Object Calculi in Linear Logic

M. Bugliesi and G. Delzanno and L. Liquori and M. Martelli

To appear in *Journal of Logics and Computation*

Abstract

Several calculi of objects have been studied in the recent literature, that support the central features of object-based languages: messages, inheritance, dynamic dispatch, object update and object-extension. We show that a complete semantic account of these features may be given in a fragment of higher-order linear logic.

1 Introduction

Object-based calculi have recently emerged [1] as a foundational formalism for object-oriented languages and systems. Unlike traditional object-oriented languages, which are typically centered around classes, object-based calculi provide objects as the sole unit of abstraction, and support the core object-oriented principles of inheritance and encapsulation at the level of individual objects.

A number of papers in the literature have addressed the problem of finding adequate interpretations of object-based and class-based calculi into functional and logical formalisms [3, 4, 6, 15]. In this paper we study a novel characterization for object-based calculi into linear logic. Specifically, we isolate a fragment of higher-order linear logic – called $\mathcal{L}$ – that serves as specification language for a wide class of object-oriented primitives and constructs. Then we introduce an object-based language, called $\mathbf{Ob}_{\multimap}$, and show that $\mathcal{L}$ is powerful enough to encode this language.

The language $\mathbf{Ob}_{\multimap}$ (read "Ob-lolli") provides the core features of (untyped) object-based calculi, and comprises constructs for object formation and message-send, as well as primitives for method/field addition and override. As in its companion object calculi [1, 7], in $\mathbf{Ob}_{\multimap}$ objects are first-class values that collect both data (fields) and code (methods): the distinctive feature of $\mathbf{Ob}_{\multimap}$ is that methods residing within objects are represented as logical formulas. This representation of objects and methods is accounted for in the fragment $\mathcal{L}$ using (an algebraic restriction of) simply-typed lambda terms to encode quantifiers and quantified variables occurring in method definitions. Furthermore, the linear connectives of $\mathcal{L}$ allow an elegant rendering of the semantics of method invocation: specifically, the use of embedded (nested) linear implications allows methods to be characterized as resources that reside within objects, and are *consumed* right after having been selected for evaluation upon invocation.

There are two main contributions over previous work.

Firstly, our characterization of object calculi is new, even though it shares ideas with previous presentations of object-oriented languages [4, 6, 12, 15] in logic programming. In fact, we depart from the *proofs-as-computations* principle of linear logic, distinctive of previous proposals, and rely instead on a standard mechanism of resolution where the result of a computation is a set of answer substitutions binding variables to objects. An appealing consequence of this approach is that $\mathbf{Ob}_{\multimap}$ objects may directly be used as data structures in (standard) logic programs that define relations (predicates) over objects. It is this very ability to combine object-oriented and logic programming that motivates our choice of introducing $\mathbf{Ob}_{\multimap}$ as new language, rather than using $\mathcal{L}$ as the specification language for the object-calculi of [1, 7].

Secondly, we prove that the fragment $\mathcal{L}$, hence also $\mathbf{Ob}_{\multimap}$, has a complete proof procedure, by showing that uniform proofs in $\mathcal{L}$ are complete with respect to provability in higher order linear logic. The completeness proof is new, and technically interesting in view of the difficulty involved in the use of quantification for variables ranging over formulas. More importantly, the technique we use here generalizes to other fragments of higher-order linear and intuitionistic logics with nested implications. In fact, the coexistence of quantification over formulas and nested implication makes the fragment $\mathcal{L}$ (as well as its variations discussed in Section 7) very effective in specifying a wider and interesting class of programming language features and computations: specifically, computations where modules are first-class citizens and where, therefore, direct support is provided for higher-order modules and higher-order modular programming.

A preliminary version of this paper appeared in [5].

Plan of the paper. We organize the rest of the paper as follow. In Section 2 we introduce the linear logic fragment $\mathcal{L}$ which we use as the specification language for object calculi. In Section 3 we prove that uniform provability for $\mathcal{L}$ is complete. In Section 4 we present the syntax and the semantics of the object calculus $\mathbf{Ob}_{\multimap}$, and illustrate its use with a few examples. In Section 5 we show that $\mathbf{Ob}_{\multimap}$ can be encoded in $\mathcal{L}$; in Section 6 we then describe a prototype implementation and develop further programming examples. In Section 7 we discuss the generalization of the completeness proof to other logical fragments. We conclude in Section 8 with comparisons with related research on encodings of object-oriented features in linear logic. A separate appendix describes a sequent-style proof system for higher-order linear logic, based on [20].

2 A Fragment of Higher-Order Linear Logic

Several higher-order logic programming languages have been proposed in the literature, that extend the syntax of Horn Clauses with new constructs for modular programming [18] and direct encodings of data structures that embody notions of variable bindings [19, 22]. All of these languages have been proved to be conservative extensions of standard logic programming, in that they are amenable to complete presentations in *uniform-proof* formulations of intuitionistic logic [21]. At the same time, with the advent of linear logic [9], new logic programming languages have emerged that support notions of resources and resource management, and rely on related notions of uniform provability for both intuitionistic [10, 13] and classical linear logic [20, 23].

In defining the language $\mathcal{L}$, our goal is to isolate a higher-order extension of Horn Clauses that (i) provides support for representing objects, i.e., complex data structures comprising data and methods (formulas), while at the same time (ii) allowing methods residing within objects to be dynamically *loaded* and *consumed* right after having been selected for execution.

As we shall describe next, the desired features of $\mathcal{L}$ can be accounted for by allowing certain occurrences of higher-order variables in a program, and by enriching the set of logic programming connectives with the linear implication $\multimap$. We first describe the embedding of Horn clauses in linear logic, and then define the desired extension.

Horn clauses in linear logic. Horn clauses can be embedded in linear logic using the following map, defined by Girard in [9], from intuitionistic to linear logic formulas: $(B \wedge C)^\circ = B^\circ \,\&\, C^\circ$, $(B \supset C)^\circ = B^\circ \Rightarrow C^\circ$, $(true)^\circ = \mathbf{1}$, $(\forall X.B)^\circ = \forall X.B^\circ$, $(\exists X.B)^\circ = \exists X.B^\circ$, and $A^\circ = A$. In this encoding A denotes atomic formulas, and $\Rightarrow$ stands for intuitionistic implication, defined as $A \Rightarrow B \equiv (!A) \multimap B$. Following the classical subdivision into positive and negative clauses (cf. [18]), the resulting linear logic presentation of Horn clauses is given by the following productions.

$$\begin{array}{lcl} D & ::= & A \mid G \Rightarrow A \mid D \,\&\, D \mid \forall_\tau V.D \\ G & ::= & \mathbf{1} \mid A \mid G \,\&\, G \mid \exists_\tau V.G \end{array}$$

The distinction between D-formulas and G-formulas reflects their intended use: in a logic program, closed D-formulas play the role of *program* clauses, used to define the predicates of interest, and closed G-formulas serve as *goals*, used to query that program. The embedding of Horn clauses preserves provability in the following sense: the sequent $\Gamma \longrightarrow G$ is provable in intuitionistic logic if and only if $!(\Gamma^\circ) \longrightarrow G^\circ$ is provable in linear logic.

Design of the language $\mathcal{L}$. Following the standard practice in defining higher-order languages, the syntax of $\mathcal{L}$ is typed. However, instead of relying on simply typed λ-terms as in other languages (cf. [20]), the definition of $\mathcal{L}$ is based on algebraic terms and types, enriched with an object-level representation of formulas. As we shall see, this choice does not cause any loss of generality in the use of $\mathcal{L}$ as specification language of object-calculi; on the other hand, it makes the implementation of $\mathcal{L}$ (specifically, the definition of a unification algorithm for $\mathcal{L}$-terms) a straightforward task.

We assume that there are denumerably many base types, called *sorts*. All sorts are nonempty, and a distinguished sort, denoted by o, serves as the type of formulas. The syntax is based on a signature consisting of a denumerable set of constants: these are function symbols of algebraic types $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, where the σ_i's and σ are sorts. We distinguish logical from non-logical constants: the former include the connectives $\mathbf{1}{:}\mathsf{o}$, $\&, \Rightarrow, \multimap : \mathsf{o} \times \mathsf{o} \to \mathsf{o}$, and the set of quantifiers[1] $\exists_\sigma, \forall_\sigma : \sigma \times \mathsf{o} \to \mathsf{o}$, for every sort σ.

[1] This is different from the standard encoding of the quantifiers based on λ-terms, where $\exists_\tau x.P$ and $\forall_\tau.x.P$ are defined as shorthands for $(\Sigma_\tau\, \lambda x.P)$ and $(\Pi_\tau\, \lambda x.P)$, with Σ_τ and Π_τ constants of type $(\tau \to \mathsf{o}) \to \mathsf{o}$.

Formulas and terms are built over the signature Σ and a denumerable set of variables $\mathcal{V}$. The definition of $\mathcal{L}$-formulas arises from two changes in the syntax of Horn clauses: we extend the structure of D-formulas to include variable D-formulas, and we extend the structure of G-formulas by allowing linear implications to occur nested within G-formulas. Letting V range over variables of any sort (including the sort o) the resulting productions may be written as follows:

$$\begin{array}{rcl} D & ::= & A \mid G \Rightarrow A \mid D \,\&\, D \mid \forall_\tau V.D \mid V \\ G & ::= & \mathbf{1} \mid A \mid G \,\&\, G \mid \exists_\tau V.G \mid D \multimap A. \end{array}$$

A problem with the above definition arises from considering quantified formulas, in that the productions allow the formation of D-formulas such as $\forall_{\mathsf{o}} x.x$. These formulas are undesirable in the specification of logic programming languages for two reasons: firstly they make it possible to write inconsistent programs; secondly, as noted in [21], they are in straight contrast with the intended use of closed D-formulas as *program* clauses used to specify procedures to be evaluated by resolution steps. To disallow such formulas, we restrict the above productions as described in the following definition: as usual in defining a higher-order logic, terms and formulas are introduced simultaneously.

Definition 1 (Formulas and Terms) An atomic $\mathcal{L}$-term is either a variable or a *rigid* term $(h\, t_1 \ldots t_n) : \sigma$, where $h : \sigma_1 \times \ldots \times \sigma_n \to \sigma$ is a non-logical constant in Σ, and every $t_i : \sigma_i$ is either an atomic or a non-atomic $\mathcal{L}$-term. An atomic formula is a *rigid* atomic $\mathcal{L}$-term of type o. A non-atomic $\mathcal{L}$-term is a D-formula generated by the productions below, where A ranges over atomic (hence rigid) formulas, and V ranges over variables.

$$\begin{array}{lcl} D & ::= & A \mid G \Rightarrow A \mid D \,\&\, D \mid \forall_\tau V.D \\ G & ::= & \mathbf{1} \mid A \mid G \,\&\, G \mid \exists_\tau V.G \mid D^{\mathrm{V}} \multimap A \\ D^{\mathrm{V}} & ::= & D \mid V \mid D^{\mathrm{V}} \,\&\, D^{\mathrm{V}}. \end{array}$$

We identify two $\mathcal{L}$-terms s and t up to renaming of bound variables, i.e., we work modulo α-conversion (the identity being denoted by $s \equiv_\alpha t$, or simply $s \equiv t$). Note that variables D-formulas may only occur in $\mathcal{L}$-terms in the following position: either as antecedents of linear implications (i.e., immediately to the left of the *logical* symbol $\multimap$), or nested within the scope of *non-logical* constants (i.e., as parameters of terms and atomic predicates).

The structure of formulas and terms ensures an important property for $\mathcal{L}$-formulas, namely that the result of substituting D-formulas for the free variables of an $\mathcal{L}$-formula is again an $\mathcal{L}$-formula (i.e., $\mathcal{L}$-formulas are closed under substitution with D-formulas).

The above definition of D-formulas is consistent with their use as procedure definitions, and at the same time provides support for higher-order features needed in the specification of object calculi.

Proof rules for $\mathcal{L}$. A proof system for $\mathcal{L}$ is shown in Figure 1. The proof rules result from specializing the proof rules of the system Forum [20]. Forum is a complete presentation of linear logic built around a subset of the linear connectives. In [20],

Structural Rule

$$\frac{\Gamma, B \xrightarrow{B} A}{\Gamma, B \longrightarrow A}\ (decide)$$

Left Rules

$$\frac{}{\Gamma \xrightarrow{A} A}\ (initial)$$

$$\frac{\Gamma \xrightarrow{B_i} A \quad i \in \{1, 2\}}{\Gamma \xrightarrow{B_1 \,\&\, B_2} A}\ (\&_l)$$

$$\frac{\Gamma \longrightarrow B \quad \Gamma \xrightarrow{C} A}{\Gamma \xrightarrow{B \Rightarrow C} A}\ (\Rightarrow_l)$$

$$\frac{\Gamma \xrightarrow{B[t/x]} A}{\Gamma \xrightarrow{\forall_\tau x.B} A}\ (\forall_l)$$

Right Rules

$$\frac{}{\Gamma \longrightarrow \mathbf{1}}\ (\mathbf{1})$$

$$\frac{\Gamma \longrightarrow C_1 \quad \Gamma \longrightarrow C_2}{\Gamma \longrightarrow C_1 \,\&\, C_2}\ (\&_r)$$

$$\frac{\Gamma \xrightarrow{B} A}{\Gamma \longrightarrow B \multimap A}\ (\multimap_r)$$

$$\frac{\Gamma \longrightarrow C[t/x]}{\Gamma \longrightarrow \exists_\tau x.C}\ (\exists_r)$$

Proviso: in $(\forall_l)$ and $(\exists_r)$, $t : \tau$ is a closed $\mathcal{L}$-term of type τ. A denotes an atomic formula.

Figure 1: The proof system $\Pi_{\mathcal{L}}$.

$\mathcal{L}$	Forum
$\Gamma \longrightarrow C$	$\Sigma : \Gamma; \emptyset \longrightarrow C; \emptyset$
$\Gamma \xrightarrow{B} A$	$\Sigma : \Gamma; \emptyset \xrightarrow{B} A; \emptyset$

Figure 2: $\Pi_{\mathcal{L}}$ versus Forum sequents for a given Σ

.

Dale Miller defined a multi-conclusion sequent calculus for this logic, where proofs are uniform by construction. Sequents in this calculus have the form[2]:

$$\Sigma : \Gamma; \Delta \longrightarrow \Omega; \Upsilon \qquad \text{and} \qquad \Sigma : \Gamma; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon,$$

where Σ is a signature, Γ and Υ (the *unbounded* parts of the contexts) are sets of formulas, Δ (the *bounded* part of the context) is a multiset of formulas, Ω is a list of (atomic and non-atomic) formulas, $\mathcal{A}$ a list of atomic formulas, and B is a formula. All formulas are over the signature Σ.

The correspondence between sequents in the system $\Pi_{\mathcal{L}}$ and Forum sequents is established as shown in Figure 2. Since the signature for $\mathcal{L}$ is fixed ahead, and we do not have a $(\forall_r)$ rule, we omit explicit references to Σ in the sequents and proof rules of $\Pi_{\mathcal{L}}$. To ease the notation, we will henceforth use a corresponding abbreviation for Forum sequents as well.

In other words, our sequents are single-conclusion Forum sequents with empty or singleton bounded context on the left-hand side. The rules (*initial*) and (*decide*) correspond respectively to the rules (*initial*$_1$) and (*decide*$_\Gamma$) in Forum. The constant $\mathbf{1}$ can be defined using the equivalence $\mathbf{1} \equiv \bot \multimap \bot$ and the rule $(\mathbf{1})$ can be derived in Forum by $(\multimap_r)$, $(\bot_r)$ and $(\bot_l)$. The Forum rules for the exponential ?, and for the connectives $\parr$, $\top$ and $\bot$, as well as the rules $(\forall_r)$, $(\multimap_l)$, $(\Rightarrow_r)$, (*initial*$_2$), (*decide*$_\Delta$) and (*decide*?) do not have any corresponding rule in our system. Also note that, as in Forum, the right rules of the system are those for sequents of the form $\Gamma \longrightarrow C$. On the other hand, the left rules are *focussed* on (i.e., they are applied only to) the formula B that labels the sequent arrow in $\Gamma \xrightarrow{B} A$, with A an atomic formula.

When the formulas in Γ are closed D-formulas, B is a closed D-formula, C is a closed G-formula, and A is a closed atomic formula, we say that the sequents $\Gamma \longrightarrow C$ and $\Gamma \xrightarrow{B} A$ are *$\mathcal{L}$-sequents*. A proof of an $\mathcal{L}$-sequent that uses the proof rules of Figure 1 is said to be an $\mathcal{L}$-proof.

An inspection of the proof rules of system $\Pi_{\mathcal{L}}$ shows that the left rules are only applicable to sequents whose succedent is an atomic formula (cf. Figure 1): hence $\mathcal{L}$-proofs are uniform by construction. In the next section we show that $\mathcal{L}$-proofs are complete with respect to provability of $\mathcal{L}$-sequents in higher-order linear logic.

3 Completeness of $\mathcal{L}$-proofs

The completeness result we wish to prove may be stated precisely as follows:

[2]Sequents in Forum have, in fact, a more general structure. However, Forum proofs involve sequents with the structure we give here (cf. the $\mathcal{F}_1$ proof system in [20], reported in Appendix A).

every $\mathcal{L}$-sequent has a proof in higher-order linear logic if and only if it has an $\mathcal{L}$-proof.

There is one main restriction in $\mathcal{L}$-proofs with respect to proofs in higher-order linear logic, namely that the substitution terms used in $\mathcal{L}$-proofs are required to be $\mathcal{L}$-terms. As we already noted, the definition of D-formulas guarantees that the use of D-formulas (i.e., $\mathcal{L}$-terms of type o) as substitution terms for variables of type o preserves the structure of D-formulas, and hence of $\mathcal{L}$-sequents, along an $\mathcal{L}$-proof. Instead, the substitution terms introduced along a proof in higher-order linear logic can be any λ-terms, not necessarily $\mathcal{L}$-terms. Thus, in particular, terms substituted for variables of type o can be formulas other than D-formulas: consequently, the sequents arising in the proof of an $\mathcal{L}$-sequent may happen not to be $\mathcal{L}$-sequents. To prove completeness, we need to show that every such term may be systematically "$\mathcal{L}$-normalized", i.e., replaced by a corresponding $\mathcal{L}$-term, so that the resulting proof is, in fact, an $\mathcal{L}$-proof.

The idea may be described as follows: given an $\mathcal{L}$-sequent, call Ξ a proof (in higher-order linear logic) for this sequent. The "$\mathcal{L}$-normalization" of Ξ is accomplished in two steps. First we isolate the "sub-proof" of Ξ that has the structure of an $\mathcal{L}$-proof; then we show that every sequent in this sub-proof may be $\mathcal{L}$-normalized, and the sub-proof completed, to produce the desired $\mathcal{L}$-proof.

In the completeness proof we use Forum [20] as logic as reference: this choice does not involve any loss of generality, as it was shown in [20] that provability in Forum is the same as provability in higher-order linear logic. To ease the presentation, we shall use the derived Forum axiom ($\mathbf{1}$), defined as $\Gamma; \emptyset \longrightarrow \mathbf{1}; \Upsilon$ in place of the Forum proof of the equivalent formula $\perp \multimap \perp$. Also, to avoid possible ambiguities, we shall distinguish λ-terms from $\mathcal{L}$-terms using capital letters such as M and N to denote λ-terms, and lower-case letters like t and s to denote $\mathcal{L}$-terms. As for $\mathcal{L}$-terms, α-convertible λ-terms are identified, and we write $M \equiv N$ to state that M and N are identical terms; instead, we write $M =_\beta N$ when M and N are β-convertible. A final remark on quantifiers is in order. As we noted (footnote 1, page 4) our encoding of quantifiers differs from the standard one, adopted in Forum: however, since a bijection clearly exists between the two encodings, in the rest of this section we will identify the two formulas $\forall_\sigma x.F$ and $(\Pi_\sigma\, \lambda x.F)$, and similarly the two formulas $\exists_\sigma x.F$ and $(\Sigma_\sigma\, \lambda x.F)$.

3.1 $\mathcal{L}$-slices

The notion of "sub-proof" is made precise with the definition of $\mathcal{L}$-slice, given below. Before that, we introduce the following class of $\mathbf{D}$-formulas, a generalization of the D-formulas defined in Section 2.

Definition 2 (D-Formulas) We say that a formula F is a $\mathbf{D}$-formula if and only if either one of the following conditions hold: (i) F is a D-formula; (ii) there exists a $\mathbf{D}$-formula $\forall_\tau x.H$, and F is the formula $H[M/x]$ that results from substituting any closed λ-term in normal form M for every free occurrence x in H; (iii) F is a conjunction of $\mathbf{D}$-formulas.

The set of **D**-formulas can be characterized, equivalently, as the set of formulas generated by the following productions:

$$\begin{array}{lll} \mathbf{D} & ::= & \mathbf{A} \mid \mathbf{G} \Rightarrow \mathbf{A} \mid \mathbf{D} \,\&\, \mathbf{D} \mid \forall_\tau V.\mathbf{D} \\ \mathbf{G} & ::= & \mathbf{1} \mid \mathbf{A} \mid B \multimap \mathbf{A} \mid \mathbf{G} \,\&\, \mathbf{G} \mid \exists_\tau V.\mathbf{G}. \end{array}$$

Here **A** denotes any *rigid* atomic formula of the form $(h\ t_1\ \ldots\ t_n)$ where h is a nonlogical symbol and the t_i's are arbitrary λ-terms (not necessarily $\mathcal{L}$-terms), and B denotes arbitrary Forum formulas.

Definition 3 ($\mathcal{L}$-slice) Let Ξ be a Forum proof of an $\mathcal{L}$-sequent S. We define the $\mathcal{L}$-slice of Ξ, denoted by $\varsigma_{\mathcal{L}}(\Xi)$, to be the partial proof-tree that results from dropping the subproofs of all the sequents $\Gamma; \Delta \longrightarrow \mathcal{A}, B \multimap C, \Omega; \Upsilon$ occurring in Ξ, whenever the formula B is not a (closed) **D**-formula.

The definition of $\mathcal{L}$-slice motivates the introduction of the class of **D**-formulas: in fact, it is easily seen that the formulas in the antecedent of the sequents of $\varsigma_{\mathcal{L}}(\Xi)$ are closed **D**-formulas, whereas the formulas in the succedent are closed **G**-formulas.

We next prove a few lemmas that provide a more precise characterization of the structure of $\mathcal{L}$-slices of Forum proofs for $\mathcal{L}$-sequents. We first introduce a new class of sequents, and then give the desired results.

Definition 4 (L-sequents) We say that a Forum sequent is an **L**-sequent if it has one of the following forms: $\Gamma; \mathbf{D} \longrightarrow \mathbf{A}; \emptyset$, $\Gamma; \emptyset \longrightarrow \mathbf{G}; \emptyset$, or $\Gamma; \emptyset \xrightarrow{\mathbf{D}} \mathbf{A}; \emptyset$, where Γ is a set of closed D-formulas, **D** is a closed **D**-formula, **G** is a closed **G**-formula, and **A** is a closed **A**-formula.

L-sequents generalize $\mathcal{L}$-sequents in a way similar to how **D** and **G**-formulas generalize D and G-formulas. What we show next is that if we take a Forum proof of an **L**-sequent, then every sequent in the $\mathcal{L}$-slice of the proof is, in fact, an **L**-sequent. We first need the following auxiliary result.

Lemma 5 Let Γ be a set of closed D-formulas, **D** be a closed **D**-formula, **G** a closed **G**-formula, **A** a closed **A**-formula, and Δ a multiset of closed **D**-formulas. Then we have:

- if the sequent $\Gamma; \Delta \xrightarrow{\mathbf{D}} \mathbf{A}; \emptyset$ is provable in Forum, then Δ must be the empty multiset;
- if the sequent $\Gamma; \Delta \longrightarrow \mathbf{A}; \emptyset$ is provable in Forum, and Δ is non empty, then Δ must be a singleton.

Proof By a straightforward induction on the structure of the Forum proof.

Lemma 6 Let S be an **L**-sequent, and Ξ be a Forum proof of S. Then, every sequent in $\varsigma_{\mathcal{L}}(\Xi)$ is an **L**-sequent.

Proof The proof is by induction on the height of Ξ. The claim is clearly true when Ξ has height 1, as in this case S must be an axiom ($initial_1$) of the form $\Gamma; \emptyset \xrightarrow{\mathbf{A}} \mathbf{A}; \emptyset$, or an axiom ($\mathbf{1}$) of the form $\Gamma; \emptyset \longrightarrow \mathbf{1}; \emptyset$. Then, assume that the claim holds for all proofs (of **L**-sequents) of height less than h, and consider a Forum proof Ξ of height h. The proof is now by a case analysis on the last rule used in Ξ. Most cases are vacuous since the rule could not have been applied to S, S being an **L**-sequent. Below we give the most interesting cases.

- $(\multimap_r)$. Since S is an **L**-sequent, it must be of the form $\Gamma; \emptyset \longrightarrow B \multimap \mathbf{A}; \emptyset$. We distinguish two subcases, depending on the structure of B. If B is not a **D**-formula, the thesis follows immediately, as the sequent is a leaf of $\varsigma_{\mathcal{L}}(\Xi)$. If, instead, B is a **D**-formula, the premise of the rule is the sequent $\Gamma; B \longrightarrow \mathbf{A}; \emptyset$, which is an **L**-sequent. The thesis follows then from the induction hypothesis.
- $(decide_\Gamma)$ The claim follows directly from the induction hypothesis, as the sequent in question may only be of the form $\Gamma; \emptyset \longrightarrow \mathbf{A}; \emptyset$. The other case, when the sequent in question is $\Gamma; \mathbf{D} \longrightarrow \mathbf{A}; \emptyset$, is vacuous. To see why, note that the upper sequent of the rule would be the sequent $\Gamma; \mathbf{D} \xrightarrow{D'} \mathbf{A}; \emptyset$ for some D' in Γ. By Lemma 5, we know that the latter sequent is not provable in Forum: hence Ξ would not be a Forum proof, a contradiction.
- $(decide_\Delta)$ The sequent in question is again $\Gamma; \mathbf{D} \longrightarrow \mathbf{A}; \emptyset$ and the premise of the rule is $\Gamma; \emptyset \xrightarrow{\mathbf{D}} \mathbf{A}; \emptyset$, an **L**-sequent. The claim follows then from the induction hypothesis.

With the above proof, we implicitly have proved the following result.

Corollary 7 Let Ξ be a Forum proof of an **L**-sequent. Then, every instance of $(\multimap_r)$ occurring in $\varsigma_{\mathcal{L}}(\Xi)$ and focusing on the formula **D**, always occurs just below an instance of $(decide_\Delta)$ with **D** as principal formula.

In other words, all the derivation schemes of the form

$$\dfrac{\dfrac{\Gamma; \emptyset \xrightarrow{\mathbf{D}} \mathbf{A}; \emptyset}{\Gamma; \mathbf{D} \longrightarrow \mathbf{A}; \emptyset}\ (decide_\Delta)}{\Gamma; \emptyset \longrightarrow \mathbf{D} \multimap \mathbf{A}; \emptyset}\ (\multimap_r)$$

occurring in an $\mathcal{L}$-slice can be coalesced and replaced by an application of the $(\multimap_r)$ of system $\Pi_{\mathcal{L}}$ (cf. Section 2, Figure 1). As a consequence, we may simplify the definition of **L**-sequents to comprise only sequents in the forms $\Gamma; \emptyset \longrightarrow \mathbf{G}; \emptyset$ or $\Gamma; \emptyset \xrightarrow{\mathbf{D}} \mathbf{A}; \emptyset$.

Given this observation, we will further simplify the notation of **L**-sequents and use the following presentation:

$$\begin{array}{lll} \Gamma \longrightarrow \mathbf{G} & \text{for} & \Gamma; \emptyset \longrightarrow \mathbf{G}; \emptyset, \\ \Gamma \xrightarrow{\mathbf{D}} \mathbf{A} & \text{for} & \Gamma; \emptyset \xrightarrow{\mathbf{D}} \mathbf{A}; \emptyset. \end{array}$$

A final lemma establishes the relations between the **D** and **G**-formulas occurring in an $\mathcal{L}$-slice, and their corresponding D and G-formulas.

Lemma 8 Let S be an $\mathcal{L}$-sequent, Ξ be a Forum proof of S, and let $\Gamma \longrightarrow \mathbf{G}$ and $\Gamma \xrightarrow{\mathbf{D}} \mathbf{A}$ be **L**-sequents in the $\mathcal{L}$-slice $\varsigma_{\mathcal{L}}(\Xi)$. Then:

- every formula in $\Gamma \cup \{\mathbf{D}\}$ may be written as $D[M_1/x_1]\ldots[M_n/x_n]$, where D is a D-formula that has the same top-level connective as the formula in question, the variables $x_1,\ldots,x_n$ are free in D, and the M_i's are closed λ-terms in normal form;
- the formula $\mathbf{G}$ (respectively $\mathbf{A}$) may be written as $G[N_1/y_1]\ldots[N_m/y_m]$ (resp. $A[N_1/y_1]...[N_m/y_m]$) where G (resp. A) is a G-formula that has the same top-level connective as $\mathbf{G}$ (resp. A), the variables $y_1,\ldots,y_m$ are free in G (resp. A), and the N_i's are closed λ-terms in normal form.

Proof The proof is by induction on the height h of the $\mathcal{L}$-slice. In order for induction to work, we prove a slightly more general result, where S is an $\mathbf{L}$-sequent, rather than an $\mathcal{L}$-sequent. The lemma follows then immediately, noting that $\mathcal{L}$-sequents are also $\mathbf{L}$-sequents (this follows immediately by definition of $\mathbf{D}$- and $\mathbf{G}$-formulas).

The base of induction, when $h = 1$, follows immediately: the only nontrivial case is when S is the sequent $\Gamma \longrightarrow B \multimap \mathbf{A}$, which can be written as $(x \multimap A)[\ldots, B/x, \ldots]$ for some atomic formula A (note that B is a closed non-D-formula).

When $h > 1$, the proof is a case analysis on the last Forum rule in the slice. Most cases are vacuous, as S is an $\mathbf{L}$-sequent and the rule could not have been applied to S; the non-vacuous cases follow directly by induction hypothesis on the upper sequents of the rule, which are easily verified to be $\mathbf{L}$-sequents, when the lower sequent is itself an $\mathbf{L}$-sequent.

Having formalized the notion of "sub-proof" we move on to the next step of our construction. As we anticipated, the idea is to "$\mathcal{L}$-normalize" every "wrong" term arising in the proof, i.e., replace that term with a corresponding $\mathcal{L}$-term. The argument we use is constructive, and relies on essentially the same idea used in the completeness proof for the language of higher-order hereditary Harrop formulas (*hohh*) of [21]. As in that case, there are two cases to consider, depending on whether the "wrong" term is substituted for a variable that occurs in the scope of a non-logical or logical constant. In the first case the solution is simple because the wrong term may be $\mathcal{L}$-normalized to any $\mathcal{L}$-term of type o. In the second case, since we are restricting attention to $\mathcal{L}$-slices, the "wrong" term must be a formula occurring to the left of a linear implication, as B in $B \multimap A$, with $B \multimap A$ resulting from substituting a variable D^{V}-formula in the antecedent of a G-formula $D^{\mathrm{V}} \multimap \mathbf{A}$. Now, it would seem natural to $\mathcal{L}$-normalize $B \multimap A$ to the tautological formula $\nu(\mathbf{A}) \multimap \nu(\mathbf{A})$, where $\nu(\mathbf{A})$ is the $\mathcal{L}$-normalization of $\mathbf{A}$. However, a problem with this simple $\mathcal{L}$-normalization scheme arises in situations where the same substitution term is used in different sequents along the $\mathcal{L}$-slice.

Consider the following $\mathcal{L}$-slice of a derivation, where $\phi : \mathsf{o}$ is not a $\mathbf{D}$-formula, and where Γ contains the following D-formulas:

$$D_1 = \forall v.\, \underbrace{((v \multimap p(v))\,\&\,p(v)) \Rightarrow q}_{D_1'}; \qquad D_2 = \forall w.\, \underbrace{((w \multimap r(w))) \Rightarrow p(w)}_{D_2'}.$$

$$
\cfrac{
\cfrac{
\cfrac{}{\Gamma \longrightarrow (v \multimap p(v))[\phi/v]}\,(\multimap_r) \qquad \Xi'
}{\Gamma \longrightarrow (v \multimap p(v)) \,\&\, p(v)[\phi/v]}\,(\&_r)
\qquad
\cfrac{}{\Gamma \xrightarrow{q} q}\,(\mathit{initial})
}{
\cfrac{\Gamma \xrightarrow{D_1'[\phi/v]} q}{\Gamma \longrightarrow q}\,(\mathit{decide} + \forall_l)
}\,(\Rightarrow_l)
$$

Here Ξ' is the following proof:

$$
\cfrac{
\cfrac{}{\Gamma \longrightarrow (w \multimap r(w))[\phi/w]}\,(\multimap_r)
\qquad
\cfrac{}{\Gamma \xrightarrow{p(w)[\phi/w]} p(v)[\phi/v]}\,(\mathit{initial})
}{
\cfrac{\Gamma \xrightarrow{D_2'[\phi/w]} p(v)[\phi/v]}{\Gamma \longrightarrow p(v)[\phi/v]}\,(\mathit{decide}) + (\forall_l)
}\,(\forall_l)
$$

We assume that $\Gamma \xrightarrow{\phi} p(\phi)$ and $\Gamma \xrightarrow{\phi} r(\phi)$ have Forum proofs. Now, if we used the naive $\mathcal{L}$-normalization we outlined above, we would end up with two $\mathcal{L}$-normalizing terms for ϕ, namely: $p(\nu(\phi))$ and $r(\nu(\phi))$ where $\nu(\phi)$ denotes any $\mathcal{L}$-term of type o. Clearly, then, the result of $\mathcal{L}$-normalization would not be a proof, for what we actually need is the $\mathcal{L}$-term $p(\nu(\phi)) \,\&\, r(\nu(\phi))$. Below, we give a technique for systematically computing $\mathcal{L}$-normalization terms for any given $\mathcal{L}$-slice. The technique generalizes the idea we just illustrated yielding $\mathcal{L}$-normalization terms in the form of (conjoined) universally quantified D-formulas.

3.2 Computing $\mathcal{L}$-Normalization Terms

As illustrated by the previous example, $\mathcal{L}$-normalization terms may not, generally, be determined by simply looking at the sequent where they are required: a global inspection of the $\mathcal{L}$-slice is needed to find *the* appropriate D-formula for every "wrong" witness of type o in the original proof. However, what we can do, locally to every sequent, is to collect the $\mathcal{L}$-normalizing terms for every potentially "wrong" witness that might be introduced by a substitution over a variable of type o. This is the idea behind the following definition.

Definition 9 (Local $\mathcal{L}$-Normalization) We say that a variable v has a *formula occurrence* in a formula F if and only if v has at least one occurrence in F that is not in the scope of a non-logical constant. Let $\mathbf{F}$ be a $\mathbf{D}$ or $\mathbf{G}$-formula, let B denote any formula, and let v be a variable. The local $\mathcal{L}$-normalization terms for v in $\mathbf{F}$, written $\mathbf{N}[v, \mathbf{F}]$ is defined inductively as follows:

$$
\begin{aligned}
\mathbf{N}[v, \mathbf{F}_1 \,\&\, \mathbf{F}_2] &= \mathbf{N}[v, \mathbf{F}_1] \cup \mathbf{N}[v, \mathbf{F}_2];\\
\mathbf{N}[v, \forall_\tau x.\mathbf{D}] &= \mathbf{N}[v, \mathbf{D}];\\
\mathbf{N}[v, \exists_\tau x.\mathbf{G}] &= \mathbf{N}[v, \mathbf{G}];\\
\mathbf{N}[v, \mathbf{G} \Rightarrow \mathbf{A}] &= \mathbf{N}[v, \mathbf{G}];
\end{aligned}
$$

$$\begin{array}{rcl}\mathbf{N}[v, B \multimap \mathbf{A}] & = & \{\mathbf{A}\} \cup \mathbf{N}[v, B] \text{ if } v \text{ has a formula occurrence in } B, \\ \mathbf{N}[v, B \multimap \mathbf{A}] & = & \mathbf{N}[v, B], \text{ otherwise;} \\ \mathbf{N}[v, B] & = & \emptyset \text{ if none of the above applies.}\end{array}$$

To exemplify, we have $\mathbf{N}[v, D_1'] = \{p(v)\}$ and $\mathbf{N}[w, D_2'] = \{r(w)\}$ in the formulas of the previous derivation.

Given the $\mathcal{L}$-slice of a Forum proof, we may compute the $\mathcal{L}$-normalization term of type o by "decorating" each sequent in the $\mathcal{L}$-slice with a set $\mathcal{N}$ of local $\mathcal{L}$-normalization terms. Where $\mathcal{N} \mid \Gamma \longrightarrow G$ denotes a decorated sequent, this process can be accomplished in the following manner:

- the set $\mathcal{N}$ at the leaves of the slice is empty;
- for every instance of a rule with two premises, the set $\mathcal{N}$ at the lower sequent is the union of the corresponding sets at the upper sequents;
- for every instance of a rule with one premise, with the exception of the $(\forall_l)$ and $(\exists_r)$ rules, the set $\mathcal{N}$ attached to the upper sequent is simply "copied" in the lower sequent;
- for every instance of $(\forall_l)$ the set $\mathcal{N}$ is computed as follows:

$$\frac{\mathcal{N} \mid \Gamma \xrightarrow{\mathbf{D}[t/x]} \mathbf{A}}{\mathcal{N} \cup \mathbf{N}[x, \mathbf{D}] \mid \Gamma \xrightarrow{\forall_{\mathsf{o}} x.\mathbf{D}} \mathbf{A}} \ (\forall_l)$$

- for every instance of $(\exists_r)$ the set $\mathcal{N}$ is computed as follows:

$$\frac{\mathcal{N} \mid \Gamma \longrightarrow \mathbf{G}[t/x]}{\mathcal{N} \cup \mathbf{N}[x, \mathbf{G}] \mid \Gamma \longrightarrow \exists_{\mathsf{o}} x.\mathbf{G}} \ (\exists_r)$$

Proceeding in this way at every sequent of the $\mathcal{L}$-slice, the set $\mathcal{N}$ occurring at the root of the $\mathcal{L}$-slice can be characterized as follows.

Lemma 10 Let Ξ be a Forum proof of an $\mathcal{L}$-sequent $\Gamma \longrightarrow G$, and let $\mathcal{N}$ be the decoration associated to the root of $\varsigma_{\mathcal{L}}(\Xi)$ by the process just described. Then, for every $\mathbf{A}$ such that $\Gamma \longrightarrow B \multimap \mathbf{A}$ is a leaf sequent of $\varsigma_{\mathcal{L}}(\Xi)$, it is the case that $\mathbf{A} \in \mathcal{N}$.

Proof By construction, noting that, by the definition of $\mathbf{G}$-formulas, B corresponds to the formula $v[t_1/x_1, \ldots, B/v, t_n/x_n]$, for some variable $v : \mathsf{o}$.

Definition 11 (Canonical $\mathcal{L}$-Normalization Terms) We introduce a *canonical $\mathcal{L}$-normalization term* at each sort. For sorts other than o choose any distinguished constant of that sort as the normalization term, and denote it with ν_σ. For the sort o, $\mathcal{L}$-normalization terms are defined as follows. Let $\varsigma_{\mathcal{L}}$ be an $\mathcal{L}$-slice, and let $\mathcal{N}$ be the decoration computed at the root of $\varsigma_{\mathcal{L}}$ by decorating each sequent in $\varsigma_{\mathcal{L}}$ in the manner described above. The canonical $\mathcal{L}$-normalization term of type o, denoted by ν_{o}, is defined as follows:

· if $\mathcal{N} = \{\mathbf{A}_1, \ldots, \mathbf{A}_n\}$, and $\mathbf{A}_i = (h_i\, M_1 \ldots M_k)$, then $\nu_\circ = A_1^\forall \,\&\, \cdots \,\&\, A_n^\forall$, where $A_i^\forall = \forall_{\tau_1} x_1. \ldots \forall_{\tau_n} x_k.(h_i\, x_1 \ldots x_k)$.

· If $\mathcal{N} = \emptyset$, then $\nu_\circ = c_\circ$, where $c_\circ$ is any constant symbol of type $\circ$ in Σ.

When $\mathcal{N} \neq \emptyset$, the definition of $\nu_\circ$ formalizes the idea we already described. When $\mathcal{N} = \emptyset$, all the λ-terms to be $\mathcal{L}$-normalized in the slice occur nested within a nonlogical constant: in this case, any choice of $\nu_\circ$ serves our purposes. The notion of $\mathcal{L}$-normalization terms is well defined, given our initial assumption that all sorts (including $\circ$) are nonempty. It only remains, now, to define the $\mathcal{L}$-normalization mapping that transforms every formula into a corresponding D-formula (preserving $\mathcal{L}$-terms and D-formulas).

The normalization mapping is defined over the class of λ-terms that arise from substituting normal-form λ-terms for variables of the $\mathcal{L}$-terms occurring in the $\mathcal{L}$-slice of a Forum proof.

Definition 12 ($\mathcal{L}$-Normalization) Let Ξ be a Forum proof, σ be a sort, and let $M : \sigma$ be a λ-term in normal form occurring in $\varsigma_{\mathcal{L}}(\Xi)$. The $\mathcal{L}$-normalization of M in $\varsigma_{\mathcal{L}}(\Xi)$, written $\nu(M)$, is defined inductively as follows:

· if $M : \sigma$ is a typed constant or a variable, then $\nu(M) = M$;

· if $M : \sigma$ is the term $(M_0\, M_1 \ldots M_n)$, then distinguish the following three subcases:

  – if $M_0 : \sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ is a nonlogical constant, and $M_i : \sigma_i$ for $i = 1..n$, then $\nu(M) = (M_0\, \nu(M_1) \ldots \nu(M_n))$;
  – if M_0 is a logical constant, and M is a $\mathbf{D}$-formula, then $\nu(M) = \nu^-(M)$ where ν^- is the mapping defined by the following mutual recursion:

$$
\begin{array}{rcl}
\nu^-(\mathbf{A}) & = & \nu(\mathbf{A}) \\
\nu^-(\mathbf{G} \Rightarrow \mathbf{A}) & = & \nu^+(\mathbf{G}) \Rightarrow \nu(\mathbf{A}) \\
\nu^-(\mathbf{D} \,\&\, \mathbf{D}) & = & \nu^-(\mathbf{D}) \,\&\, \nu^-(\mathbf{D}) \\
\nu^-(\forall_\tau v.\mathbf{D}) & = & \forall_\tau v.\nu^-(\mathbf{D})
\end{array}
\qquad
\begin{array}{rcl}
\nu^+(\mathbf{1}) & = & \mathbf{1} \\
\nu^+(\mathbf{A}) & = & \nu(\mathbf{A}) \\
\nu^+(B \multimap \mathbf{A}) & = & \nu(B) \multimap \nu(\mathbf{A}) \\
\nu^+(\mathbf{G} \,\&\, \mathbf{G}) & = & \nu^+(\mathbf{G}) \,\&\, \nu^+(\mathbf{G}) \\
\nu^+(\exists_\tau v.\mathbf{G}) & = & \exists_\tau v.\nu^+(\mathbf{G})
\end{array}
$$

  – $\nu(M) = \nu_\sigma$ otherwise.

Remarks The definition of $\mathcal{L}$-normalization is given without explicit treatment of λ-abstraction: this is consistent with the intended domain of the normalization mapping. In fact, as λ-terms are substituted in $\mathcal{L}$-terms for variables of basic types, they may not be λ-abstractions; furthermore, since substitution terms are assumed to be in normal form, λ-abstractions may only occur within substitution terms in the argument position of a λ-normal application subterm: all such occurrences are normalized away by the last clause of the definition of $\mathcal{L}$-normalization.

The normalization mapping satisfies some important properties: it preserves equality over the class of terms of interest, and it commutes with substitution of λ-terms into $\mathcal{L}$-terms. As a reminder, we recall that equality over simply-typed λ-terms is β-convertibility, denoted by $=_\beta$. When restricting to λ-terms in normal form (or to

$\mathcal{L}$-terms) equality reduces to α-convertibility, hence to syntactical identity, as we work modulo renaming of bound variables.

Lemma 13 Let σ be a sort, and $M, N : \sigma$ be two λ-terms in normal form. Then $\nu(M) \equiv \nu(N)$ whenever $M \equiv N$.

Proof Immediate, since the normalization mapping ν does not depend on the names of bound variables: hence it maps α-convertible λ-terms into α-convertible $\mathcal{L}$-terms. Also note that the result holds when M and N are $\mathcal{L}$-terms, as $\mathcal{L}$-terms are also λ terms in normal form.

Lemma 14 Let M be a closed λ term in normal form, and let t be any $\mathcal{L}$-term with x free in t, and M free for x in t. Then $\nu(t[M/x]) \equiv t[\nu(M)/x]$, and $t[\nu(M)/x]$ is itself an $\mathcal{L}$-term.

Proof The proof is by induction on the structure of t. We note, however, that not every subterm of a given $\mathcal{L}$-term is itself an $\mathcal{L}$-term. Hence, in order for induction to work, we prove the following, more general, result.

1. if $t : \sigma$ is an atomic $\mathcal{L}$-term then $\nu(t[M/x]) \equiv t[\nu(M)/x]$, and $t[\nu(M)/x]$ is an $\mathcal{L}$-term;
2. if $t : \mathsf{o}$ is a D-formula, then $\nu^-(t[M/x]) \equiv t[\nu(M)/x]$, and $t[\nu(M)/x]$ is a D-formula;
3. if $t : \mathsf{o}$ is a G-formula, then $\nu^+(t[M/x]) \equiv t[\nu(M)/x]$, and $t[\nu(M)/x]$ is a G-formula.

The proof is by induction on the structure of t, simultaneously for (1), (2) and (3).

1.
   - If t is a variable or a constant, the claim follows directly from the definition of normalization (cf. Definition 12).
   - If $t \equiv (h\, t_1 \ldots t_n) : \sigma$, then $\nu((h\, t_1 \ldots t_n)[M/x]) \equiv \nu(h\, t_1[M/x] \ldots t_n[M/x])$ and the latter is equal to $(h\, \nu(t_1[M/x]) \ldots \nu(t_n[M/x]))$ by definition. Then the proof follows from the induction hypothesis (1) or (2) depending on the structure of the t_i's.
2. If t is atomic the proof follows exactly as in the second subcase above. In all the remaining cases, the thesis follows from the induction hypothesis (1), (2) or (3).
3. If $t \equiv \mathbf{1}$ the claim follows immediately. If t is atomic the claim follows exactly as in case (2) above. In all the induction cases the thesis follows from the induction hypothesis (1), (2) or (3).

Lemma 15 Let t, s be two $\mathcal{L}$-terms with x free in t and y free in s, and M, N two closed λ-terms in normal form such that M is free for x in t and N is free for y in s. Then, $t[\nu(M)/x] =_\beta s[\nu(N)/y]$ whenever $t[M/x] =_\beta s[N/y]$.

Proof Since M and N are assumed to be in normal form, and substitution of terms of basic types in $\mathcal{L}$-terms does not produce β-redexes, one has that $t[M/x] =_\beta s[N/y]$ if and only if $t[M/x] \equiv s[N/y]$. Now, by lemma 13 and by lemma 14, it follows that $t[\nu(M)/x] \equiv \nu(t[M/x]) \equiv \nu(s[N/y]) \equiv s[\nu(N)/y]$.

As a consequence, the following property holds.

Lemma 16 Let D, D' be two D-formulas, and $M_1, \ldots, M_n, N_1, \ldots, N_k$ be closed λ-terms in normal form. If $D[M_1/x_1]\ldots[M_n/x_n] =_\beta D'[N_1/y_1]\ldots[N_k/y_k]$, then $D[\nu(M_1)/x_1]\ldots[\nu(M_n)/x_n] =_\beta D'[\nu(N_1)/y_1]\ldots[\nu(N_k)/y_k]$.

Proof Observe that $D[\nu(M_1)/x_1]\ldots[\nu(M_n)/x_n] \equiv D[\nu(M_1)/x_1, \ldots, \nu(M_n)/x_n]$, as substitutions of closed λ terms in normal form for variables of basic types do not compose. Then the proof follows from Lemma 15 by induction on n.

We may now prove the desired completeness result.

Theorem 17 (Completeness vs Forum) Given an $\mathcal{L}$-sequent of the form $\Gamma \longrightarrow G$, this sequent has a proof in Forum if and only if it has an $\mathcal{L}$-proof.

Proof The "only if" part of the proof is trivial, as $\mathcal{L}$-proofs are, in fact, Forum proofs. For the "if" part, we reason as follows.

Given Ξ, the Forum proof of the sequent, consider the $\mathcal{L}$-slice of Ξ: from Lemma 6, we know that sequents in the $\mathcal{L}$-slice have either the form $\Gamma \longrightarrow \mathbf{G}$, or $\Gamma \xrightarrow{\mathbf{D}} \mathbf{A}$, where $\mathbf{D}$ is a $\mathbf{D}$-formula, $\mathbf{G}$ is a $\mathbf{G}$-formula, and $\mathbf{A}$ is an $\mathbf{A}$-formula. Given any formula $\mathbf{F} \equiv F[M_1/x_1]\ldots[M_n/x_n]$, let now $\widehat{\mathbf{F}}$ denote the formula $F[\nu(M_1)/x_1]\ldots[\nu(M_n)/x_n]$. We show that for every sequent of the form $\Gamma \longrightarrow \mathbf{G}$ or $\Gamma \xrightarrow{\mathbf{D}} \mathbf{A}$ of the $\mathcal{L}$-slice an $\mathcal{L}$-proof exists for the sequent $\Gamma \longrightarrow \widehat{\mathbf{G}}$ and $\Gamma \xrightarrow{\widehat{\mathbf{D}}} \widehat{\mathbf{A}}$, respectively. The theorem follows then easily from this claim. The proof of the claim is by induction on height of the subderivation rooted at the sequent in question.

Base Case. If the subderivation has height 1, then the sequent in question either $(\mathbf{1})$ or $(initial)$. In the former case the claim immediately follows. In the latter case, from Lemma 8, we know that it may be written as

$$\Gamma \xrightarrow{A[M_1/x_1]\ldots[M_n/x_n]} A'[N_1/y_1]\ldots[N_m/y_m],$$

with $A[M_1/x_1]\ldots[M_n/x_n] =_\beta A'[N_1/y_1]\ldots[N_m/y_m]$. By Lemma 16, this implies that also $A[\nu(M_1)/x_1]\ldots[\nu(M_n)/x_n] =_\beta A'[\nu(N_1)/y_1]\ldots[\nu(N_m)/y_m]$. Thus, the sequent

$$\Gamma \xrightarrow{A[\nu(M_1)/x_1]\ldots[\nu(M_n)/x_n]} A'[\nu(N_1)/y_1]\ldots[\nu(N_m)/y_m]$$

is initial, and hence derivable.

Induction Cases. By a case analysis on the last rule of the subderivation Ξ. We first consider the cases of the *left* rules. Let $\Gamma \xrightarrow{\mathbf{D}} \mathbf{A}$ be the sequent in question:

- $(\&_l)$. In this case $\mathbf{D} = D[M_1/x_1]\ldots[M_n/x_n] = D_1 \,\&\, D_2$, for some formulas D_1 and D_2. From Lemma 8 we know that D is a D-formula of the form $D'_1 \,\&\, D'_2$, for suitable D'_1 and D'_2, which implies that the D_i's, (for $i = 1, 2$), are of form $D'_i[M_1/x_1]\ldots[M_n/x_n]$. Furthermore, since D is a D-formula, so are the two formulas D_i'; from the induction hypothesis, it then follows that the two sequents

$$\Gamma \xrightarrow{D_i'[\nu(M_1)/x_1]\ldots[\nu(M_n)/x_n]} \widehat{\mathbf{A}},$$

(for $i = 1, 2$) have an $\mathcal{L}$-proof. An application of ($\&_l$) yields now the desired $\mathcal{L}$-proof for

$$\Gamma \xrightarrow{D[\nu(M_1)/x_1]\ldots[\nu(M_n)/x_n]} \widehat{\mathbf{A}}.$$

- The same reasoning applies to the remaining left rules: the cases ($\bot_l$), ($\multimap_l$) and ($\invamp_l$) follow vacuously because none of these rules could have been applied, given the hypothesis on the structure of $\mathbf{D}$. We give the case of ($\forall_l$) as representative of the remaining possible cases. In this case $D[M_1/x_1]\ldots[M_n/x_n]$ is of the form $\forall x.D'[M_1/x_1]\ldots[M_n/x_n]$ for some D-formula D', and the premise of the rule is the sequent

$$\Gamma \xrightarrow{D'[M_1/x_1]\ldots[M_n/x_n][t/x]} \mathbf{A},$$

  for some closed λ-term t in normal form. From the induction hypothesis, we know that

$$\Gamma \xrightarrow{D'[\nu(M_1)/x_1]\ldots[\nu(M_n)/x_n][\nu(t)/x]} \widehat{\mathbf{A}}$$

  has an $\mathcal{L}$-proof, and the $\mathcal{L}$-proof for the desired sequent may be obtained by an application of ($\forall_l$) on the last sequent.

Next we consider the *right* rules. Most cases are vacuous, for the rule could not have been applied to the sequent in question. The case of ($\exists_r$) is similar to the case of ($\forall_l$) we just considered. The cases ($\&_r$) and ($decide_\Gamma$) follow from the induction hypothesis. The case of ($\multimap_r$) is worked out below.

Assume that the sequent is $\Gamma \longrightarrow B \multimap \mathbf{A}$, derived by ($\multimap_r$). ¿From Lemma 8, we know that $B \multimap \mathbf{A}$ may be written as $(D \multimap A)[N_1/y_1]\ldots[N_m/y_m]$ for some G-formula $D \multimap A$. Now we distinguish the following cases, depending on the structure of D.

- If D is a D-formula, the claim follows from the induction hypothesis, for

$$(D \multimap A)[N_1/y_1]\ldots[N_m/y_m] = D[N_1/y_1]\ldots[N_m/y_m] \multimap A[N_1/y_1]\ldots[N_m/y_m],$$

  and the upper sequent of the rule is the sequent

$$\Gamma \xrightarrow{D[N_1/y_1]\ldots[N_m/y_m]} A[N_1/y_1]\ldots[N_m/y_m],$$

  derivable by hypothesis.

- If, instead, D is not a D-formula, then it must be either a variable, or a conjunction of $D^{\vee}$-formulas. The first case is worked out next, the second is similar.

  Given that $D[N_1/y_1]\ldots[N_m/y_m]$ is closed, if D is a variable it must be the case that $D = y_i$, for some $i = 1, \ldots, m$, and $D[N_1/y_1]\ldots[N_m/y_m] = N_i$. Now, we have two possible subcases: (i) either there exists a D-formula D', and closed terms N_i', and variables y_i' free in D' such that $D'[N_1'/y_1']\ldots[N_k'/y_k'] = N_i$, or

(ii) no such D' exists. In the first subcase the claim follows from the induction hypothesis, for

$$D[N_i/y_i]\dots[N_m/y_m] \multimap A[N_1/y_1]\dots[N_m/y_m]$$

may be written as

$$D'[N_1'/y_1']\dots[N_k'/y_k'] \multimap A[N_1/y_1]\dots[N_m/y_m],$$

and we can apply the induction hypothesis on the premise of the rule. In the second subcase, the sequent $\Gamma \longrightarrow B \multimap \mathbf{A}$ must be a leaf-sequent in the $\mathcal{L}$-slice. ¿From Lemma 10 it follows that $\mathbf{A} \in \mathcal{N}$, where $\mathcal{N}$ is the decoration occurring at the root of the $\mathcal{L}$-slice. Letting $\mathbf{A} = (h\, M_1 \dots M_k)$, from Definition 11, we have that

$$\nu_\circ = A_1^\forall \dots \,\&\, \forall_{\tau_1} x_1.\dots\forall_{\tau_k} x_k.h(x_1,\dots,x_k) \,\&\, \dots A_n^\forall,$$

for some $A_1^\forall,\dots,A_n^\forall$. Finally, since by definition, it holds that

$$\nu(B \multimap \mathbf{A}) = \nu_\circ \multimap (h\,\nu(M_1)\dots\nu(M_k)),$$

an $\mathcal{L}$-proof for $\Gamma \longrightarrow \nu(B \multimap \mathbf{A})$ can be formed as shown below:

$$\dfrac{}{\Gamma \xrightarrow{(h\,x_1\dots x_k)[\nu(M_1)/x_1]\dots[\nu(M_k)/x_k]} (h\,\nu(M_1)\dots\nu(M_k))}\ (initial)$$

$$\vdots\ k\ (\forall_l)\ steps$$

$$\Gamma \xrightarrow{\forall x_1.\dots\forall x_k.(h\,x_1\dots x_k)} (h\,\nu(M_1)\dots\nu(M_k))$$

$$\vdots\ n\ (\&_l)\ steps$$

$$\Gamma \xrightarrow{\nu_\circ} (h\,\nu(M_1)\dots\nu(M_k))$$

4 A Calculus of Objects

In this section we describe the syntax and the semantics of $\mathbf{Ob}_{\multimap}$, an untyped object-based language that has all the essential ingredients of the object calculi we wish to characterize. The syntax of $\mathbf{Ob}_{\multimap}$ resembles the syntax of the untyped calculus of [2]. Unlike [2], however, we use a logic-programming style for the syntax of methods, so that methods may be written as formulas.

Objects

$\mathrm{O} ::=$	$s, x, y, \dots$	variables
$\mid$	`obj[ ]`	the empty object
$\mid$	$\mathrm{O}.m := \forall s\, \mathrm{M}$	method addition/override

Method Definitions

$\mathrm{M} ::=$	$\mathrm{A} \mapsto \mathrm{O}$ **if** B	conditional definition
$\mid$	$\mathrm{A} \mapsto \mathrm{O}$	atomic definition

$\mid$	$\forall \overline{x}\, \mathrm{M}$	quantified definition

Method Bodies

$\mathrm{B} ::=$	$\mathrm{O}.m\ \mathrm{A} \mapsto \mathrm{O}$	message send
$\mid$	B, B	conjunction

Argument List

$\mathrm{A} ::=$	$(\mathrm{O}_1, \ldots, \mathrm{O}_n)$	$n \geq 0$ arguments

Objects are formed as the result of a sequence of updates, starting from the empty object. An object update is written $\mathrm{O}.m := \forall s\, \mathrm{M}$, and represents the object obtained by either replacing the current method definition for the label m in O with the new definition $\forall s\, \mathrm{M}$, or by adding a new label m and associated definition. In either case the semantics of update is functional: first it produces a copy of the object being updated, and then updates (or extends) the copy. The object containing a given method is called the object's host object, and the quantified variable s represents the self-parameter for the method, to be bound to the host object upon invocation of that method (see below).

Method definitions have the form of clauses, where the *head* $\mathrm{A} \mapsto \mathrm{O}$ defines the correspondence between the input arguments A and the result O, and the *body* defines the conditions to be satisfied for this correspondence to hold. Method definitions may contain free variables, as long as each of these variables occur in the scope of a quantifier in the surrounding context. This generality, which is required to express objects and methods with "nested" occurrences of "self" (see Example 21) also justifies the explicit use of quantifiers in the syntax of methods.

Method bodies are formed as conjunctions of method invocations (message sends) of the form $\mathrm{O}_1.m\, A \mapsto \mathrm{O}_2$ whose intended semantics is as follows. Assume that the definition for m in O_1 is $\forall s\, \mathrm{M}$: to evaluate $\mathrm{O}_1.m\, A \mapsto \mathrm{O}_2$, evaluate $\mathrm{M}[\mathrm{O}_1/s]$, i.e., the method definition with the object O_1 bound to the self-parameter, using arguments A and expecting O_2 as a result.

Remarks. In the above productions we assume that the method labels m are chosen from a denumerable set of method labels $\mathcal{M}$, and that there are denumerably many names for variables.

Primitives for object extension are not available in the calculus of [2], whereas they are provided as a separate extension operator (different from the overriding operator) in [7]. An equivalent, extensional, presentation of objects could be adopted, where objects are defined as collections $\mathtt{obj}[m_1 = \forall s\, \mathrm{M}_1, \ldots, m_n = \forall s\, \mathrm{M}_n]$ of components $m_i = \forall s\, M_i$, with distinct labels $m_i \in \mathcal{M}$, and associated method definitions M_i, for $i \in 1..n$. The two views (i.e., the intentional view we have adopted, and the extensional one) could be unified by defining an equational theory over objects based on the following two axioms (the notation $\mathtt{obj}[m_i = \forall s\, \mathrm{M}_i]^{i \in 1..n}$ is short for $\mathtt{obj}[m_1 = \forall s\, \mathrm{M}_1, \ldots, m_n = \forall s\, \mathrm{M}_n]$):

(Eq-Override) $\quad (j \in 1..n)$

$$(\mathtt{obj}[m_i = \forall s\, \mathrm{M}_i]^{i \in 1..n}.m_j := \forall s\, \mathrm{M}) \quad = \quad (\mathtt{obj}[m_j = \forall s\, \mathrm{M}, m_i = \forall s\, \mathrm{M}_i]^{i \in 1..n, i \neq j})$$

(Eq-Extend) $\quad (j \notin 1..n)$

$$(\mathtt{obj}[m_i = \forall s\, \mathrm{M}_i]^{i \in 1..n}.m_j := \forall s\, \mathrm{M}) \quad = \quad (\mathtt{obj}[m_j = \forall s\, \mathrm{M}, m_i = \forall s\, \mathrm{M}_i]^{i \in 1..n})$$

Similar axioms are used in the equational theory of [2], and in the extended calculus studied in []. While defining an equational theory based on the above axioms would not be conceptually problematic, it is of no interest in the context of the present paper.

4.1 A Proof System

Unlike [2], where the semantics of the calculus is defined by reduction, evaluation in $\mathbf{Ob}_{\multimap}$ is a process of proof search, that we formalize in a natural semantics style with a set of inference rules. A goal, or query, is an existentially quantified conjunction of message sends:

$$\text{Query} \quad Q ::= \textsc{b} \mid \exists x.Q$$

Below we give a proof system for $\mathbf{Ob}_{\multimap}$, together with a few examples that should help clarify how the evaluation works.

In defining the proof rules, a mechanism is needed for extracting the method definitions that reside within objects. The following function serves this purpose.

- $\textsc{select}(m, (\textsc{o}.m := \forall s\, \textsc{m})) = \forall s\, \textsc{m}$;
- $\textsc{select}(n, (\textsc{o}.m := \forall s\, \textsc{m})) = \textsc{select}(n, \textsc{o})$ if $n \neq m$;

As we said, a goal in $\mathbf{Ob}_{\multimap}$ is a conjunction of message sends: since all the methods needed to handle the message reside within the receiver of that message, no "program" is really needed to evaluate a goal. Given the SELECT function defined above, the proof rules can be formalized as follows:

$$\frac{\vdash [\textsc{o}/x]Q \quad (\textsc{o}\ \mathit{closed})}{\vdash \exists x.Q}\ (\mathit{exist}) \qquad \frac{\vdash \textsc{b}_1 \quad \vdash \textsc{b}_2}{\vdash \textsc{b}_1, \textsc{b}_2}\ (\mathit{and}) \qquad \frac{\textsc{a} \mapsto \textsc{o} \in \langle \textsc{m} \rangle}{\textsc{m} \vdash \textsc{a} \mapsto \textsc{o}}\ (\mathit{initial})$$

$$\frac{\textsc{m}[\textsc{o}_1/s] \vdash \textsc{a} \mapsto \textsc{o}_2 \qquad \textsc{select}(m, \textsc{o}_1) = \forall s\, \textsc{m}}{\vdash \textsc{o}_1.m\ \textsc{a} \mapsto \textsc{o}_2}\ (\mathit{send})$$

$$\frac{\vdash \textsc{b} \qquad \textsc{a} \mapsto \textsc{o}\ \texttt{if}\ \textsc{b}\ \in \langle \textsc{m} \rangle}{\textsc{m} \vdash \textsc{a} \mapsto \textsc{o}}\ (\mathit{backchain})$$

These rules define evaluation in terms of two mutually recursive relations of provability: a principal, unary, relation that evaluates messages (and conjunctions thereof), and a subsidiary, binary, relation that accounts for the backchaining steps needed to evaluate a method definition. The notation $\langle \textsc{m} \rangle$ in the (*backchain*) and (*initial*) rules indicates the set of *closed* instances of M, i.e., the smallest set that satisfies the following conditions:

$$\textsc{m} \in \langle \textsc{m} \rangle;$$
$$\forall x\, \textsc{m}' \in \langle \textsc{m} \rangle \implies \textsc{m}'[\textsc{o}/x] \in \langle \textsc{m} \rangle \text{ for every closed object } \textsc{o}.$$

As a further remark, we note that the notion of provability we refer to here is strictly operational. In Section 5 we will show that this operational characterization has an equivalent formulation in terms of the notion of $\mathcal{L}$-provability.

4.2 Examples

When instrumented with a unification algorithm for computing bindings for the logical (i.e., existentially quantified) variables of a query, the rules given above can be directly employed as the core of an interpreter. In presenting the examples we use logical variables, (denoted by capital letters) instead of existentially quantified variables in queries, and we make implicit appeal to the existence of such unification algorithm (see Section 6 for further discussion).

The following shorthands are used in the example to ease readability. We write $\texttt{obj}[m_1 = \forall s\,\mathrm{M}_1, \ldots, m_n = \forall s\,\mathrm{M}_n]$ to denote the object that results from the sequence of extensions $(\ldots(\texttt{obj}[\,].m_1 := \forall s\,\mathrm{M}_1)\ldots).m_n := \forall s\,\mathrm{M}_n$. Also, we omit empty argument lists and write "$\mapsto \mathrm{O}$" instead of "$() \mapsto \mathrm{O}$"

Example 18 (Diagonal points) The first example represents a two-dimensional diagonal point with two fields, x, and y. In $\mathbf{Ob}_{\multimap}$, this object can be expressed as follows:

$$\texttt{d} \stackrel{\triangle}{=} \texttt{obj}[\texttt{x} = \forall s \mapsto 1, \texttt{y} = \forall s \forall v\ (\ \mapsto v \ \texttt{if}\ s.\texttt{x} \mapsto v)].$$

Both fields are encoded as methods: the x field is constant (it does not depend on the self variable s) whereas y is a "true" method, which does depend of the self variable.

Given a logical variable V, consider evaluating the query $\texttt{d.x} \mapsto V$. The evaluation takes two steps: a (*send*) step selects the definition of the method definition for x, and unifies the head of the definition with $\mapsto V$, producing the substitution $[1/V]$. A subsequent (*initial*) step terminates the process, and returns the substitution $[1/V]$ as result.

A similar sequence of steps is used to evaluate the query $\texttt{d.y} \mapsto V$. After the first (*send*) step, a (*backchain*) step leads to evaluating the body of the y method; this is just the message $\texttt{d.x} \mapsto V$, which is evaluated as before. The result is again the substitution $[1/V]$.

Example 19 (One-dimensional point) The second example is a one-dimensional point with a "move" method. This object can be represented as follows:

$$\texttt{pt} \stackrel{\triangle}{=} \texttt{obj}[\texttt{x} = \forall s \mapsto 3,\ \texttt{mv} = \forall s_1\, \mathrm{M}_{\texttt{mv}}[s_1]],$$

where

$$\mathrm{M}_{\texttt{mv}}[s_1] \stackrel{\triangle}{=} \forall x, y\ ((y) \mapsto (s_1.\texttt{x} := \forall s_2 \mapsto x{+}y)\ \ \texttt{if}\ \ s_1.\texttt{x} \mapsto x).$$

Consider the message send $\texttt{pt.mv}(2) \mapsto P$. As in the previous example, we first select the definition associated with mv, perform the self-substitution $\mathrm{M}_{\texttt{mv}}[\texttt{pt}/s_1]$, and continue evaluating the query $(2) \mapsto P$. Backchaining over $\mathrm{M}_{\texttt{mv}}[\texttt{pt}/s_1]$, produces the substitution

$$[2/y, (\texttt{pt.x} :=\forall s_1 \mapsto x{+}2)/P].$$

Evaluating the body of the definition yields then the new substitution $[3/x]$ and, composing the two substitutions, P gets bound to $\texttt{pt.x} :=\forall s_1 \mapsto 3{+}2$, which is just the object:

$$\texttt{obj}[\texttt{x} = \forall s \mapsto 3{+}2,\ \texttt{mv} = \forall s_1\, \mathrm{M}_{\texttt{mv}}[s_1]].$$

Example 20 (Object-based Inheritance) The next example illustrates method inheritance between objects. Consider again the pt object of the previous example, and let $\mathrm{M}_{\texttt{col}}$ be the definition $\forall s \mapsto blue$. Now, consider the following composite query:

$$(\texttt{pt.col} := \mathrm{M}_{\texttt{col}}).\texttt{move}(2) \mapsto P,\ P.\texttt{col} \mapsto C.$$

As in the previous example, evaluating the first message yields the binding

$$[(\texttt{pt.x} := \forall s_1 \mapsto x{+}2)/P],$$

and then the second message returns $[blue/C]$ as expected.

Example 21 (Object numerals) As a further example, we show how natural numbers can be represented in $\mathbf{Ob}_{\multimap}$. Following [2], 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.

$$\texttt{obj}[\texttt{is\_zero} = \forall s \mapsto true,\ \texttt{pred} = \forall s \mapsto s,\ \texttt{succ} = \forall s \mapsto \mathrm{O}[s]]$$

where $\mathrm{O}[s]$ is the following object:

$$\mathrm{O}[s] = (s.\texttt{is\_zero} := (\forall s' \mapsto false)).\texttt{pred} := \forall s' \mapsto s.$$

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 the self-parameter: 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.

Example 22 (Representing Classes) As a final example we show that classes and class-based inheritance can be represented in a fairly natural way in $\mathbf{Ob}_{\multimap}$. The representation we outline below differs from the *record-of-pre-methods* model of [1]: instead, it is inspired to that of [8] where object extension is used in an essential way to render the effect of class inheritance. A class is represented as an object like the following:

$$class_A \triangleq \texttt{obj}[\texttt{new} = \forall s, args_A\,(args_A) \mapsto \texttt{obj}[\ldots, \texttt{m}_\texttt{i} = \forall s_i\, body(args_A), \ldots]].$$

$class_A$ consists of just one method, the *constructor* function new: a call to the constructor on the class creates an instance by initializing the constructor parameters with corresponding arguments passed along with the call. A subclass $class_B$ of $class_A$ may then be defined by inheritance as follows, using method addition or override:

$$class_B \triangleq \texttt{obj}[\texttt{new} = \forall s, args_{AB}\,(args_{AB}) \mapsto (class_A.new(args_A)).\texttt{m} := \forall s\, body(args_B)].$$

Here m may either be a new method not present in (instances of) $class_A$, or an existing method that is overridden in (instances of) the subclass $class_B$.

5 Encoding $\mathbf{Ob}_{\multimap}$ in $\mathcal{L}$

In this section we show that $\mathcal{L}$ is well suited as a specification language for $\mathbf{Ob}_{\multimap}$. We do this by first defining an encoding function that maps object expressions and queries in $\mathbf{Ob}_{\multimap}$ into, respectively, $\mathcal{L}$-terms and G-formulas from $\mathcal{L}$; then we show that the SELECT function as well as the (*send*) inference rule used in the proof system of $\mathbf{Ob}_{\multimap}$ can be axiomatized as an $\mathcal{L}$-theory (i.e., as a set of closed D-formulas). Finally, we show that evaluating an $\mathbf{Ob}_{\multimap}$ query corresponds to finding an $\mathcal{L}$-proof for the encoding of that query in $\mathcal{L}$.

The intuition behind the encoding is rather straightforward: objects from $\mathbf{Ob}_{\multimap}$ are encoded in $\mathcal{L}$ as lists of pairs (*method label, method definition*), while method definitions are encoded as D-formulas from $\mathcal{L}$. Care must be used only to ensure that $\mathcal{L}$-terms of appropriated sorts are chosen to represent object-expressions from $\mathbf{Ob}_{\multimap}$. Since $\mathbf{Ob}_{\multimap}$ is untyped, this is easily accomplished by choosing one of the types of $\mathcal{L}$ as the universal type for all the object expressions from $\mathbf{Ob}_{\multimap}$: we denote this type with ω in the following. The details of the encoding are described below: we define it in terms of four mutually recursive functions, one for each syntactic category of $\mathbf{Ob}_{\multimap}$.

Encoding of Objects $[\![\,\mathrm{O}\,]\!]^1 : \omega$.
Choose a type μ as the type of method labels from $\mathcal{M}$. Then:

- choose a type π to represent the type of values built as pairs (m, D) where m is a method label, D is a D-formula in $\mathcal{L}$, and $(\cdot,\cdot) : \mu \times \mathsf{o} \to \pi$ is a non-logical constant;
- choose a type ω to represent the type of every list of π's, and two non-logical constants, $[\,] : \omega$ and $:: \pi \times \omega \to \omega$ to represents the constructors of lists.

Given these choices, the encoding of objects can be defined as follows:

- $[\![\,x\,]\!]^1 = x$;
- $[\![\,\mathtt{obj}[\,]\,]\!]^1 = [\,]$;
- $[\![\,\mathrm{O}.m := \forall s\,\mathrm{M}\,]\!]^1 = (m, \forall_\omega s\,[\![\,\mathrm{M}\,]\!]^2_{m,s}) :: [\![\,\mathrm{O}\,]\!]^1$

Having chosen ω as the universal type for the terms of $\mathbf{Ob}_{\multimap}$, the untyped quantifiers from $\mathbf{Ob}_{\multimap}$ is represented by the typed quantifier $\forall_\omega$ from $\mathcal{L}$.

Encoding of Method Definitions $[\![\,\mathrm{M}\,]\!]^2_{m,\mathrm{SELF}} : \mathsf{o}$.
As anticipated, method definitions are encoded as D-formulas in $\mathcal{L}$. The encoding function is indexed by a method label and by the self variable to allow a proper encoding of the head of the definition. Again, the untyped quantifiers from $\mathbf{Ob}_{\multimap}$ are represented by the typed quantifier $\forall_\omega$. The definition uses a further type, α, to represent the type of arguments (lists of objects), and of a non-logical constant $\mathtt{meth} : \mu \times \omega \times \alpha \times \omega \to \mathsf{o}$ from the signature.

$$\begin{aligned}
[\![\,\forall x\,\mathrm{M}\,]\!]^2_{m,\mathrm{SELF}} &= \forall_\omega x\,[\![\,\mathrm{M}\,]\!]^2_{m,\mathrm{SELF}};\\
[\![\,\mathrm{A} \mapsto \mathrm{O}\ \mathtt{if}\ \mathrm{B}\,]\!]^2_{m,\mathrm{SELF}} &= [\![\,\mathrm{B}\,]\!]^3 \Rightarrow \mathtt{meth}\ m\ \mathrm{SELF}\ [\![\,\mathrm{A}\,]\!]^4\ [\![\,\mathrm{O}\,]\!]^1\\
[\![\,\mathrm{A} \mapsto \mathrm{O}\,]\!]^2_{m,\mathrm{SELF}} &= \mathtt{meth}\ m\ \mathrm{SELF}\ [\![\,\mathrm{A}\,]\!]^4\ [\![\,\mathrm{O}\,]\!]^1.
\end{aligned}$$

Encoding Method Bodies. $[\![\,\mathrm{B}\,]\!]^3 : \mathtt{o}$.
Messages are represented as corresponding atomic formulas from $\mathcal{L}$: the definition uses the non-logical constants $\mathtt{send} : \omega \times \mathtt{o} \to \mathtt{o}$ to construct an atomic formula out of a message send, and $\mathtt{msg} : \mu \times \alpha \times \omega \to \mathtt{o}$ to form the message. The comma operator from $\mathbf{Ob}_{\multimap}$ is interpreted in $\mathcal{L}$ as &.

$$\begin{array}{rcl} [\![\,\mathrm{O}.m\ \mathrm{A} \mapsto \mathrm{O}\,]\!]^3 & = & \mathtt{send}\ [\![\,\mathrm{O}\,]\!]^1(\mathtt{msg}\ m\ [\![\,\mathrm{A}\,]\!]^4\ [\![\,\mathrm{O}\,]\!]^1) \\ [\![\,\mathrm{B}_1, \mathrm{B}_2\,]\!]^3 & = & [\![\,\mathrm{B}_1\,]\!]^3 \,\&\, [\![\,\mathrm{B}_2\,]\!]^3. \end{array}$$

Encoding of Arguments. $[\![\,\mathrm{A}\,]\!]^4 : \alpha$.
Arguments are encoded as lists of objects, choosing a type α, and two non-logical constants $() : \alpha$, and $(\cdot,\cdot) : \omega \times \alpha \to \alpha$ from the signature.

$$\begin{array}{rcl} [\![\ ()\ ]\!]^4 & = & (); \\ [\![\,(\mathrm{O}_1, \ldots, \mathrm{O}_n)\,]\!]^4 & = & ([\![\,\mathrm{O}_1\,]\!]^1, [\![\,\mathrm{O}_2, \ldots, \mathrm{O}_n\,]\!]^4). \end{array}$$

Specification of the operational semantics. We first define the *selection D*-formulas, used to extract methods from objects. The definition uses the non-logical constant $\mathtt{select} : \omega \times \mu \times \mathtt{o} \to \mathtt{o}$ from the signature. Given the encoding of objects as lists of pairs (*method label, method definitions*), the definition should be clear if one thinks of D as the encoding of a method definition.

$$\forall_\mu \mathrm{M}.\ \forall_\omega \mathrm{O}.\ \forall_{\mathtt{o}} \mathrm{D}.\ \mathtt{select}\ \ ((\mathrm{M}, \mathrm{D})\ ::\ \mathrm{O})\ \mathrm{M}\ \mathrm{D}.$$

$$\forall_\mu \mathrm{M}.\ \forall_\mu \mathrm{N}.\ \forall_\omega \mathrm{O}.\ \forall_{\mathtt{o}} \mathrm{D}.\ \mathrm{M} \neq \mathrm{N}\ \&\ \mathtt{select}\ \mathrm{O}\ \mathrm{M}\ \mathrm{D} \Rightarrow \mathtt{select}\ \ ((\mathrm{N}, \_)\ ::\ \mathrm{O})\ \mathrm{M}\ \mathrm{D}.$$

Here $\mathrm{M} \neq \mathrm{N}$ denotes that the two labels are different. Then we define the *evaluation D*-formula that renders the effect of self-substitution.

$$\begin{array}{l} \forall_\mu \mathrm{M}.\ \forall_\omega \mathrm{O}_1.\ \forall_\omega \mathrm{O}_2.\ \forall_\alpha \mathrm{A}. \\ \quad \exists_{\mathtt{o}} \mathrm{D}.(\mathtt{select}\ \ \mathrm{O}_1\ \mathrm{M}\ \mathrm{D}\ \&\ (\mathrm{D}\ \multimap\ (\mathtt{meth}\ \mathrm{M}\ \mathrm{O}_1\ \mathrm{A}\ \mathrm{O}_2)))\ \Rightarrow\ \mathtt{send}\ \ \mathrm{O}_1\ (\mathtt{msg}\ \mathrm{M}\ \mathrm{A}\ \mathrm{O}_2). \end{array}$$

It is worth taking the time to show why this captures the intended semantics of method invocation. Let Γ_{eval} be the set of evaluation and selection D-formulas we introduced above; let then *obj* be the encoding of an object containing a definition for the label m, and let *arg* be the encoding of a list of arguments. Now consider the following $\mathcal{L}$-sequent:

$$\Gamma_{eval} \longrightarrow \exists_\omega \mathrm{O}.\mathtt{send}\ \ obj\ (\mathtt{msg}\ m\ arg\ \mathrm{O}).$$

Considering how a (uniform) $\mathcal{L}$-proof could be constructed for this sequent, one easily sees that the self-substitution semantics is rendered correctly. The sequence of proof steps can be described as follows:

1. Apply $(\exists_r)$ to find a suitable $\mathcal{L}$-term, say *val*, as a witness for O;
2. Apply (*decide*) to move the *evaluation D*-formula in the bounded part of the context; then apply four $(\forall_l)$ steps to substitute m, *obj*, *arg* and *val*, respectively for the quantified variables M, O_1, A, and O_2;

3. Apply ($\Rightarrow_l$): the right sequent on the premises is initial; the left-sequent is

$$\Gamma_{eval} \longrightarrow \exists_\circ \text{D}.(\texttt{select}\ \ obj\ m\ \text{D}\ \&\ (\text{D}\ \multimap\ (\texttt{meth}\ m\ obj\ arg\ val))).$$

4. Now apply ($\exists_r$) to find a D-formula D_m as substitute for D such that the sequent $\Gamma \longrightarrow \texttt{select}\ \ obj\ m\ \text{D}_m$ is provable, and then consider the sequent

$$\Gamma \xrightarrow{\text{D}_m} \texttt{meth}\ m\ obj\ arg\ val,$$

that results from applying ($\multimap_r$) on $\Gamma \longrightarrow \text{D}_m\ \multimap\ (\texttt{meth}\ m\ obj\ arg\ val)$.

5. It is at this point that the self-substitution takes place: looking at the encoding of method definitions, one notices that D_m is a D-formula of the form:

$$\forall_\omega \text{SELF}.\forall \ldots (G \Rightarrow \texttt{meth}\ m\ \text{SELF}\ \text{A}\ \text{O}),$$

for suitable choices of the variables A and O, and of the G-formula G encoding the method body. Then, to continue the proof, more applications of ($\forall_l$) are needed that substitute the receiver of the message, i.e., the object *obj*, for the SELF parameter in the method definition and the parameters.

6. Finally, after performing the required ($\forall_l$) steps, the proof continues on the sequent $\Gamma \longrightarrow G$. New method definitions will be made available as bounded resources as needed in the evaluation of new message-sends: since objects are non identified, this ability to consume methods after their selection is crucial to avoid conflicts among methods of different objects.

We can now give the main result of this section.

Theorem 23 (Adequacy of the Encoding) Let $\exists\overline{x}.\text{B}$ be query in $\mathbf{Ob}_{\multimap}$, where B is a conjunction of message sends, and $\overline{x}$ is the list of the free variables of B, and let $G = [\![\, \text{B} \,]\!]^3$ be the G-formula that encodes B in $\mathcal{L}$. Then $\vdash \exists\overline{x}.\text{B}$ has a proof in $\mathbf{Ob}_{\multimap}$ if and only if the $\mathcal{L}$-sequent $\Gamma_{eval} \longrightarrow \exists_\omega\overline{x}.G$ has an $\mathcal{L}$-proof.

Given the somewhat informal presentation of the encoding for $\mathbf{Ob}_{\multimap}$, we only state the theorem without giving a proof. The proof is intuitively simple, although time consuming, once we observe that the proof system $\Pi_{\mathcal{L}}$ may be equivalently formulated using a (*backchain*) rule as we have done in $\mathbf{Ob}_{\multimap}$. More precisely,

· the rules ($\&_l$), ($\Rightarrow_l$) and ($\forall_l$) of $\Pi_{\mathcal{L}}$ may be replaced by the rule

$$\frac{\Gamma \longrightarrow C \quad C \Rightarrow A \in \langle D \rangle}{\Gamma \xrightarrow{D} A}\ (bc)$$

· the axiom (*initial*) of $\Pi_{\mathcal{L}}$ may be replaced by the axiom

$$\frac{A \in \langle D \rangle}{\Gamma \xrightarrow{D} A}\ (init)$$

The notation $\langle D \rangle$ stands the set of *closed* $\mathcal{L}$-instances of D, i.e., the smallest set that satisfies the following conditions.

$$\begin{array}{rcl} D \in \langle D \rangle & & \\ B_1 \& B_2 \in \langle D \rangle & \Longrightarrow & B_1 \in \langle D \rangle \text{ and } B_2 \in \langle D \rangle \\ \forall x\, B \in \langle D \rangle & \Longrightarrow & B[t/x] \in \langle D \rangle \text{ for every closed } \mathcal{L}\text{-term } t \end{array}$$

The equivalence between the two formulations of $\Pi_{\mathcal{L}}$ follows directly from the fact that $\Pi_{\mathcal{L}}$ preserves *focusing* proofs as discussed in Section 2.

6 Implementation

In this section we describe a prototypical implementation of $\mathbf{Ob}_{\multimap}$ based on the system developed by Hodas in [11]. While the specification of $\mathbf{Ob}_{\multimap}$ in $\mathcal{L}$ given in the previous section could directly be implemented in Forum, an implementation of the Forum system is still subject of study (cf. [14]). Instead, the system described in [11] implements an extension of the first-order syntax of an intuitionistic fragment of Forum (called Lolli) that provides the sort of higher-order features distinctive of $\mathcal{L}$: specifically it allows variables of type `o` to occur nested within terms and in formula position, and it provides a unification algorithm for the resulting set of terms[3]. These two aspects make the task of implementing an interpreter for $\mathbf{Ob}_{\multimap}$ in that system immediate, given our choice of using algebraic terms in the definition of $\mathcal{L}$.

6.1 An interpreter for $\mathbf{Ob}_{\multimap}$

Refining the idea described in Section 5, we represent objects in the prototype by distinguishing fields from proper methods: an object is a pair `(Fields,Methods)`, where `Fields` is a list of pairs `(Field-Name,Value)` and `Methods` is a list of pairs `(Method-Name,Definition)`. We further distinguish methods into "functional" and "predicative": the former give a direct account of methods in $\mathbf{Ob}_{\multimap}$, the latter allow predicates to be represented as methods with a "null" return value. Functional methods are invoked using a "send" primitive, whereas predicative methods are invoked using a "call" predicate. Following [11], in the examples below we use `A <= B` to denote the implication $B \Rightarrow A$.

```
MODULE obj.            % Interpreter for the Core Language

 send Ob msg M P V <=  % method call
    Ob = (St,Mts)         &
    select Mts M Def      &
    generalize Def QDef   &
    QDef  -o  (meth Ob P V).

 call Ob msg M P <=    % predicate call
    Ob = (St,Mts)       &
    select Mts M Def    &
```

[3]The unification algorithm is essentially first-order, as unification of quantified formulas can be carried out, modulo α-conversion, treating bound variables as constants.

```
    generalize Def QDef &
    QDef  -o  (pred Ob P).

 get (St,Mts) M V <=    % access a field
    select St M V.

 f_update (St,Mts) M V (NewSt,Mts) <=    % update a field
   replace St M V NewSt.

 f_extend (St,Mts) M V (NewSt,Mts) <=    % add a new attribute
   add St M V NewSt.

 m_override (St,Mts) M Def (St,NewMts) <= % override a method definition
   replace Mts M Def NewMts.

 m_extend (St,Mts) M Def (St,NewMts) <=   % to add a method definition
   add Mts M Def NewMts.
```

Method and field addition and override are realized by calls to corresponding primitive predicates defined by the interpreter. The definitions of `send` and `call` use the builtin predicate `generalize` available in the system of [11]: the call `generalize(Def,QDef)` takes the formula `Def` with free variables $\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}$ and returns the formula `QDef` adding universal quantifiers over $\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}$. The use of `generalize` could be objected to, as it is highly "extra logical"; however, its use here is motivated solely by the desire to simplify the notation of methods: calls to this primitive could be avoided using explicit quantification in the syntax of methods.

6.2 Logic Programming with $\mathbf{Ob}_{\multimap}$

The implementation $\mathcal{L}$ we just outlined supports a very natural combination of object-oriented and logic programming styles of computations: as we noted in Section 1, this combination is one of the payoffs of our approach. In the following examples $\mathcal{L}$-programs are used to specify classes and objects, in the style of the "Logic and Objects" language of [17], as well as relations (over such classes and objects) in the traditional logic programming style.

The encoding of classes given here differs from the encoding given in Example 22: in that case, classes were encoded as $\mathbf{Ob}_{\multimap}$ objects, while here they are encoded outside $\mathbf{Ob}_{\multimap}$ and inside $\mathcal{L}$ instead: specifically, we now define classes using atomic formulas that describe collections of objects with a common pattern: objects may then be create by instantiation. A class declaration has the form:

```
class <ClassName> <FList> <MList>
```

where `<FList>` introduces the field names of objects of the class, together with their initial values, and `MList` is a list of method names. Method names are associated with their definitions by method (predicate) definitions of the form:

```
meth <ClassName>::<MethName> Self <Parameters> <Result> (<Body>)
pred <ClassName>::<MethName> Self <Parameters> (<Body>).
```

Instances of a class are generated by calls to the primitive `new`, that creates a clone of the field-list and installs the method definitions within the clone. The primitive `new` is defined below (`A -> B | C` is short for the standard `if-then-else` predicate):

```
new Class (St,Mts) <=
   (class Class St MtNs) -> true |
      (nl & write_sans("Error: Class ") &
          write(Class) &
          write_sans(" is not defined") &
       nl & fail) &
   buildMts Class (MtNs) Mts.
```

The predicate `buildMts` simply *compiles* methods written in the high-level syntax according to the encoding presented in the previous section.

```
buildMts Class (Name::R) ((Name,QDef)::Mts):-
   meth Name X Ps V Body ,
   QDef = ((m X Ps V) <= Body),
   buildMts Class R Mts.

buildMts Class (Name::R) ((Name,QDef)::Mts):-
   pred Name X Ps Body,
   QDef = ((p X Ps) <= Body),
   buildMts Class R Mts.

buildMts Class nil nil.
```

Note that the definition of methods (e.g. `QDef` in the code above) consists of formulas with *free variables*. As mentioned before, methods are transformed in universally quantified formulas at the time when `call` and `send` messages are evaluated.

Class Inheritance. Class-based inheritance can be accounted for without further primitives. As an example, we define two classes, a class `person`, with two methods, and a class `employee` that inherits from `person` and defines a new method `check` that verifies whether the salary of an employee is less then the salary of her/his manager.

```
MODULE People.

 class person ((name,nil)::(salary,nil)::nil) (write::init::nil).

 class employee ((manager,nil)::St) (check::Mts) <=
   class person St Mts.

 pred employee check Self nil (
   get Self manager Man  &
   get Man salary MS     &
   get Self salary S     &
   MS > S -> write("Ok") | write("Suspect").
```

Object Update. The support for method update at the level of objects is useful in several situations (cf. [1]). One such situation arises when dealing with exceptions in a data scheme: in this case, method updates allow the designer to handle the exception locally to the "exceptional" object, without need of re-designing the scheme. Assume, for instance, that a given employee is granted an extra over his standard salary. To model this fact, we add a new field, `extras`, to the set of attributes of that employee, and override its `check` method as shown below

```
newdef ((pred Self nil) <= (get Self manager Man &
                            get Man salary MS    &
                            get Self salary S    &
                            get Self extras E    &
                            Tot is S+E           &
                            MS > Tot -> write("Ok")
                               | write("Suspect") &
                            nl)).
modify Emp MoreMoney <=
    newdef D &
    f_extend Emp extras MoreMoney NEmp &
    m_override NEmp check D NewEmp.
```

Here a query like `(modify emp 1000)` modifies the employee `emp` with a new field and a new definition of the `check` method: the new definition is built in two steps, using the auxiliary predicate `newdef` only for convenience of notation.

7 Discussion

The encoding of $\mathbf{Ob}_{\multimap}$ of Section 5 shows that $\mathcal{L}$ is very effective in specifying the object-oriented features encompassed by $\mathbf{Ob}_{\multimap}$. The key ingredients of the encoding are the presence of quantifiers over formulas and the use of nested implications with variable D-formulas as antecedents. The coexistence of these two features allows methods that are embedded within objects to be first dynamically loaded to respond to the corresponding messages, and then immediately consumed, thus guaranteeing the absence of conflicts between methods of different objects. The use of algebraic terms (as opposed to λ-terms) to encode methods embedded within objects has itself practical interest, as it allows us to rely on an essentially first-order unification algorithm in the implementation of the language.

7.1 Quantification over Formulas and higher-order programming

Combinations of variable D-formulas and nested implications similar to those used in $\mathcal{L}$ may be used to specify other and more general forms of higher-order programming. Consider, to this regard, the fragment $\mathcal{M}$ of intuitionistic logic defined by the following extension of Horn clauses with nested implication:

$$\begin{array}{lcl} M & ::= & A \mid G \supset A \mid M \wedge M \mid \forall_\tau V.M \\ G & ::= & true \mid A \mid G \wedge G \mid \exists_\tau V.G \mid M^{\mathrm{V}} \supset A \\ M^{\mathrm{V}} & ::= & M \mid V \mid M^{\mathrm{V}} \wedge M^{\mathrm{V}}. \end{array}$$

These productions define a higher-order extension of the modular logic language proposed by Miller in [18]. Similarly to $\mathcal{L}$, variable formulas are allowed to occur as M-formulas in the antecedents of nested implications, whereas they are forbidden as G-formulas. Defining $\mathcal{M}$-terms in ways similar to how we defined $\mathcal{L}$-terms (this may be done by simply using the intuitionistic connectives in place of the corresponding linear connectives in Definition 1), the same technique we have devised to prove completeness of $\mathcal{L}$-proofs, may be used to show completeness of uniform (intuitionistic) proofs for the fragment $\mathcal{M}$.

The interest in M-formulas derives from considering formulas like the following:

$$\forall_{\mathsf{o}}\mathrm{M}.(\mathrm{M} \supset g) \supset f(\mathrm{M}).$$

This is a legal M-formula, which allows a direct specification of computations where modules are used as parameters and dynamically bound to module "values" during the computation.

7.2 $\mathcal{M}$-formulas vs higher-order hereditary Harrop formulas

Another powerful higher-order feature that can be expressed with $\mathcal{M}$-formulas derives from the possibility of describing computations that build programs used by subsequent computations. A typical example of such situation is the goal formula $\exists_{\mathsf{o}}\mathrm{P}.(\mathtt{compile}\ m\ \mathrm{P})\ \wedge\ (\mathrm{P} \supset g)$, that we borrow from [21]. This formula is legal for any (specific) M-formula m and G-formula g, and may be thought of as describing a computation that first compiles the term m into a program P and then uses the resulting program to solve the goal g.

As noted in [21] formulas like the one above are not legal *hohh* formula, as only rigid formulas may occur as antecedents of a nested implication in the language *hohh*. On the other hand, *hohh* formulas are more powerful than $\mathcal{M}$-formulas in other respects, as they allow occurrences of variable G formulas, which are instead forbidden in $\mathcal{M}$. There is, in fact, a more fundamental difference between $\mathcal{M}$ (equivalently $\mathcal{L}$) formulas and the *hohh* formulas of [21], that lies in the structure of terms. Specifically *hohh* terms may be formed as simply typed λ-terms, but are then restricted so as to rule out terms containing occurrences of (intuitionistic) implication; on the other hand, $\mathcal{M}$-terms are algebraic terms where we do allow such occurrences of implication.

7.3 Desirable Extensions

While it would be desirable to be more liberal in the definition of $\mathcal{L}$-terms, so as to allow λ-terms, there seems to be a fundamental tradeoff between the use of λ-terms and uses of implications within such terms.

As discussed in [21] lifting the restriction on the occurrences of implications within *hohh*-terms would break the completeness of uniform proofs. Similarly, allowing arbitrary λ-terms in $\mathcal{M}$ could not be accounted for easily without losing completeness of the proof procedure. The reason is that the $\mathcal{L}$-normalization mapping ν, as we have defined it, does not generally commute with β-conversion; this is problematic since substituting λ-terms for variables with higher-order types may introduce β-redexes in the resulting terms, even though the substitution λ-terms are in normal form. A simple example is the following: take a variable $x : \mathsf{o} \to \mathsf{o}$, a constant $g : \mathsf{o}$ and form the

λ-normal term $(x\ g)$. Now consider substituting x with the abstraction $\lambda y : \mathsf{o}.y \Rightarrow a$, where a is a constant type o. The resulting term β-converts to the D-formula $g \Rightarrow a$. On the other hand, the result of normalizing $\lambda y : \mathsf{o}.y \Rightarrow a$ may (at best) be defined as the term $\lambda y : \mathsf{o}.\nu_{\mathsf{o}}$, as $y \Rightarrow a$ is not a D-formula: consequently, the result of reducing $(x\ g)[\lambda y : \mathsf{o}.\nu_{\mathsf{o}}/x]$ is ν_{o}.

Situations like one just described may not arise in the simplified setting based on algebraic terms we have devised. The reason is easily seen when one considers that every legal substitution λ-term in the algebraic setting is required to have basic types: hence, in particular, it may not be a λ-abstraction.

A possible extension to the notion of terms would be to allow a restricted form of λ-terms so as to ensure that β-redexes of the form $(\lambda x : \tau.a)b$ are always formed around types τ other than o. This would guarantee the desired properties of $\mathcal{L}$-normalization, and hence make the completeness proof go through. More work seems to be needed, however, to understand how this or other extensions could be accommodated, and to estimate how they would affect the expressiveness of the fragment $\mathcal{L}$.

8 Related Work

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 [4], the linear logic language *F&O* [6] and the the *HACL* language of [15]. Below we discuss relations with these proposals.

In *LO*, objects are represented as $\parr$-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 $\mathbf{Ob}_{\multimap}$, instead, objects are encoded as terms that encapsulate their methods as subterms. With our encoding, while retaining the form of method inheritance distinctive of [4], we also obtain a natural modeling of dynamic method redefinition, a functionality that could hardly be accounted for in [4].

The encoding of objects in $\mathbf{Ob}_{\multimap}$ is inspired to the language *F&O* of [6], 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 [15]. *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, $\mathbf{Ob}_{\multimap}$ can be viewed, essentially, as 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-substitution semantics of method invocation rather than on the explicit use of fixed-point operators.

Acknowledgments

Comments and suggestions by the anonymous referees are gratefully acknowledged: they were very helpful in improving the presentation of the paper. The revised version of this paper was written while the first author was visiting the Max-Planck-Institut für Informatik. He would like to thank Prof. Harald Ganzinger and the Computational Logic Group for financial support and for the ideal working conditions they provided.

References

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

[2] M. Abadi and L. Cardelli. A Theory of Primitive Objects. *Information and Computation*, 125(2):78–102, 1996.

[3] M. Abadi, L. Cardelli, and R. Viswanathan. An Interpretation of Objects and Object Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 396–409. ACM Press, 1996.

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

[5] M. Bugliesi, G. Delzanno, L. Liquori, and M. Martelli. A Linear Logic Calculus of Objects. In Michael Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 67–81. The MIT Press, 1996.

[6] G. Delzanno and M. Martelli. Objects in Forum. In John W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 115–129. The MIT Press, 1995.

[7] 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.

[8] K. Fisher and J. C. Mitchell. On the Relationship between Classes, Objects, and Data Abstraction. *Theory and Practice of Object Systems* 4(1):3–32, 1998.

[9] J. Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[10] J. A. Harland and D. J. Pym. The Uniform Proof-theoretical Foundation of Linear Logic Programming (Extended Abstract). In Vijay A. Saraswat and Kazunori Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 304–318. The MIT Press, 1991.

[11] J. Hodas. *Logic Programming in Intuitionistic Linear Logic.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[12] J. Hodas and D. Miller. Representing Objects in a Logic Programming Language with Scoping Constructs. In David H. Warren and Peter Szeredi, editors, *Proceedings of 7th International Conference on Logic Programming*, pages 511–526. The MIT Press, 1990.

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

[14] J. Hodas and J. Polakow. Forum as a Logic Programming Language (Preliminary Report). *Electronic Notes in Theoretical Computer Science*, 3, 1996.

[15] N. Kobayashi and A. Yonezawa. Type-Theoretic Foundations for Concurrent Object-Oriented Programming. In *Proceedings of the Ninth ACM-SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 31–45, 1994.

[16] L. Liquori. An Extended Theory of Primitive Objects: First Order System. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 146–169. Springer-Verlag, 1997.

[17] F. G. McCabe. *Logic and Objects*. International Series in Computer Science. Prentice Hall, 1992.

[18] D. Miller. A Logical Analisys of Modules in Logic Programming. *Journal of Logic Programming*, 6:79–108, 1989.

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

[20] D. Miller. Forum: A Multiple-Conclusion Meta-Logic. *Theoretical Computer Science*, pages 110(1):201–232, 1996.

[21] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[22] G. Nadathur and D. Miller. An Overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the International Conference Symposium on Logic Programming*, pages 810–827, 1988.

[23] D. J. Pym and J.A. Harland. A Uniform Proof-Theoretical Investigation of Linear Logic Programming. *Journal of Logic and Computation*, 4(2):175–207, 1994.

A Forum: a Uniform Proof System for Linear Logic

This part is taken from [20]. The syntax of Forum consists of types and simply typed λ-terms. Formulas are built over the following logical connectives: $\top$, $\bot$, $\multimap$, $\Rightarrow$, $⅋$, $\&$ and $\forall$. Note that left-rules can be applied only to sequents with right-hand side consisting of a list of *atomic* formulas. The following notation is used: Σ is a signature, Γ, Δ and Υ are multisets of formulas, Ω is a list of formulas, and $\mathcal{A}$ a list of atomic formulas. The meaning of "," is overloaded to stand for set or multiset union, depending on the context where it is used; $\mathcal{A}_1 + \mathcal{A}_2$ denotes the list resulting by merging the two lists $\mathcal{A}_1$ and $\mathcal{A}_2$. We identify $\alpha{-}\beta$-convertible formulas.

$$\frac{\Sigma : \Gamma; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon}{\Sigma : \Gamma; \Delta, B \longrightarrow \mathcal{A}; \Upsilon}\ (decide_\Delta) \qquad \frac{\Sigma : \Gamma, B; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon}{\Sigma : \Gamma, B; \Delta \longrightarrow \mathcal{A}; \Upsilon}\ (decide_\Gamma)$$

$$\frac{}{\Sigma : \Gamma; \emptyset \xrightarrow{A} A; \Upsilon}\ (initial_1) \qquad \frac{}{\Sigma : \Gamma; \emptyset \xrightarrow{A} \emptyset; A, \Upsilon}\ (initial_2)$$

$$\frac{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B; B, \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}; B, \Upsilon}\ (decide?) \qquad \frac{}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, \top, \Omega; \Upsilon}\ (\top_r)$$

$$\frac{}{\Sigma : \Gamma; \emptyset \xrightarrow{\bot} \emptyset; \Upsilon}\ (\bot_l) \qquad \frac{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, \Omega; \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, \bot, \Omega; \Upsilon}\ (\bot_r)$$

$$\frac{\Sigma : \Gamma; B \longrightarrow \emptyset; \Upsilon}{\Sigma : \Gamma; \emptyset \xrightarrow{?B} \emptyset; \Upsilon}\ (?_l) \qquad \frac{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, \Omega; B, \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, ?B, \Omega; \Upsilon}\ (?_r)$$

$$\frac{\Sigma : \Gamma; \Delta \xrightarrow{B_i} \mathcal{A}; \Upsilon \quad i \in \{1, 2\}}{\Sigma : \Gamma; \Delta \xrightarrow{B_1 \,\&\, B_2} \mathcal{A}; \Upsilon}\ (\&_l) \qquad \frac{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B, \Omega; \Upsilon \quad \Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, C, \Omega; \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B \,\&\, C, \Omega; \Upsilon}\ (\&_r)$$

$$\frac{\Sigma : \Gamma; \Delta_1 \xrightarrow{B} \mathcal{A}_1; \Upsilon \quad \Sigma : \Gamma; \Delta_2 \xrightarrow{C} \mathcal{A}_2; \Upsilon}{\Sigma : \Gamma; \Delta_1, \Delta_2 \xrightarrow{B \parr C} \mathcal{A}_1 + \mathcal{A}_2; \Upsilon}\ (\parr_l) \qquad \frac{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B, C, \Omega; \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B \parr C, \Omega; \Upsilon}\ (\parr_r)$$

$$\frac{\Sigma : \Gamma; \Delta_1 \longrightarrow \mathcal{A}_1, B; \Upsilon \quad \Sigma : \Gamma; \Delta_2 \xrightarrow{C} \mathcal{A}_2; \Upsilon}{\Sigma : \Gamma; \Delta_1, \Delta_2 \xrightarrow{B \multimap C} \mathcal{A}_1 + \mathcal{A}_2; \Upsilon}\ (\multimap_l) \qquad \frac{\Sigma : \Gamma; \Delta, B \longrightarrow \mathcal{A}, C, \Omega; \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B \multimap C, \Omega; \Upsilon}\ (\multimap_r)$$

$$\frac{\Sigma : \Gamma; \emptyset \longrightarrow B; \Upsilon \quad \Sigma : \Gamma; \Delta \xrightarrow{C} \mathcal{A}; \Upsilon}{\Sigma : \Gamma; \Delta \xrightarrow{B \Rightarrow C} \mathcal{A}; \Upsilon}\ (\Rightarrow_l) \qquad \frac{\Sigma : \Gamma, B; \Delta \longrightarrow \mathcal{A}, C, \Omega; \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B \Rightarrow C, \Omega; \Upsilon}\ (\Rightarrow_r)$$

$$\frac{\text{t:}\tau \text{ is a } \Sigma\text{-term} \quad \Sigma : \Gamma; \Delta \xrightarrow{B[t/x]} \mathcal{A}; \Upsilon}{\Sigma : \Gamma; \Delta \xrightarrow{\forall_\tau x:\tau.B} \mathcal{A}; \Upsilon}\ (\forall_l) \qquad \frac{y : \tau, \Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B[y/x], \Omega; \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, \forall x_\tau.B, \Omega; \Upsilon}\ (\forall_r)$$

A rule for existentially quantified formulas can be derived by considering the family of non-logical constants $\exists_\tau : (\tau \to o) \to o$ and enriching Γ with the following definition:

$\forall_\tau x.(Bx) \multimap (\exists_\tau\ B)$:

$$\frac{\text{t:}\tau \text{ is a } \Sigma\text{-term} \qquad \Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, B[t/x], \Omega; \Upsilon}{\Sigma : \Gamma; \Delta \longrightarrow \mathcal{A}, (\exists_\tau x.B), \Omega; \Upsilon} \quad (\exists_r)$$