. To account for these operations, we introduce the following E-definition of the `send` and `select` primitives (the abuse of notation $\mathrm{M} \neq \mathrm{N}$ below is only apparent as this test could, in principle, be expressed as a message to an object with an `eq` method):

$$\mathrm{O}.\,\mathtt{send}\;[\,\mathrm{M}\,\mathrm{T}\,] \mapsto \mathrm{R} :\!-\; \mathrm{O}.\,\mathtt{select}\;\mathrm{M} \mapsto \mathrm{D},\; (\mathrm{D} \multimap \mathrm{O}.\,\mathrm{M}\,\mathrm{T} \mapsto \mathrm{R}).$$

$$\langle \mathrm{M} :: \mathrm{D}\; ;\; \mathrm{O} \rangle.\,\mathtt{select}\;\mathrm{M} \mapsto \mathrm{D}.$$
$$\langle \mathrm{N} :: \_\; ;\; \mathrm{O} \rangle.\,\mathtt{select}\;\mathrm{M} \mapsto \mathrm{D} :\!-\; \mathrm{M} \neq \mathrm{N},\; \mathrm{O}.\,\mathtt{select}\;\mathrm{M} \mapsto \mathrm{D}.$$

The effect of evaluating a `send` primitive can be illustrated as follows. Let Γ be a set of E-formulas including the above E-definitions of `send` and `select` , and let O be an object containing an m method defined by D. Then, proving the goal formula $O.\,\texttt{send}\ [\ m\,T\ ] \mapsto R$ amounts to the following derivation in the system $\mathcal{L}$:

$$
\dfrac{\dfrac{\dfrac{\vdots}{\Sigma : \Gamma \xrightarrow{E} O.\texttt{select}\ m \mapsto D}}{\Sigma : \Gamma \longrightarrow O.\texttt{select}\ m \mapsto D}\,(absorb) \qquad \dfrac{\dfrac{\dfrac{\cdots}{\Sigma : \Gamma \xrightarrow{D} O.\,m\,T \mapsto R}\,(bc)}{\Sigma : \Gamma \longrightarrow D \multimap O.\,m\,T \mapsto R}\,(\multimap R)}{}}{\dfrac{\Sigma : \Gamma \longrightarrow O.\texttt{select}\ m \mapsto D,\ (D \multimap O.\,m\,T \mapsto R)}{\Sigma : \Gamma \longrightarrow O.\texttt{send}\ [\,m\,T\,] \mapsto R}\,(absorb + bc)}\,(\&R)
$$

The upper-right application of the (bc) rule in the right branch of the proof selects (non-deterministically) one formula in $[D]_\Sigma$, which is used to reduce the formula $O.\,m\,T \mapsto R$: matching the term that encodes the host object in the head of D with O captures desired effect of "self-applying" the m method to the receiver of the message. Note further that methods are made available as bounded resources for backchaining only when explicitly requested by a message. Since objects are non identified, this ability to activate methods selectively upon evaluating a message is crucial to avoid conflicts among methods of different objects.

Method Extension and Override The remaining primitive operations on objects are method addition and overriding defined by the following E-formulas, whose intended meaning should be self-explanatory.

$\langle\rangle.\,\texttt{ext}\ [\,\mathrm{M\ D}\,] \mapsto \langle \mathrm{M} :: \mathrm{D} \rangle.$
$\langle \mathrm{N} :: \_\ ;\ \mathrm{O} \rangle.\,\texttt{ext}\ [\,\mathrm{M\ D}\,] \mapsto \langle \mathrm{N} :: \_\ ;\ \mathrm{O}_1 \rangle \ :-\ \mathrm{M} \neq \mathrm{N},\ \mathrm{O}.\,\texttt{ext}\ [\,\mathrm{M\ D}\,] \mapsto \mathrm{O}_1.$

$\langle \mathrm{M} :: \_\ ;\ \mathrm{O} \rangle.\,\texttt{over}\ [\,\mathrm{M\ D}\,] \mapsto \langle \mathrm{M} :: \mathrm{D}\ ;\ \mathrm{O} \rangle.$
$\langle \mathrm{N} :: \_\ ;\ \mathrm{O} \rangle.\,\texttt{over}\ [\,\mathrm{M\ D}\,] \mapsto \langle \mathrm{N} :: \_\ ;\ \mathrm{O}_1 \rangle \ :-\ \mathrm{M} \neq \mathrm{N},\ \mathrm{O}.\,\texttt{over}\ [\,\mathrm{M\ D}\,] \mapsto \mathrm{O}_1.$

3.1 Examples of Objects and Methods

We next present a few short examples, borrowed from the *Lambda Calculus of Objects* of [5], that help clarify the computational behavior of $\mathcal{O}_\multimap$. In presenting the examples, we informally appeal to an operational semantics that uses unification to compute bindings for the existentially quantified variables of a query. A formal definition of an interpreter, based on unification, is given in Section 4.

One-dimensional point with a move method. First consider the definition of the point object with the constant x-coordinate and the move method of Section 2.2. This object can be represented as the pt object:

$$\texttt{pt} \ \stackrel{def}{=}\ \langle \texttt{x} :: \mathrm{SELF}.\,\texttt{x} \mapsto 3.\ ;\ \texttt{move} :: D_{\texttt{move}} \rangle,$$

where $D_{\texttt{move}}$ is the following definition:

$\mathrm{SELF}.\,\texttt{move}\ \mathrm{Z} \mapsto \mathrm{O} :-$
$\qquad \mathrm{SELF}.\,\texttt{send}\ \texttt{x} \mapsto \mathrm{V},\ \mathrm{SELF}.\,\texttt{over}\ [\,\texttt{x}\ (\forall \mathrm{S}.\,\mathrm{S}.\,\texttt{x} \mapsto \mathrm{V}{+}\mathrm{Z}.)\,] \mapsto \mathrm{O}.$

Consider then the G-formula $\texttt{pt.send}\ [\texttt{move}\ \ 2] \mapsto \textsc{p}$, where $\textsc{p}$ is a free (i.e., existentially quantified) variable: evaluating this formula, leads to evaluating the conjoined formula:

$$\texttt{pt.send}\ \texttt{x} \mapsto \textsc{v}, \texttt{pt.over}\ [\,\texttt{x}\ \ (\forall \textsc{s}.\ \textsc{s}.\texttt{x} \mapsto \textsc{v}{+}2.)\,] \mapsto \textsc{p}.$$

which, in turn, binds $\textsc{v}$ to the value 3 and then $\textsc{p}$ to the new object $\langle \texttt{x} :: \forall\textsc{self}.\ \textsc{self}.\texttt{x} \mapsto 3{+}2.; \texttt{move} :: D_{\texttt{move}}\rangle$.

Inherit `move` from points to colored-points. The next example illustrates how methods are inherited from prototypes to derived objects. Consider again the `pt` object of the previous example, and let $D_{\texttt{col}}$ be the formula $(\forall\textsc{self}.\ \textsc{self}.\texttt{col} \mapsto \texttt{blue}.)$. Now, consider the following composite query:

$$\texttt{pt.ext}\ [\,\texttt{col}\ D_{\texttt{col}}\,] \mapsto \textsc{p}, \textsc{p}.\texttt{send}\ [\,\texttt{move}\ \ 2\,] \mapsto \textsc{q}, \textsc{q}.\texttt{send}\ \texttt{col} \mapsto \textsc{c}.$$

In the left-most G formula, `pt` is used as a "prototypical" point from which we derive a colored-point with a new constant method `col`. The query illustrates how the new object inherits the `move` method (indeed, also the `x` method) from its prototype. In fact, evaluating the method extension on `pt` yields the following binding:

$$\textsc{p} = \langle \texttt{x} :: \textsc{s}.\texttt{x} \mapsto 3.\ ;\ \texttt{move} :: D_{\texttt{move}}\ ;\ \texttt{col} :: D_{\texttt{col}}\ .\rangle.$$

Then, proceeding from left to right, $\textsc{p}.\texttt{send}\ [\,\texttt{move}\ \ 2\,] \mapsto \textsc{q}$ yields the binding:

$$\textsc{q} = \langle \texttt{x} :: \textsc{s}.\texttt{x} \mapsto 3+2.\ ;\ \texttt{move} :: D_{\texttt{move}}\ ;\ \texttt{col} :: D_{\texttt{col}}\ .\rangle$$

and, finally, the last message yields $\textsc{c} = \texttt{blue}$ as expected.

Object numerals. As a further example, we show how natural numbers can be represented in $O_{\multimap}$. Following [1], we define object-numerals as objects that respond to the methods `is_zero` (test for zero), `pred` (predecessor) and `succ` (successor) and behave like natural numbers. In fact, we need only to define the numeral zero as the "prototypical" number, and let all other numerals be generated by repeated applications of the `succ` method.

$$\begin{array}{lll}
\langle \texttt{is_zero} & :: & \textsc{self}.\texttt{is_zero} \mapsto \texttt{true}.; \\
\ \ \texttt{pred} & :: & \textsc{self}.\texttt{pred} \mapsto \textsc{self}.; \\
\ \ \texttt{succ} & :: & \textsc{self}.\texttt{succ} \mapsto \textsc{o}\ :- \\
 & & \qquad \textsc{self}.\texttt{over}\ [\ \texttt{is_zero}\ \ (\forall\textsc{s}.\ \textsc{s}.\texttt{is_zero} \mapsto \texttt{false}.)\ ] \mapsto \textsc{p}, \\
 & & \qquad \textsc{p}.\texttt{over}\ [\ \texttt{pred}\ \ (\forall\textsc{s}.\ \textsc{s}.\texttt{pred} \mapsto \textsc{self}.)\ ] \mapsto \textsc{o}.\ \rangle.
\end{array}$$

One easily verifies, with a few tests, that the operational semantics of natural numbers is well represented. In particular, note that the body of `succ` consists of two cascaded updates for SELF: when invoked on any object-numeral, `succ` updates the `is_zero` method to answer `false`, and updates the `pred` method to return the current value of $self$ when `succ` is invoked.

3.2 Mixing Object-Oriented and Logic Programming in $O_{\multimap}$

So far we have only exemplified the object-oriented features of $O_{\multimap}$. However, it is also possible to write $O_{\multimap}$ programs that use objects as data structures. We only need to be more liberal in the syntax, and allow atomic formulas to also have the conventional predicative structure $p(a_1, \ldots, a_n)$. Clauses built over predicative atomic formulas may then be placed in the unbounded context of a sequent, together with the object-related E-definitions, to form logic programs.

To exemplify, consider again the `pt` point-object of the previous section: we can now use the following simple program to create colored-points:

$$\texttt{make\_cpt}(\textsc{p}, \textsc{col}, \textsc{cp}) \quad :- \quad \textsc{p}.\,\texttt{ext}\,[\,\texttt{col}\;\; (\forall \textsc{s}.\,\textsc{s}.\,\texttt{col} \mapsto \textsc{col}.)\,] \mapsto \textsc{cp}.$$

Similarly, the following clause creates two-dimensional points from points:

$$\begin{array}{lll} \texttt{make\_2dpt}(\textsc{p}, \textsc{y}, \textsc{p2d}) & :- & \textsc{p}.\,\texttt{ext}\,[\,\texttt{y}\;\; (\forall \textsc{s}.\,\textsc{s}.\,\texttt{y} \mapsto \textsc{y}.)\,] \mapsto \textsc{yp}, \\ & & \textsc{yp}.\,\texttt{over}\,[\,\texttt{move}\;\; D_{\texttt{move}}\,] \mapsto \textsc{p2d}. \end{array}$$

where $D_{\texttt{move}}$ is the following definition:

$$\begin{array}{lll} \textsc{self}.\,\texttt{move}\,[\,\textsc{z}\;\textsc{t}\,] \mapsto \textsc{no} & :- & \textsc{self}.\,\texttt{send}\;\texttt{x} \mapsto \textsc{v},\ \textsc{self}.\,\texttt{send}\;\texttt{y} \mapsto \textsc{w}, \\ & & \textsc{self}.\,\texttt{over}\,[\,\texttt{x}\;\; (\forall \textsc{s}.\,\textsc{s}.\,\texttt{x} \mapsto \textsc{v}{+}\textsc{z}.)\,] \mapsto \textsc{o}, \\ & & \textsc{o}.\,\texttt{over}\,[\,\texttt{y}\;\; (\forall \textsc{s}.\,\textsc{s}.\,\texttt{y} \mapsto \textsc{w}{+}\textsc{t}.)\,] \mapsto \textsc{no}. \end{array}$$

Now consider the following program to move a point in two-dimensions:

$$\texttt{move\_pt}(\textsc{p}, \textsc{np}, \textsc{x}, \textsc{y}) \quad :- \quad \textsc{p}.\,\texttt{send}\,[\,\texttt{move}\;\; [\,\textsc{x}\;\textsc{y}\,]\,] \mapsto \textsc{np}.$$

Then, it is possible to formulate queries as the following:

$$\texttt{make\_2dpt}(\texttt{pt}, \texttt{0}, \textsc{p2d}),\ \texttt{make\_cpt}(\textsc{p2d}, \texttt{blue}, \textsc{cp}),\ \texttt{move\_pt}(\textsc{cp},\ \textsc{np}, 3, 4).$$

where, if `pt` is the one-dimensional point at position 3 on the line, the result is to bind NP to a two-dimensional colored point at position (6,4) on the plane.

4 An Interpreter for $O_{\multimap}$

In describing the interpreter for $O_{\multimap}$ we follow the paradigmatic approach of [8] for interpreting the language L_λ. As in that case, interpretation is described as a theorem prover that mixes logical deduction with unification: unification is, in a sense, higher-order as our terms may encode quantified formulas, but the absence of abstractions and β-redexes from our terms makes it a much easier matter than in L_λ.

4.1 State Formulas and Substitutions

Following [8], we formalize the state of the interpreter using a meta-logic that comprises the constants $\wedge$, $\top$, $\perp$, and the quantifiers $\forall_\tau$ and $\exists_\tau$ (albeit identical, $\forall_\tau$ should not be confused with the quantifier from the object-level logic). Meta-level atomic propositions are called *judgements*, and we identify four kinds of judgements: the two constants $\top$ and $\perp$, the equality judgement $t \overset{\tau}{=} s$, and the sequent judgement $\Gamma \xrightarrow{\Delta} G$. We then call a formula in the meta-logic a *state-formula* if and only if (i) all the free variables in the object-level formulas are bound by meta-level quantifiers, and (ii) all these (bound by meta-level quantifiers) variables have distinct names.

Given a state formula $\mathcal{S}$ and a substitution ϑ, we say that ϑ is an $\mathcal{S}$-substitution if the domain of ϑ does not contain any of the meta-level universally quantified variables, but does contain all of the meta-level existentially quantified variables. Also, if $\exists_\tau \mathrm{X}$ occurs in $\mathcal{S}$ and if Σ is the set of universally quantified variables in which $\exists_\tau \mathrm{X}$ is in the scope, then $\vartheta \mathrm{X}$ must be a Σ-term of type τ. We say that an $\mathcal{S}$-substitution ϑ satisfies $\mathcal{S}$ if (i) $\mathcal{S}$ does not contain $\perp$, (ii) for every equation $t \overset{\tau}{=} s$ in $\mathcal{S}$, $\vartheta t \ \equiv_\alpha \ \vartheta s$ (where $\equiv_\alpha$ denotes $\alpha-$convertibility), and (iii) for every sequent $\Gamma \xrightarrow{\Delta} G$ in $\mathcal{S}$, the sequent $\Sigma : \Gamma \xrightarrow{\vartheta(\Delta)} \vartheta G$ has a proof in $\mathcal{L}$, (where Σ is the set of typed universal variables in which this sequent is in the scope). Finally, we say that an $\mathcal{S}$-substitution is a *solution* to $\mathcal{S}$ if it satisfies $\mathcal{S}$. Interpretation and unification are described as a collection of labeled transitions $\mathcal{S} \overset{\vartheta}{\leadsto} \mathcal{S}'$ where $\mathcal{S}$ and $\mathcal{S}'$ are state formulas and ϑ is an $\mathcal{S}'$-substitution (to ease the notation, we will omit type from quantifiers and equations occurring in sequent and equality judgements.)

4.2 Unification

State transitions for unification are defined specializing the transitions of [8] according to the syntax of our terms. By virtue of their simplified structure, the only subtle point in unifying two terms concerns the instantiation of existentially quantified variables: no such variable may, in fact, be instantiated to a term containing an occurrence of a variable that is universally quantified in the scope of the existential quantifier. For instance, the state formula $\exists \mathrm{Y}.\forall \mathrm{X}.\,(p\ \mathrm{X} = p\ \mathrm{Y})$ has no solution, because solving it would lead to bind Y to the universal variable X occurring in the scope of the existential quantifier that binds Y. The following definition enforces this constraint.

Unification Transitions. Given the state formula $\mathcal{S}$, build a new state formula $\mathcal{S}'$ and a substitution ϑ by applying one of the following steps over one equation $s = t$ of $\mathcal{S}$.

- *Rigid–rigid.* Assume that $t = s$ is of the form $f\ t_1\ \ldots\ t_n = g\ s_1\ \ldots\ s_n$. If $f \neq g$ then replace the equation with $\perp$. Otherwise, replace $t = s$

with the conjunction $t_1 = s_1 \wedge \ldots \wedge t_n = s_n$ to form $\mathcal{S}'$ (if $n = 0$, then simply replace $t = s$ with $\top$). Set ϑ to be the empty substitution.

- *α-step.* Assume that $t = s$ is of the form $\forall \mathrm{X}.\, t = \forall \mathrm{Y}.\, s$. Then apply $\alpha-$conversion to reduce the equation to the new form $\forall \mathrm{Z}.\, t' = \forall \mathrm{Z}.\, s'$. Then, to form $\mathcal{S}'$, replace $t = s$ with $t' = s'$ in $\mathcal{S}$ and add $\forall \mathrm{Z}$ at the (right-)end of the quantifier prefix of $\mathcal{S}$. The substitution ϑ is empty.

- *var–term.* Assume that $t = s$ is of the form $\mathrm{V} = s$, where s is a term other than V, and V is existentially quantified in $\mathcal{S}$ (if the equation is of the form $t = \mathrm{V}$ consider $\mathrm{V} = t$ instead). If V is free in s, or s contains a free occurrence of a variable Z bound by a universal quantifier in the scope of $\exists \mathrm{V}$, then replace the equation with $\perp$ and let ϑ be the empty substitution. Otherwise set $\vartheta = [s/\mathrm{V}]$ and form $\mathcal{S}'$ by replacing the equation with $\top$, dropping $\exists \mathrm{V}$ from $\mathcal{S}$ and applying ϑ to all remaining judgements of $\mathcal{S}$. Otherwise if $t = s$ is of the form $\mathrm{V} = \mathrm{V}$, where V is existentially quantified in $\mathcal{S}$, set $\vartheta = [\mathrm{V}'/\mathrm{V}]$ and form $\mathcal{S}'$ by replacing the equation with $\top$, dropping $\exists \mathrm{V}$ from $\mathcal{S}$ adding $\exists \mathrm{V}'$ to $\mathcal{S}$ and applying ϑ to all remaining judgements of $\mathcal{S}$.

None of the remaining transitions of [8] are applicable here, since our terms are free of λ-abstractions; also, the *flexible-flexible* transition of [8] reduces here to equations between variables which can be instantiated only by parameters. Termination and correctness of the unification algorithm derives by the results of [8].

4.3 Interpretation

The core of the interpreter is given by six labelled transitions of the form $\mathcal{S} \overset{\varepsilon}{\rightsquigarrow} \mathcal{S}'$ where $\mathcal{S}$ and $\mathcal{S}'$ are state-formulas and the ε is the empty substitution. The following notation is useful in describing the interpreter. Let E be an E-formula, and let $\mathcal{S}$ be a state formula: then define $[\![E]\!]_{\mathcal{S}}$ as the smallest set of pairs such that:

- $\langle \Box, E \rangle \in [\![E]\!]_{\mathcal{S}}$, where $\Box$ is the empty list of quantifiers;

- if $\langle \mathcal{Q}, E_1 \,\&\, E_2 \rangle \in [\![E]\!]_{\mathcal{S}}$ for some list of quantifiers $\mathcal{Q}$, then $\langle \mathcal{Q}, E_1 \rangle \in [\![E]\!]_{\mathcal{S}}$ and $\langle \mathcal{Q}, E_2 \rangle \in [\![E]\!]_{\mathcal{S}}$;

- if $\langle \mathcal{Q}, \forall \mathrm{X}.\, E \rangle \in [\![E]\!]_{\mathcal{S}}$ then $\langle \mathcal{Q} \exists \mathrm{Y}, [\mathrm{Y}/\mathrm{X}]E \rangle \in [\![E]\!]_{\mathcal{S}}$ where Y is a new variable that is not bound in $\mathcal{S}$;

The use of the notation $[\![E]\!]_{\mathcal{S}}$ parallels the use of $[E]_{\Sigma}$ in the proof system $\mathcal{L}$. Intuitively, if $\langle \mathcal{Q}, G \Rightarrow A \rangle \in [\![E]\!]_{\mathcal{S}}$, then A may be proved from E for some substitution ϑ if ϑG can be proved in E.

$$
\begin{array}{llcl}
(\mathit{with}) & \mathcal{S}\,[\Gamma \longrightarrow G_1 \,\&\, G_2] & \overset{\varepsilon}{\leadsto} & \mathcal{S}\,[\Gamma \longrightarrow G_1 \ \wedge\ \Gamma \longrightarrow G_2]; \\
(\mathit{lolli}) & \mathcal{S}\,[\Gamma \longrightarrow D \multimap A] & \overset{\varepsilon}{\leadsto} & \mathcal{S}\,[\Gamma \xrightarrow{D} A]; \\
(\mathit{forall}) & \mathcal{S}\,[\Gamma \longrightarrow \forall \mathrm{X}.G] & \overset{\varepsilon}{\leadsto} & \mathcal{S}\,[\forall \mathrm{Y}.(\Gamma \longrightarrow [\mathrm{Y}/\mathrm{X}]G)],\ \mathit{with}\ \mathrm{Y}\ \mathit{new\ in}\ \mathcal{S}; \\
(\mathit{absorb}) & \mathcal{S}\,[\Gamma \longrightarrow A] & \overset{\varepsilon}{\leadsto} & \mathcal{S}\,[\Gamma \xrightarrow{E} A]; \\
(\mathit{bc}) & \mathcal{S}\,[\Gamma \xrightarrow{E} A] & \overset{\varepsilon}{\leadsto} & \mathcal{S}\,[\mathcal{Q}(A = A' \ \wedge\ \Gamma \longrightarrow G)],\ \mathit{with}\ \langle \mathcal{Q}, G \Rightarrow A' \rangle \in [\![E]\!]_{\mathcal{S}}. \\
(\mathit{identity}) & \mathcal{S}\,[\Gamma \xrightarrow{A'} A] & \overset{\varepsilon}{\leadsto} & \mathcal{S}\,[\mathcal{Q}(A = A')],\ \mathit{with}\ \langle \mathcal{Q}, A \rangle \in [A']_{\mathcal{S}}.
\end{array}
$$

Figure 4.1: $\mathcal{S}[\mathcal{Q}.s]$ denotes an occurrence of the judgement s in $\mathcal{S}$, where $\mathcal{Q}$ is the tail of the quantifiers list in $\mathcal{S}$.

Sequent Transitions. The set of sequent transitions, depicted in Figure 4.1, parallels the set of rules of the proof-system $\mathcal{L}$. The correctness of the interpreter is a consequence of the following result.

Lemma 4.1 *If the interpreter makes a single sequent transition from $\mathcal{S}$ to $\mathcal{S}'$, and ϑ satisfies $\mathcal{S}'$, then ϑ satisfies $\mathcal{S}$.*

Since we allow quantification over propositional formulas, occurrences of free variables at the top level in a judgement could yield incompleteness in the interpreter. For instance, the judgement $\exists \mathrm{X}.(\Gamma \longrightarrow \mathrm{X})$, has a proof for an instance of X, e.g., an atomic formula in Γ, but the interpreter may not be able to proceed. This situation is similar to cases of *floundering* for derivations in logic programs with negation. In the following theorem we consider consider only *non-floundering*-transitions, i.e, transitions over states whose sequent judgements are free of occurrences of such variables.

Theorem 4.2 (Correctness of interpretation) *Let $\mathcal{S}$ be a state formula, $\mathcal{S}'$ be the state formula obtained by a sequence of non-floundered transitions of the interpreter, and let ς be the composition of the substitutions computed at each transition. Then $\mathcal{S}$ has a solution ϑ if an only if all the judgements contained in $\mathcal{S}'$ are $\top$, and there exists an $\mathcal{S}'$-substitution ϑ' such that $\varsigma \circ \vartheta' = \vartheta$ over the domain of $\mathcal{S}$.*

5 Conclusions

We have presented a linear logic programming language that gives a complete logical account of an object calculus with overriding and inheritance by delegation. We have defined a proof-system for the language, as well as an interpreter based on unification.

5.1 Related Work

As we already mentioned, the use of Linear Logic as a tool for modeling object-oriented programming in logic has already been addressed in the literature: among others, notable examples are the *LO* language of [2], the Linear Logic Language *F&O* [4] and the the *HACL* language of [7].

In LO, objects are represented as $\wp$–disjunctions of atomic formulas playing the role of attributes, while methods are specified as linear clauses that are used to rewrite (possibly modifying them) the attributes of objects. In $O_{\multimap}$, instead, objects are encoded as terms that encapsulate their methods as subterms. With this encoding, while retaining the form of method inheritance peculiar to [2], we also obtain a natural modeling of dynamic method redefinition, a functionality that could hardly be accounted for in [2].

The encoding of objects in $O_{\multimap}$ is inspired to the language $F\&O$ of [4], from which, however, our approach differs for the choice of both the computational model and the object model. $F\&O$ subscribes to the proofs-as-computations principle of Linear Logic which interprets sequents as encodings of the state of the computation and proofs as descriptions of the state evolution. Furthermore, $F\&O$ takes, essentially, a class-based approach where objects are created by instantiating a class and referenced to by means of the identifiers they are associated with at creation time. A similar approach is taken in the $HACL$ language of [7]. $HACL$ is a concurrent linear logic calculus which also adheres to the proofs-as-computations and formulas-as-processes principles of Linear Logic. Again, the object model is, essentially, a class-based model where objects are encoded as (λ-abstraction of) records that result as the fixed points of their associated class definitions.

On the other hand, $O_{\multimap}$ is a standard logic programming language, that uses unification to compute values returned as results in answer substitutions, and shared variables to capture the semantics of cascaded method invocations peculiar to the companion functional calculi. Furthermore, the underlying data model is an object-based model, where the recursive nature of objects is captured relying on the self-application semantics of method invocation rather than on the on the explicit use of fixed-point operators. Using this encoding, objects in $O_{\multimap}$ could be construed as elements of recursive types in a way similar to that described in [5]. The rest of this section briefly describes how this can be done.

5.2 An Object-Type System for $O_{\multimap}$

In the type system of [5], the types that are assigned to objects have the form $\texttt{object}\, t.\langle\!\langle m_1 : \tau_1, \ldots, m_k : \tau_k \rangle\!\rangle$, describing the methods (and corresponding types) available from objects with this type. Given an expression $\texttt{e} : \texttt{object}\, t.\langle\!\langle \ldots, m : \tau \rangle\!\rangle$, the result of the method invocation $\texttt{e} \Leftarrow m$ is a value of type $[\texttt{object}\, t.\langle\!\langle \ldots, m : \tau \rangle\!\rangle / t]\tau$, with the substitution of the object-type for t in τ reflecting the recursive nature of object-types.

Object representation in $O_{\multimap}$ was designed precisely to allow types of objects to be encoded exactly in this manner. For instance, assuming to replace the generic type o in the current encoding with the mode expressive class of object-types, we could, for instance, assign the point object $\texttt{pt}$ of section 3.1 the following type: $\texttt{object}\, t.\langle\!\langle \texttt{x} : int,\ \texttt{move} : int \rightarrow t \rangle\!\rangle$.

Allowing object-types in the set of legal types would thus be a smooth extension that does not break the type structure we have devised, provided that unification be instrumented to deal with the new types. This, and the other related problems relative to the extension with object-types represent subject of our future work.

Acknowledgements. The authors wish to thank the anonymous referees for their comments and suggestions.

References

[1] M. Abadi and L. Cardelli. A Theory of Primitive Objects. In *Proceedings of Theoretical Aspect of Computer Software*, volume 789 of *LNCS*, pages 296–320. Springer-Verlag, 1994.

[2] J. M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. *New Generation Computing*, 9:445–473, 1991.

[3] L. Cardelli and J.C. Mitchell. Operations on Records. *Mathematical Structures in Computer Sciences*, 1(1):3–48, 1991.

[4] G. Delzanno and M. Martelli. Objects in Forum. In *Proceedins of the International Logic Programming Symposium, Portland, Oregon*, pages 115–129. The MIT Press, 1995.

[5] 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.

[6] J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, pages 110(2):327–365, 1994.

[7] N. Kobayashi and A. Yonezawa. Type-Theoretic Foundations for Concurrent Object-Oriented Programming. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, 1994.

[8] D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[9] D. Ungar and R. B. Smith. Self: the Power of Simplicity. In *Proceedings of ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–241. ACM Press, 1987.