Math. Struct. in Comp. Science (1999), *vol.* 9, *pp.* 253–286. Printed in the United Kingdom
© 1999 Cambridge University Press

A specification logic for concurrent object-oriented programming

G. DELZANNO[†], D. GALMICHE[‡] and M. MARTELLI[§]

[†] *Max Planck Institut für Informatik, Im Stadtwald, Gebaude 46.1*
66123 Saarbrücken, Germany.
Email: delzanno@mpi-sb.mpg.de

[‡]*LORIA UMR 7503 - UHP Nancy 1, Campus Scientifique - B.P. 239*
54506 Vandœuvre-lès-Nancy Cedex, France.
Email: galmiche@loria.fr

[§]*Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova,*
Via Dodecaneso, 35, I-16146 Genova, Italy,
Email: martelli@disi.unige.it

Received 10 December 1997; revised 12 December 1998

This paper focuses on the use of linear logic as a specification language for the operational semantics of advanced concepts of programming such as concurrency and object-orientation. Our approach is based on a refinement of linear logic sequent calculi based on the proof-theoretic characterization of logic programming. A well-founded combination of higher-order logic programming and linear logic will be used to give an accurate encoding of the traditional features of concurrent object-oriented programming languages, whose corner-stone is the notion of *encapsulation.*

1. Introduction

This paper focuses on the use of linear logic as a language to specify advanced concepts of programming, and, in particular, to specify the salient aspects of concurrent object-oriented programming.

Our approach is based on the refinement, which, from the original logic defined by Girard in Girard (1987), led to *executable linear logic specification languages* such as *LO* (Andreoli and Pareschi 1991) and Lolli (Hodas and Miller 1994). From a general point of view, such a refinement is based on specific *operational* interpretations of formulae, sequents and proofs. From a technical point of view, it consists of a proof theoretical analysis of the underlying logic aimed at the definition of classes of proofs that correspond to the operational view taken into consideration. In a sense, these are the ideas behind the design of traditional logic programming languages (Miller *et al.* 1991).

[†] At the time of submission the author was a member of Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

The refinement process will be presented through a classification of the different approaches in the literature. Such a classification is based on the following points: the set of connectives used as *primitive constructs* of the specification language (for example, the different fragments used in ACL (Kobayashi and Yonezawa 1994a), *LO* (Andreoli and Pareschi 1991), Lolli (Hodas 1994), and so on); the level of non-determinism of the proof-search procedure (for example, canonical proofs (Galmiche and Perrier 1994b), uniform proofs (Hodas 1994), and so on); the form of sequents (for example, single and multi-conclusion). In the course of this paper we shall explain which considerations led to the choice of the fragments that were studied in the literature.

To achieve our main goal, that is, to study concurrent objects in the linear logic setting, we shall adopt a particular interpretation of sequents and proofs, that is, *uniform proofs as computations*, as proposed by Miller in Miller (1996) using the language Forum, a presentation of full higher-order linear logic. This choice is made on the basis of previous work on extensions of logic programming in which the logical connectives are given a precise operational interpretation in terms of *search directives*, see, for example, Miller (1989a), Miller (1989b), Andreoli and Pareschi (1991), Hodas and Miller (1994) and Miller (1996). In a sense, Forum (Miller 1996) can be viewed as an intermediate refinement step to achieve a *readable* form of linear logic specification.

Our final refinement step is to consider a specific sublanguage, namely $\mathscr{E}_{hhf}$ (Delzanno 1997; Delzanno and Martelli 1998), by which we emphasize the view of proofs as *state-based* computations (essential in the object-oriented paradigm). The restriction on the form of formulae adopted in $\mathscr{E}_{hhf}$ allows us to define executable linear logic programs with a semantics dictated by the specialized proof system of $\mathscr{E}_{hhf}$. This restriction represents a good compromise between the expressiveness of the logic and the efficiency (and readability) of the proof-search process. Following the outlines given in our previous works (Delzanno and Martelli 1995; Boudinet and Galmiche 1996), we shall illustrate all of these points by presenting an accurate encoding of the most common features of concurrent object-oriented programming. More specifically, we shall focus on *encapsulation*, *method invocation*, *inheritance* and *overriding* at the object-level. Concurrency at the execution level will be modelled in a natural way by assigning an *interleaving* semantics to the execution of methods.

As will become clear from the discussion in the first part of the paper, other fragments of linear logic can be applied in this context. For instance, thanks to the strong symmetry of linear logic it is possible to design *dual* encodings in which sequents and proofs assume different operational interpretations, see, for example, the approach based on multi-conclusion intuitionistic linear logic in Boudinet and Galmiche (1996). In our opinion, such flexibility is one of the main reasons to further inspect the potentiality of linear logic and, more generally, of proof-theoretic approaches for the specification of the semantics of programming languages.

1.1. *Contents of the paper*

In Section 2 we present the basic notions of linear logic. In Section 3 we present various sequent calculi for linear logic discussing their operational interpretation. In Section 4 we

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

briefly introduce the language $\mathscr{E}_{hhf}$. In Sections 5 and 6 we present a specification of an object-based concurrent calculus based on $\mathscr{E}_{hhf}$. In Section 7 we compare our approach with related works. Finally, in Section 8 we address some conclusions.

2. Linear logic and aspects of programming

Linear logic (LL) is a powerful and expressive logic connected to a variety of topics in computer science (Alexiev 1994). From a proof-theoretical point of view, LL derives from Classical Logic (CL) sequent calculus by eliminating the structural rules of *weakening* and *contraction*. The result is a logic in which it is intuitively possible to treat formulae as *resources*. Contraction and weakening are re-introduced in a restricted way, *i.e.*, they can be applied only to the subclass of formulae prefixed by the two modalities ! and ?.

The consequences of eliminating the structural rules are important for the formulation of the logical rules: the additive and the multiplicative formulations of the system are no longer equivalent. As a consequence, each CL connective is split into two LL connectives (an additive and a multiplicative version). For instance,

— $\wedge$ is split into $\&$ and $\otimes$
— $\vee$ is split into $\oplus$ and $\mathbin{⅋}$
— $\supset$ into $\leadsto$ and $\multimap$
— *true* into $\top$ and $\mathbf{1}$
— $false$ into $\mathbf{0}$ and $\perp$.

Negation, that is, $^{\perp}$, is inductively defined on the structure of the formulae, as illustrated in Appendix A. Beacause of its symmetry, the inference systems of LL can be given with two equivalent formulations: one with one-sided sequents $\vdash \Delta$, the other with two-sided sequents $\Gamma \vdash \Delta$ (see Appendix A). Here, Δ and Γ are *multisets* of formulae.

Intuitionistic linear logic (ILL) (Schellinx 1991) is usually defined by restricting the right-hand side of sequents to an empty or a singleton multiset. However, as in the classical case, multi-conclusion formulations of ILL with special restrictions on some of the rules, for example, full intuitionistic linear logic (FILL) (Hyland and de Paiva 1993) exist.

LL is often referred to as a logic for *concurrency* (see, for example, Meseguer (1991) and Abramsky (1993)). The reason for this can be illustrated by considering the *process-view* of Kobayashi and Yonezawa (1994a). Their approach is based on an interpretation whereby formulae are viewed as processes, and connectives as algebraic operations on processes: $\mathbin{⅋}$ represents the parallel composition, whereas $^{\perp}$ (in combination with $\otimes$) implements message passing. The rules shown in Appendix A assign a natural operational semantics to this interpretation.

As an example, let us consider the following formula: $m \mathbin{⅋} (m^{\perp} \otimes P_1) \mathbin{⅋} P_2$. Here, m is viewed as a process that halts its execution after having sent the message m, and $(m^{\perp} \otimes P_1)$ is viewed as a process waiting for the message m and running in parallel with P_2. Using the rules shown in Appendix A, the sequent $\vdash m \mathbin{⅋} (m^{\perp} \otimes P_1) \mathbin{⅋} P_2$ can be simplified via the $⅋_R$ rule into $\vdash m, (m^{\perp} \otimes P_1), P_2$ and then via the $\otimes_R$ rule into $\vdash P_1, P_2$ (note that $\vdash m, m^{\perp}$ is an axiom). Thus, the resulting (partial) proof can be interpreted in terms of process reduction.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

With this in mind, let us now adopt a fragment that does not include the connective $\otimes$, but does include the connective $\multimap$. By the duality of the linear connectives, the receiver can be rewritten as $P_1 \multimap m$. However, we need a two-sided sequent formulation, namely $P_1 \multimap m \vdash m, P_2$ in order to have the same operational behaviour as before.

Through the example, we can see the importance of the interaction within the logical fragment, the use of specific formulae and sequents to specify computational aspects, and the proof-search strategy. Each choice represents a compromise between the expressiveness of the fragment and the efficiency of the corresponding proof-search strategy. In the following section we shall give a brief overview of the different components involved in this process.

3. Proof-theoretic analysis

The formalization of a logic based on sequent calculus provides us with better comprehension of the *operational* aspects of proof construction. In the meantime, sequent calculi provide a powerful organization of the knowledge specified through the formulae of the logic language taken into consideration. Some ideas concerning these two aspects when dealing with LL are given in the rest of this section.

3.1. *Sequents and proofs*

Though the basic use of sequents is to express theorems of the logic taken into consideration, they can be given other interpretations. For instance, under the previously mentioned *formulae-as-processes* view of Kobayashi and Yonezawa (1994a), one-sided sequents can be used to specify configurations of processes. By contrast, in Lincoln and Saraswat (1993) and Perrier (1995) a sequent is viewed as a reduction of processes (that is, $P \vdash P'$ is interpreted as P reduces to P'). This can be achieved in a single-conclusion setting as in fragments of ILL (Perrier 1995) or in a multi-conclusion setting as in fragments of FILL (Boudinet and Galmiche 1996).

In the context of logic programming, one usually considers sequents of the form $P \vdash G$, where the set of formulae P represents the program and the formula G represents the goal to be satisfied. In the linear logic programming setting, further refining the sequent syntax in order to take into account the notion of *bounded-use* formulae comes naturally. For example, as will be explained later, the Forum sequents (Miller 1994) have the form $\Sigma : \Gamma_1; \Gamma_2 \vdash \Delta; \Upsilon$, where Σ is a signature, and Γ_1, Υ and Γ_2, Δ are, respectively, the re-usable and bounded-use context. Such refinements are motivated by the need to provide simple and readable *judgements* that take into account the operational interpretation one keeps in mind.

We shall see that the choice between single-conclusion sequents, for instance in intuitionistic linear logic (ILL) (Hodas and Miller 1994), and multiple-conclusion sequents, for instance in LL (Miller 1994) and in FILL (Hyland and de Paiva 1993), has consequences both at the specification and at the proof-search levels.

In a sense, the operational interpretation of sequents makes the difference between theorem proving and logic programming. Both theorem proving and logic programming

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

can be viewed as the process of constructing the proof of a sequent from the bottom up (*i.e.*, from the sequent to be proved). As mentioned in Hodas and Polakow (1996), the line between theorem provers and logic programming systems is drawn by what sort of proof procedure is used to prove the goal formulae Δ from Γ. In theorem proving, any 'reasonable' procedure is a good choice. On the other hand, in logic programming the programmer must be able to clearly understand the behaviour of a program. Thus, the proof procedure has to be simple, predictable and *goal-directed*. In other words, it must reflect the operational interpretation assigned to formulae and sequents.

3.2. *Proof-search in linear logic*

There are many works devoted to proof-search in linear logic and some of its sub-fragments. All of these proposals are based on the non-permutability results of the LL inference rules in the corresponding sequent calculus. Examples are given by the proof-search strategies defined in Galmiche and Perrier (1994a), Lincoln and Shankar (1994) and Tammet (1994), and by the classes of proofs defined in Andreoli (1992), Galmiche and Perrier (1994b), Hodas (1994) and Pym and Harland (1994), that are complete with respect to the provability.

Let us analyze how we can naturally design and justify such strategies and proofs for a given logical fragment. For this purpose, we recall a general, two-step, method applied to LL and to other fragments in Galmiche and Perrier (1994a) and Galmiche and Perrier (1994b). The first step consists in studying the *permutability* of the inference rules and in analyzing the possibilities of inference movements (by permutability) in a proof. This can be done systematically by an exhaustive case analysis. After establishing a given strategy (for example, bottom-up or top-down), the second step consists in defining a notion of (cut-free) *normal* proof, which reflects the strategy we are considering, *i.e.*, in which we impose some order in the application of the rules. The degree of non-determinism left in this phase depends on the operational interpretation assigned to the proofs (*i.e.*, it can reflect the non-determinism of the object-level). Further constraints on the proof-search (for example, goal-directed proofs) can lead to different classes of normal proofs. Normal proofs are significant only if they can be proved complete with respect to the full class of proofs of the fragment taken into consideration. As an example, let us consider the design of *canonical proofs* in LL as defined in Galmiche and Perrier (1994b). Starting from the one-sided sequent calculus (see Appendix A), we study the inference permutability and the possible movements of inferences in a proof. If we select a bottom up search strategy, the set of inferences that can be moved up in a proof is $I_\uparrow = \{\otimes, \oplus_1, \oplus_2, c?, w?, ?, \exists\}$, whereas the set of inferences that can be moved down is $I_\downarrow = \{⅋, \&, \forall, \bot\}$.

Canonical proofs result from imposing an order in the application of the rules according to the two previously defined sets.

Definition 3.1. A *canonical proof* in (full) LL for bottom-up proof-search is a proof without *cut*s, with weakening and contraction reduction, where, for any intermediate conclusion, we first apply the $c?$ rule, then the $!$ rule, then a rule of $I_\downarrow$ and, finally, a rule of $I_\uparrow$ (Galmiche and Perrier 1994b).

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

This proof normalization includes a reduction of the possible interactions between weakening and contraction rules. Canonical proofs are complete with respect to provability in linear logic, that is, if a sequent s is provable, then it has a canonical proof. This definition illustrates the generality of the above *methodology* for a given logic quite well. To move to a more concrete framework, we need to select a particular interpretation of sequents and a special proof-search strategy. Therefore, in the following section we shall adopt the so-called *proofs as computations* interpretation in the context of linear logic programming.

3.3. *Proof-search as computation*

As already mentioned in the previous section, after a preliminary proof-theoretic analysis, it is important to establish a strategy that is in agreement with the operational interpretation assigned to sequents and proofs. An important example is goal-directed proof-search, which is at the basis of logic programming. This strategy is reflected in the notion of *uniform provability*, originally introduced in the case of intuitionistic (classical) logic in Miller *et al.* (1991). Uniform proofs are suitable for assigning a clear operational meaning to logical connectives in terms of search control directives. Goal-directed proof construction strongly depends on the form of sequents considered in the calculus, *i.e.*, single or multi-conclusion calculi.

3.3.1. *Single-conclusion sequents.* As customary in proof-theoretic characterizations of logic programming, a sequent $P \vdash G$ is viewed as the instantaneous configuration of an ideal interpreter, where the formulae of P represent the current program and the formula G represents the current goal to be satisfied. Goal-directed provability is formalized by the class of uniform proofs defined as follows:

Definition 3.2. A *uniform proof* is a cut-free proof in which every occurrence of a sequent whose right-hand side is non-atomic is the conclusion of a right-introduction rule (Miller *et al.* 1991).

Thus, in this setting, left-rules can be applied only after the right-hand side of a sequent has been reduced to a singleton consisting of an atomic formula. This definition has been extended to the case of linear logic for particular subclasses of formulae (Hodas and Miller 1994; Pym and Harland 1994). The resulting languages, for example, Lolli (Hodas and Miller 1994) and Lygon (Harland *et al.* 1996), provide new *programming constructs* with respect to traditional extensions of logic programming based on intuitionistic logic such as λProlog (Nadathur and Miller 1988).

To complete the overview of the methodology used to define special purpose sublogics, we will briefly analyze the logic introduced in Hodas and Miller (1994).

Example 3.1. (Analysis of a fragment of ILL) As pointed out in Hodas and Miller (1994), the design of Lolli was guided by operational considerations: to extend hereditary Harrop formulae with resource management by introducing new connectives, namely $\multimap$ and $\otimes$, and preserving uniform provability. The fragment proposed in Hodas and Miller (1994) is defined by the following grammar:

$$\mathscr{D} \quad ::= \quad \top \mid A \mid \mathscr{D} \& \mathscr{D} \mid \mathscr{G} \multimap A \mid \forall x.\mathscr{D}.$$
$$\mathscr{G} \quad ::= \quad \top \mid A \mid \mathscr{G} \& \mathscr{G} \mid \mathscr{G} \otimes \mathscr{G} \mid \mathscr{G} \oplus \mathscr{G} \mid \mathscr{D} \multimap \mathscr{G} \mid !\mathscr{D} \multimap \mathscr{G} \mid !\mathscr{G} \mid \exists x.\mathscr{G} \mid \forall x.\mathscr{G}.$$

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

$$\frac{}{A, !\Gamma \vdash A}\ id' \qquad \frac{\Gamma_1, !\Gamma \vdash G_1 \quad \Gamma_2, !\Gamma \vdash \Gamma_2}{\Gamma_1, \Gamma_2, !\Gamma \vdash G_1 \otimes G_2}\ \otimes'_R \qquad \frac{\Gamma \vdash G}{G \multimap A, \Gamma \vdash A}\ \multimap'_L$$

$$\frac{R, \Gamma \vdash G}{\Gamma \vdash R \multimap G}\ \multimap_R \qquad \frac{A, \Gamma \vdash G}{A \& B, \Gamma \vdash G}\ \&_L \qquad \frac{B, \Gamma \vdash G}{A \& B, \Gamma \vdash G}\ \&_L$$

$$\frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \& G_2}\ \&_R \qquad \frac{\Gamma \vdash G_1}{\Gamma \vdash G_1 \oplus G_2}\ \oplus_R \qquad \frac{\Gamma \vdash G_2}{\Gamma \vdash G_1 \oplus G_2}\ \oplus_R$$

$$\frac{}{\Gamma \multimap A, \Gamma_1, \Gamma_2 \vdash \top}\ \top'_R \qquad \frac{A, !A, \Gamma \vdash G}{!A, \Gamma \vdash G}\ !'_L \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\ !_R$$

$$\frac{A[t/x], \Gamma \vdash G}{\forall x A, \Gamma \vdash G}\ \forall_L \qquad \frac{\Gamma \vdash G}{\Gamma \vdash \forall x G}\ \forall_R \qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x A}\ \exists_R$$

Fig. 1. The uniform proof system for Lolli (Hodas and Miller 1994).

The sequents have the specific form $!\Gamma, \Delta \vdash G$, where Γ is a set of $\mathscr{D}$-formulae, Δ is a multiset of $\mathscr{D}$-formulae and G is a goal, that is, a $\mathscr{G}$-formula.

By the subformula property, we can limit our analysis to the following logical rules:

$$\mathscr{R} = \{\multimap_L, id, \&_L, !_L, c!_L, w!_L, \forall_L, \top_R, \otimes_R, \multimap_R, \&_R, \oplus_R, \forall_R, !_R, \exists_R\}$$

We now study the permutability properties of the $\mathscr{R}$ rules (see Appendix B). Analysis of the possible movements of inference rules leads to the following considerations. First of all, being in a single-conclusion setting, two right rules are never in permutation position. Moreover, since a resource cannot have the form $!D$, the inference of type $!_R$ is always permutable with inference of type $\multimap_L, \&_L$ and $\forall_L$. As a consequence, the set of inferences $I_\downarrow$ of the considered fragment that can be permuted downward consists of $\{\multimap_R, \&_R, \forall_R, \exists_R, \oplus_R, c!_L\}$, whereas the set $I_\uparrow$ of inferences that can be permuted upward consists of $\{\otimes_R, \oplus_R, \exists_R, \multimap_L, \&_L, !_L, w!_L, \forall_L\}$. The inferences of $!_R$ type can be moved down by jumps but not because of permutability results. Note that, if we want to establish a particular order in the application of the rules, we still have various choices (for example, for $\oplus_R$). However, except for the $c!_L$ and $\otimes_R$ inferences, it appears that the right (left) rules can easily be moved down (up). Having established such specific movements it is now possible to move the $\otimes_R$ inferences down easily, due to the permutability with the left rules. This order corresponds to the idea of uniform provability defined at the beginning of this section.

In Hodas and Miller (1994), a proof that uniform provability (in the fragment taken into consideration) is complete with respect to provability in ILL is given. Let us mention that the permutability results are also used to define a variant of the initial system, in which all proofs are necessarily uniform (see Figure 1). Thus, the new sequent calculus incorporates the operational interpretation underlying uniform provability. The reader may refer to Pym and Harland (1994) for an accurate analysis of the maximal fragments of LL that are complete with respect to goal-directed provability.

As mentioned in Section 2, multi-conclusion calculi provide a natural view of concurrency at the logical level. Thus, the logical framework we are looking for should

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

incorporate features from the extensions described in this section and features from multi-conclusion logics. We will discuss this point in the following section.

3.3.2. *Multi-conclusion sequents.* According to the logic programming interpretation of sequents and proofs, being in a multi-conclusion setting provides a new dimension that increases the expressiveness of traditional logic programming languages. Let us consider a two-sided sequent $\Gamma \vdash \Delta$. In this setting the formulae of Γ represent the current program, whereas the formulae of Δ represent a *multiset* of goals to be proved. Thus, in the extended setting we can encode the process-view of Section 2, as suggested in Andreoli and Pareschi (1991) and Miller (1993).

Being in a multi-conclusion setting leads to an extended notion of goal-directed provability (Andreoli 1992; Miller 1994). The problem encountered by such an extension is well explained in Miller (1994), where the higher order specification logic Forum is presented.

Forum (Miller 1994; Miller 1996) is based on a simply typed lambda calculus presentation of LL. The connectives are constants with functional type, for example, $\& : o \rightarrow o \rightarrow o$, with o representing the type of formulae. Higher-order quantification is defined through a family of constants $\forall_\tau$ having the type $(\tau \rightarrow o) \rightarrow o$ for each type τ. A quantified formula is then expressed by the expression $\forall_\tau(\lambda x.F)$. These aspects can be hidden when considering a proof-theoretic presentation of the logic. However, it is important to remember that λ-terms in normal form can occur inside Forum sequents as well as substitution terms in the $\forall_r$ rule. The language is based on a fragment of LL with the following set of connectives: $\multimap$, $\Rightarrow$, $\&$, $\invamp$, $\bot$, $\top$, ? and universal quantification. For the sake of simplicity, we shall omit the type in the quantification when it is clear from the context. The intuitionistic implication $A \Rightarrow B$ is defined here as $(!A \multimap B)$. In the rest of the paper we shall also use $A \circ\!\!- B$ and $A \Leftarrow B$ as an alternative notation for $B \multimap A$ and $B \Rightarrow A$, respectively. Using the logical equivalences in Appendix A, it is easy to see that by proper combinations of Forum connectives it is possible to express all the remaining ones.

Forum sequents are the multi-conclusion version of the Lolli-ones. They have the following form: $\Sigma : \Gamma; \Delta \vdash \mathscr{A}, F, \Omega; \Upsilon$, where, Σ is a signature containing all the constants appearing in the sequent (Miller 1994). The two sides are divided by ';' in two parts in order to distinguish between the re-usable formulae and bounded-use formulae. In fact, the formulae in the multiset Γ are implicitly prefixed by '!', whereas the formulae in the multiset Υ are implicitly prefixed by '?'. Finally, the bounded context on the right consists of a *list* of atomic formulae $\mathscr{A}$, a compound formula F and a multiset of formulae Ω. Intuitively, $\mathscr{A}$ is the result of the simplification of the connectives on the right-hand side of sequents. We can also give an equivalent formulation in which the right-hand side is simply a *multi-set* of formulae. The LL interpretation of such sequents is the following: $!(\&_i C_i) \otimes (\otimes_i D_i) \rightarrow (\invamp_i A_i) \invamp F \invamp (\invamp_i G_i) \invamp (\oplus_i H_i)$, where $C_i \in \Gamma$, $D_i \in \Delta$, $A_i \in \mathscr{A}$, $G_i \in \Omega$, $H_i \in \Upsilon$. Using the permutability properties of Forum rules, it is possible to define a specialized proof system based on the following extended notion of uniformity (Miller 1996).

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

Definition 3.3. A cut-free proof δ is *uniform* if for every subproof γ of δ and for every non-atomic occurrence B in the right-hand side of the end-sequent of δ, there is a proof π that is equal to γ up to a permutation of inference rules and is such that the last inference rule in π introduces the top-level logical connective of B (Miller 1994).

$$\frac{\Gamma;\Delta \xrightarrow{B} \mathscr{A};\Upsilon}{\Gamma;\Delta,B \longrightarrow \mathscr{A};\Upsilon}\ (decide_\Delta) \qquad \frac{\Gamma,B;\Delta \xrightarrow{B} \mathscr{A};\Upsilon}{\Gamma,B;\Delta \longrightarrow \mathscr{A};\Upsilon}\ (decide_\Gamma) \qquad \frac{\Gamma;\Delta \longrightarrow \mathscr{A},B;B,\Upsilon}{\Gamma;\Delta \longrightarrow \mathscr{A};B,\Upsilon}\ (decide?)$$

$$\frac{}{\Gamma;\varnothing \xrightarrow{A} A;\Upsilon}\ (initial_1) \qquad \frac{}{\Gamma;\Delta \longrightarrow \mathscr{A},\top,\Omega;\Upsilon}\ (\top_r) \qquad \frac{}{\Gamma;\varnothing \xrightarrow{A} \varnothing;A,\Upsilon}\ (initial_2)$$

$$\frac{}{\Gamma;\varnothing \xrightarrow{\bot} \varnothing;\Upsilon}\ (\bot_l) \qquad \frac{\Gamma;\Delta \longrightarrow \mathscr{A},\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathscr{A},\bot,\Omega;\Upsilon}\ (\bot_r)$$

$$\frac{\Gamma;B \longrightarrow \varnothing;\Upsilon}{\Gamma;\varnothing \xrightarrow{?B} \varnothing;\Upsilon}\ (?_l) \qquad \frac{\Gamma;\Delta \longrightarrow \mathscr{A},\Omega;B,\Upsilon}{\Gamma;\Delta \longrightarrow \mathscr{A},?B,\Omega;\Upsilon}\ (?_r)$$

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

$$\frac{\Gamma;\Delta_1 \xrightarrow{B} \mathscr{A}_1;\Upsilon \quad \Gamma;\Delta_2 \xrightarrow{C} \mathscr{A}_2;\Upsilon}{\Gamma;\Delta_1,\Delta_2 \xrightarrow{B⅋C} \mathscr{A}_1+\mathscr{A}_2;\Upsilon}\ (⅋_l) \qquad \frac{\Gamma;\Delta \longrightarrow \mathscr{A},B,C,\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathscr{A},B⅋C,\Omega;\Upsilon}\ (⅋_r)$$

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

$$\frac{\Gamma;\varnothing \longrightarrow B \quad \Gamma;\Delta \xrightarrow{C} \mathscr{A};\Upsilon}{\Gamma;\Delta \xrightarrow{B\Rightarrow C} \mathscr{A};\Upsilon}\ (\Rightarrow_l) \qquad \frac{\Gamma,B;\Delta \longrightarrow \mathscr{A},C,\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathscr{A},B\Rightarrow C,\Omega;\Upsilon}\ (\Rightarrow)$$

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

Fig. 2. The Forum proof system: Υ denotes a multiset of *atomic formulae*.

The resulting proof system is shown in Figure 2. In Miller (1996), Forum has been proved to be equivalent to the system of Andreoli (1992), and thus to full linear logic. According to the previous definition of uniformity, Forum can be considered an Abstract

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

Logic Programming Language (Miller *et al.* 1991). Some examples of applications of Forum as a high-level specification language for different programming paradigms can be found in Chirimar (1995), Delzanno and Martelli (1995), Guglielmi (1995) and Hodas and Polakow (1996). As mentioned in Hodas and Polakow (1996), further refinement seems necessary before considering Forum as a logic programming language. The main problem with this approach is that the extended notion of uniformity seems too loose to obtain readable and predictable proofs. Thus, the question of how to program with such a specification logic naturally arises. To answer it, in the following section we restrict ourselves to a special fragment of Forum.

4. Towards executable specifications

In this section we describe the main features of the logic of *extended hereditary Harrop formulae* (Delzanno 1997; Delzanno and Martelli 1998), a fragment of Forum that can be used as an *executable* specification language.

4.1. *Syntax of the language $\mathscr{E}_{hhf}$*

The logic of $\mathscr{E}_{hhf}$ is based on the fragment of linear logic connectives $\multimap$, $\Rightarrow$, $\&$, $⅋$, $\bot$, $\top$, and universal quantification. Forum formulae are further restricted by the production $\mathscr{D}$ (clauses) and $\mathscr{G}$ (goals) of the following grammar:

$$\begin{array}{lll}\mathscr{D} & ::= & \mathscr{D}\,\&\,\mathscr{D} \mid \forall x.\mathscr{D} \mid \mathscr{H} \circ\!- \mathscr{G} \mid \mathscr{G} \Rightarrow \mathscr{D} \mid \mathscr{H}. \\ \mathscr{G} & ::= & \mathscr{G}\,\&\,\mathscr{G} \mid \mathscr{G} ⅋ \mathscr{G} \mid \forall x.\mathscr{G} \mid \mathscr{D} \Rightarrow \mathscr{G} \mid \mathscr{D} \multimap \mathscr{G} \mid A_r \mid \bot \mid \top. \\ \mathscr{H} & ::= & \mathscr{H} ⅋ \mathscr{H} \mid A_r.\end{array}$$

Here, A_r is any *rigid* atomic formula with type o (that is, $A_r = (f\ t_1\ \ldots\ t_n)$ with $f \in \Sigma$ a constant symbol). $\mathscr{E}_{hhf}$-sequents have the following form: $\Gamma;\Delta \longrightarrow_\Sigma \Omega$ or $\Gamma;\Delta \overset{D}{\longrightarrow}_\Sigma \mathscr{A}$, where Γ is a set of $\mathscr{D}$-formulae, Δ is a multiset of $\mathscr{D}$-formulae, Ω is a multiset of $\mathscr{G}$-formulae, D is a $\mathscr{D}$-formula and $\mathscr{A}$ a multiset of atomic formulae. All the formulae are defined over the signature Σ. Intuitively, a sequent denotes an instantaneous configuration of the computation denoted by its proof. The left-hand side is the current program (with unbounded and bounded-use clauses) and the right-hand side is the current multiset of goals to be executed.

The set of Forum rules can be specialized to the type of sequents taken into consideration, as shown in Figure 3. All the rules are modulo λ-conversion. Note that the left rules collapse into one single backchaining rule, namely *bc*. In such a rule $\langle D\rangle$ represents the set of instances of a multiset Δ of $\mathscr{D}$-clauses over a signature Σ. More precisely, $\langle\Delta\rangle$ is the smallest set of $\mathscr{D}$-clauses such that: $\langle\Delta\rangle = \bigcup_{D\in\Delta}\langle D\rangle$; $\langle D_1\&D_2\rangle = \langle D_1\rangle \cup \langle D_2\rangle$; $\langle\forall_\tau x.D\rangle = \langle\{D[t/x] \mid t : \tau \text{ is a } \Sigma\text{-term}\}\rangle$; and $\langle A\rangle = \{A\}$ in all the other cases.

4.2. *Expressiveness of the language*

Horn clauses and hereditary Harrop formulae are a particular case of $\mathscr{E}_{hhf}$, that is, they can be embedded into $\mathscr{E}_{hhf}$ while preserving *provability*. Horn clauses can be expressed by

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

$$\frac{}{\Gamma;\varnothing \xrightarrow{D}_{\Sigma} \mathscr{A}}\ initial$$

$$(A_1 \parr \ldots \parr A_n) \in \langle D \rangle,\ \{A_1,\ldots,A_n\} \equiv \mathscr{A}.$$

$$\frac{\Gamma, D;\Delta \xrightarrow{D}_{\Sigma} \mathscr{A}}{\Gamma, D;\Delta \longrightarrow_{\Sigma} \mathscr{A}}\ decide_1 \qquad \frac{\Gamma;\Delta \xrightarrow{D}_{\Sigma} \mathscr{A}}{\Gamma;\Delta, D \longrightarrow_{\Sigma} \mathscr{A}}\ decide_2$$

$$\frac{}{\Gamma;\Delta \longrightarrow_{\Sigma} \top,\Omega}\ \top_r \qquad \frac{\Gamma;\Delta \longrightarrow_{\Sigma} \Omega}{\Gamma;\Delta \longrightarrow_{\Sigma} \bot,\Omega}\ \bot_r \qquad \frac{\Gamma;\Delta \longrightarrow_{\Sigma'} A[y/x],\Omega}{\Gamma;\Delta \longrightarrow_{\Sigma} \forall_\tau x.A,\Omega}\ \forall_r$$

$$\frac{\Gamma;\Delta \longrightarrow_{\Sigma} A_1, A_2,\Omega}{\Gamma;\Delta \longrightarrow_{\Sigma} A_1 \parr A_2,\Omega}\ \parr_r \qquad \frac{B,\Gamma;\Delta \longrightarrow_{\Sigma} A,\Omega}{\Gamma;\Delta \longrightarrow_{\Sigma} B \Rightarrow A,\Omega}\ \Rightarrow_r \qquad \frac{\Gamma; B,\Delta \longrightarrow_{\Sigma} A,\Omega}{\Gamma;\Delta \longrightarrow_{\Sigma} B \multimap A,\Omega}\ \multimap_r$$

$$\frac{\Gamma;\Delta \longrightarrow_{\Sigma} A_1,\Omega \quad \Gamma;\Delta \longrightarrow_{\Sigma} A_2,\Omega}{\Gamma;\Delta \longrightarrow_{\Sigma} A_1\ \&\ A_2,\Omega}\ \&_r$$

$$\frac{\Gamma;\ \longrightarrow_{\Sigma} G \quad \Gamma;\Delta \longrightarrow_{\Sigma} B,\mathscr{A}'}{\Gamma;\Delta \xrightarrow{D}_{\Sigma} \mathscr{A},\mathscr{A}'}\ bc$$

$$G \Rightarrow (A_1 \parr \ldots \parr A_n \mathrel{\circ\!-} B) \in \langle D \rangle,\ \{A_1,\ldots,A_n\} = \mathscr{A}.$$

Fig. 3. $\mathscr{E}_{hhf}$-derived rules. In $(\forall_r)$, $y:\tau \notin \Sigma$ and $\Sigma' = y:\tau,\Sigma$.

$\mathscr{D}$-clauses of the form $G \Rightarrow A$, where A is a rigid atomic formula and G is a goal formula in one of the following forms: $\top$, $B_1\ \& \ldots \&\ B_n$, where the B_i's are atomic formulae. Hereditary Harrop formulae are obtained from the grammar in Section 4.1 by removing the productions for the connectives $\parr, \multimap$, and $\bot$.

4.3. *Computational view of proofs*

Clauses can be rewritten as conjunctions $D_1 \& \ldots \& D_k$ of formulae in the following form:

$$\forall\ G_1\ \& \ldots \&\ G_n \quad \Rightarrow\ (A_1 \parr \ldots \parr A_m \mathrel{\circ\!-} G),\ n,m \geqslant 1$$

where G and $G_1,\ldots,G_n$ are goal formulae. Such clauses have the following intuitive operational meaning: *if the goals* $G_1,\ldots,G_n$ *can be proved, then the multiset* $A_1,\ldots,A_m$ *in the right-hand side of the current sequent can be rewritten into* G. This idea is formalized by the backchaining rule (bc) in Figure 3. Note that, given a sequent s, there are several ways of applying a rule D in a *bc* step to s. In practice, the non-determinism induced by the *bc* rule is managed by the use of *backtracking* (that is, the possibility to roll back to a choice-point).

As specified by the rule *initial*, a backchaining step over a clause D of the form $A_1 \parr \ldots \parr A_n$ (that is, without body with respect to the nested implication $\mathrel{\circ\!-}$) corresponds to the conclusion of the computation with respect to a given property of the current state, we want to observe. In fact, at this stage we *observe* the final state of the computation by

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

matching D with the right-hand side of the current sequent. The rule $decide_1$ allows one to use the clauses in a program Γ an unlimited number of times, whereas the rule $decide_2$ removes the selected clause from the current context.

4.4. *New types of goals*

Goal formulae have a richer structure than goals in classical logic programming. For instance, goals of the form $G_1 \parr G_2$ express the *concurrency* in the execution of the subgoals G_1 and G_2 in a natural way. The two subgoals are, in fact, simultaneously rewritten in the current multiset of goals to be proved. Linear implications of the form $D \multimap G$ allow one to enrich the current multiset of bounded-use clauses with a new one (see rule $\multimap_r$). The newly introduced resource must be consumed during the rest of the proof, *i.e.*, it will be applied in a subsequent backchaining step. Here, we require D to be a clause when such a goal is executed. In Delzanno and Martelli (1998), the authors study a variation on the language in which *variables* of type o can occur in $\mathscr{D}$-position within the scope of a goal, and they prove that the resulting language is closed under instantiation. We shall use this feature in the rest of the paper. The logical constant $\bot$ can be used to *remove* resources from the right-hand side of sequents, for example, by using a clause of the form $A \multimapinv \bot$. Finally, a sequent containing $\top$ in the right-hand side succeeds whatever formulae occur in the current bounded contexts. The other connectives have the operational interpretation inherited by the logic of hereditary Harrop formulae (Nadathur and Miller 1988). For instance, Miller (1989b) shows how to employ the side condition of the rule $\forall_r$ to introduce a notion of data abstraction over components of the current configuration.

4.5. *Refinement of the syntax*

In order to emphasize the computational view of proofs, we introduce a refined syntax for sequents and clauses.

Sequents assume the following form: $\Gamma; \Delta \longrightarrow_\Sigma \Omega \ \| \ \Theta$, where the right-hand side is partitioned into two multisets: the *operations*, Ω, and the *state atoms*, Θ. State atoms are nothing but atoms built upon a signature $\Sigma_{\mathscr{R}}$ of state constructors established *a priori*. By default, $\Sigma_{\mathscr{R}}$ is contained in any signature Σ mentioned in the rest of this paper. We assume that each atomic formula occurring on the right-hand side of a sequent is implicitly *moved* into the current Θ. In the rest of this paper, we shall adopt a slightly different syntax for clauses, namely

$$\underbrace{C_1 \ \&\ \ldots\ \&\ C_n}_{Cond}\ \texttt{>>}\ \underbrace{A_1 \mid \ldots \mid A_m}_{Ops}\ \ \texttt{||}\ \ \underbrace{S_1 \mid \ldots \mid S_k}_{State}\ \texttt{-->}\ \texttt{Body}\ \texttt{||}\ \underbrace{Z_1 \mid \ldots \mid Z_p}_{New\ \ State}$$

Such an expression will denote the corresponding $\mathscr{E}_{hhf}$-clause:

$$C_1 \ \&\ \ldots\ \&\ C_n \Rightarrow (A_1 \parr \ldots \parr A_m \parr S_1 \parr \ldots \parr S_k) \ \multimapinv\ (Body \ \parr\ Z_1 \parr \ldots \parr Z_p).$$

The symbol `||` is used in sequents and clauses to distinguish state-atoms from operations. The free variables of this expression (denoted by identifiers beginning with a capital letter) are implicitly universally quantified. When necessary, universally quantified formulae $\forall x.F$

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

will be explicitly represented as `pi X \ F` (according to the λProlog notation (Nadathur and Miller 1988)). If a clause has no conditional part, that is, *Cond* in the previous figure, then we use the notation OPS `||` STATE `-->` BODY `||` NEW STATE.
Facts without body are written [COND `>>`] OPS `||` STATE. The constant $\perp$, written `anti`, is used to denote an empty BODY, STATE or NEW STATE component. The constant $\top$ will be written as `all`. The previous notation emphasizes the similarities between $\mathscr{E}_{hhf}$-clauses and rewriting rules, between $\mathscr{E}_{hhf}$-sequents and instantaneous configurations, and between $\mathscr{E}_{hhf}$-proofs and state-based computations.

4.6. *Final state of a computation*

The final state of a computation corresponds to a sequent of the form: $\Gamma; \emptyset \longrightarrow_\Sigma \emptyset \parallel \Theta$, where Θ is a multiset of $\Pi_{\Sigma_\mathscr{R}}$ atoms. Such a sequent is not an axiom unless a formula Φ in Γ that *matches* the facts in Θ exists. The formula Φ can be easily *re-constructed* after building a partial proof tree with leaves of the form $\Gamma; \emptyset \longrightarrow_\Sigma \emptyset \parallel \Theta$. We therefore single out Φ from the unbounded context and adopt the following extended syntax for the sequents: $\Gamma[\Phi]; \emptyset \vdash_\Sigma \Omega \parallel \Theta$. We omit Φ when we are not interested in the final state of a proof (for example, when the proof is terminated by $\top$).

Given an $\mathscr{E}_{hhf}$-theory (program) Γ and a collection of goals Ω, a query (initial state) will assume the form $\Gamma[\Phi]; \emptyset \vdash_\Sigma \Omega \parallel \emptyset$, where Φ can be considered as a variable instantiated with the possible final states of the execution of Ω.

4.7. *An example*

Let us specify a system of counters endowed with a (nondeterministic) increment operation. Let Σ be a signature with the symbols *new*, *inc*, *counter*, *add* with adequate types and such that (*counter* t) $\in \Pi_{\Sigma_\mathscr{R}}$, for each t. The following clauses, which form Γ, implement the specification:

```
new --> counter 0.                                   (new)

add X Z Y >> inc Z || counter X --> counter Y.       (inc)
```

The former simply rewrites the atomic goal *new* into a new *counter* initialized to 0 (there is no guard in this clause), whereas the latter *synchronizes* an *inc*-message and a *counter*-formula, consuming and rewriting them into the updated counter (the guard simply executes the increment). The *add* predicate is defined by the following Horn-clauses:

```
all >> add X 0 X.

add X Y Z >> add (s X) Y (s Z).
```

In Figure 4 we show a computation (proof) derived using the previous program and the rules of Figure 3. Note that the two *inc* operations can be executed in any order, since the Ω component of a sequent is a multiset: in this case their execution always yields the same final state *counter* 4. Let us point out some observations about the structure of the proof. The main thread of the computation corresponds to the right branch, which models

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

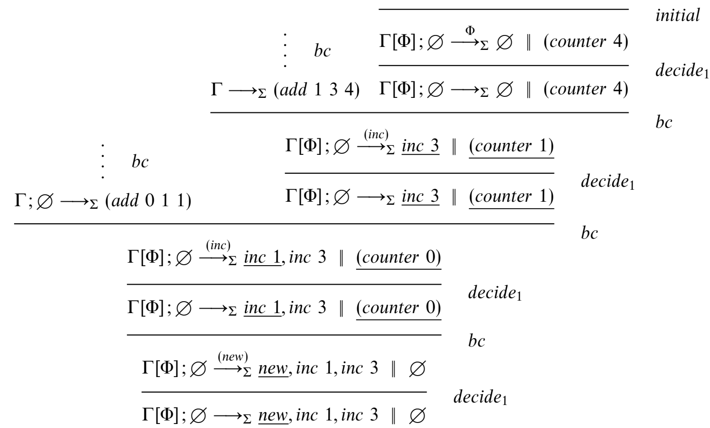

Fig. 4. Here, $0, 1, etc$ denote $0, (s\ 0), etc$.

the evolution of the state of the counter, whereas here, the left branches are used to test the conditional parts of the rules. Furthermore, the left branches are terminated by *initial* axioms matching facts in Γ with atomic goals in Ω; the right branch is terminated by an instance of the *initial* axiom, which matches a particular formula consisting only of $\Pi_{\Sigma_{\mathscr{R}}}$ predicates with the final state of the computation. According to our previous notation, the formula Φ in Figure 4 corresponds to *counter* 4.

In the above outlined scheme the non-determinism is due to the choice of the clause to be applied (which induces a choice on the multiset of atoms on the right-hand side that must be rewritten), and, in the actual implementation, to higher-order unification. In this sense the chosen class of formulae can be considered as an extension of hereditary Harrop formulae (λ-Prolog (Nadathur and Miller 1988)) with the possibility of handling resources and concurrent actions.

Following on the introduction to the use of linear logic as a specification language given earlier, the rest of this paper will be focused on the specification of the operational semantics of an object-based model. Before going into detail, we will briefly recall the main characteristics of concurrent objects.

5. Concurrent object-oriented programming

Programs provide automated management of data. From the users' perspective it is important to know what the effects of executing a program are, *i.e.*, its functionalities, but not how they are actually performed, or in which form the data are stored internally. On the other hand, a programmer needs to adapt the software to a given specification, which in turn, depends on the requirements of the real problem, which may change over time. Thus, it is important to be able to modularly modify programs. Object-orientation provides an interpretation of programs that can ease the tasks of both users and developers.

In this setting, new concepts like *class* and *object* (Wegner 1987), are added to the

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

traditional programming ones. A *class* corresponds to a data type describing what the functionalities of a given part of a program are (data and operations). The users can only access the external interface of operations and data. An *object* corresponds to a given instance of a class; it encapsulates (Snyder 1986) data and the operations (called *methods*) that handle them, and it has an internal *state* whose components can be changed by executing the methods. Objects can receive *messages*, accepting only the ones declared in their classes. A message (method invocation) corresponds to the activation of one method in the object. Methods can refer to the current object by using a *self* pointer (for example, in their definition they can invoke other methods belonging to the same object by referring to the *self* pointer). Classes can be composed by defining hierarchies that allow them to share code among different classes. The *inheritance* relations can be defined in different ways, see Snyder (1986) and Wegner (1987) again (for example, single- and multiple-inheritance, overriding, and so on). As mentioned earlier, objects are created by instantiating classes. Objects usually have unique identifiers and store a private copy of data and operations, with special links to shared ones. Object-based languages are directly based on the notion of object, and they provide operations to create them and to define clones. In this setting, *delegation* is used to share code and data among objects (*i.e.*, an object delegates another object to perform a given operation).

Object-Orientation can fit in both a sequential and a concurrent setting. In a sequential model objects are passive except when they must react to a given message, *i.e.*, the current operation in the execution thread of the program. On the other hand, in a concurrent model – where the unit of parallelism can be a process (as in CSP (Hoare 1978)), a task (as in Ada), a statement (as in Occam), or a goal (as in Concurrent Prolog (Shapiro 1989)) – objects can play different roles. For instance, objects can be considered as *active* entities having their own execution threads activated at the time of creation. An active object can be considered as an advanced form of process. Otherwise, it is possible to employ asynchronous communication, for example, the Actor model (Agha 1986), or to add processes as an orthogonal concept to the object-oriented dimension. Active objects are suitable for sharing the information of a distributed system in a secure way.

In the next section we shall move on to a logical characterization of the previously described features.

6. Specification of concurrent objects

In this section we introduce a linear logic formalization of several different features of object-oriented programming. This specification is based on the view of *objects* as *atomic formulae*. In a higher-order setting this view allows us to capture the key point of the *object-orientation*, that is, *encapsulation*. Before going into the detailed encoding of our *object-based* model, we shall briefly discuss the problems related to *class-based* models.

6.1. *Class-based representation*

In classical logic programming a direct way to design logical specification with object-oriented features is the following: we interpret a set of clauses defining a predicate p as

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

$$\frac{}{\Gamma[\Phi];\varnothing \overset{\Phi}{\longrightarrow}_{\Sigma} \varnothing \parallel (rpoly\ 5\ 4\ P')}\ initial$$

$$\frac{\Gamma[\Phi];\varnothing \overset{\Phi}{\longrightarrow}_{\Sigma} \varnothing \parallel (rpoly\ 5\ 4\ P')}{\Gamma[\Phi];\varnothing \longrightarrow_{\Sigma} \varnothing \parallel (rpoly\ 5\ 4\ P')}\ decide_1$$

$$\frac{\Gamma[\Phi];\varnothing \longrightarrow_{\Sigma} \varnothing \parallel (rpoly\ 5\ 4\ P')}{\Gamma[\Phi];\varnothing \overset{(b)}{\longrightarrow}_{\Sigma} (get\ F\ R\ P) \parallel (rpoly\ 5\ 4\ P')}\ bc$$

$$\frac{\Gamma[\Phi];\varnothing \overset{(b)}{\longrightarrow}_{\Sigma} (get\ F\ R\ P) \parallel (rpoly\ 5\ 4\ P')}{\Gamma[\Phi];\varnothing \longrightarrow_{\Sigma} (get\ F\ R\ P) \parallel (rpoly\ 5\ 4\ P')}\ decide_1$$

$$\frac{\dfrac{\dots}{\Gamma;\varnothing \longrightarrow_{\Sigma} P'\ is\ 5*4} \qquad \Gamma[\Phi];\varnothing \longrightarrow_{\Sigma} (get\ F\ R\ P) \parallel (rpoly\ 5\ 4\ P')}{\Gamma[\Phi];\varnothing \overset{(a)}{\longrightarrow}_{\Sigma} (set\ 5\ 4),(get\ F\ R\ P) \parallel (rpoly\ v\ w\ z)}\ bc$$

$$\frac{\Gamma[\Phi];\varnothing \overset{(a)}{\longrightarrow}_{\Sigma} (set\ 5\ 4),(get\ F\ R\ P) \parallel (rpoly\ v\ w\ z)}{\Gamma[\Phi];\varnothing \longrightarrow_{\Sigma} (set\ 5\ 4),(get\ F\ R\ P) \parallel (rpoly\ v\ w\ z)}\ decide_1$$

Fig. 5. A sample proof: $P = P' = 20,\ F = 5,\ R = 4.$

the class representing the set of objects of the form `p(t)` (McCabe 1992). By exploiting the additional features of linear logic, we can refine this idea as follows: each class can be represented as a program defining a given predicate `classname` such that each single method synchronizes an object of the form `classname State` and a message `message`:

```
cond >> message || classname State --> msg1 |... | msgn || classname State'.
```

Let us give an example.

Example 6.1. We want to specify the class `rpoly` of *regular polygons*. To represent a regular polygon we simply need the number of faces (the rank) and the length of one of the faces. Here, we assume that they are natural numbers. An object of type `rpoly` is defined as an atom `(rpoly face rank perimeter)`. Furthermore, we define two methods: `set`, to correctly initialize the data fields, and `get`, to retrieve the data values. Formally, we introduce the following signature:

```
rpoly:        int → int → int → o.
set:          int → int → o.
perimeter:    int → o.
```

The methods are defined as follows:

```
Perim is Face*Rank  >>
   set Face Rank  || rpoly F R P   --> anti || rpoly Face Rank Perim.    (a)

   get F R P      || rpoly  F R P  --> anti || rpoly F R P.              (b)
```

A possible proof for the query $\Gamma[\Phi];\varnothing \vdash_{\Sigma} (set\ 5\ 4),(get\ F\ R\ P) \parallel (rpoly\ v\ w\ z)$ is shown in Figure 5. To complete the proof, Φ can be set to $(rpoly\ 5\ 4\ 20)$, that is, to the *final state* of the computation. Another possible proof-tree can be obtained in *backtracking* by selecting the method *get* first.

Note that the *conditions* of a $\mathscr{D}$-clause, for example, `Perim is Face*Rank`, are tested on an *auxiliary* branch for which we do not need to know the *final state*. In fact, in this paper the predicates invoked in the conditions are defined only by *Horn logic programs*. For example, the predicate that defines '`is`'. The result of such a program is communicated to the main thread of the computation via shared variables.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

% Objects and Messages

`object`:	identifier → o → o → o.
`frozen`:	identifier → o → o → o.
`send`:	identifier → message → o.
`call`:	identifier → message → o.

Fig. 6. The signature $\Sigma_{\mathscr{R}}$ consists of `object` and `frozen`.

Inheritance can be achieved by using the approach proposed in *LO* (Andreoli and Pareschi 1991), *i.e.*, employing disjunctions ($⅋$) to represent composite objects. For instance, the subclass of *coloured polygons* can be defined by adding a further component (atom) with the `colour` data field to the class of polygons. More precisely, given the signature

`crpoly`: int → int → string → o.
`colour`: string → o,

we add the clauses:

```
anti   || crpoly Face Rank colour  --> anti || rpoly Face Rank 0 | colour colour.

setcolour colour || colour C       --> anti ||  colour colour.
```

Thus, `crpoly`-objects have the form $(rpoly\ F\ R\ P)\ ⅋\ (colour\ C)$. Note that the methods of the superclass (for example, `set` and `get`) still apply to them. Objects in *LO* have the previous form. They are spread over different branches of a proof, and communication is achieved by means of a common blackboard. However, this model does not provide *overriding* of methods. For instance, let us now define the class *circle* as the limit subclass of *rpoly*. We then add the following clauses:

```
   anti   ||  circle Ray Perim --> anti ||  rpoly Ray infinity Perim.

 Perim is 2*Ray*pi_greek >>
   set  Ray  || rpoly R infinity P --> anti || rpoly Ray infinity Perim.
```

In such a hierarchy, however, nothing prevents the method `set` of the superclass `rpoly` from being selected after *opening* a `circle`-object. This might lead to run-time errors, for example, multiplication of `Ray` by `infinity`, which, however, is just a *dummy* constant.

As outlined in Abadi and Cardelli (1996) and Fisher *et al.* (1994), a better understanding of the basic concepts of object-orientation can be achieved by focusing on *object-based* languages. In this setting, classes can be viewed as particular *meta*-objects. We shall introduce this model in the next section.

6.2. *An object-based representation*

Encapsulation is a central point in the object metaphor: *objects are black boxes, with unique identities, containing data and operations and exchanging messages in order to interact with the external world.* In our setting, an object will be defined by an atom of the following form:

`object id state methods`,

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

where `state` is the set of attributes, and `methods` are *clauses* defining the behaviour of the object (see the signature in Figure 6). A message (method invocation) is defined as an atom `call id msg` (see Figure 6 again).

Using the previous assumptions, we shall consider a sequent $\Gamma;\Delta \longrightarrow_{\Sigma} \Omega \parallel \Theta$ as a configuration of an object-based system, where Γ (the program) specifies the *external* behaviour of objects, Θ (the global state) the set of active objects, and Ω the set of active messages. We will clarify the role of Δ in such a representation, later. Below we shall consider a model of computation in which the messages are delivered without any order (for example, in the style of actors (Agha 1986)) and executed in interleaving. The methods of a given object (`object id state methods`) have the form m_1 `&` ... `&` m_n, where each conjunct is a clause of the form

```
cond >> call id Mgs || object id St Ms --> msg_1 |...| msg_n || object id St' Ms'.
```

Such a clause rewrites the object `id` and a message into a (possibly modified) copy of the object and a multiset of new messages. The newly produced messages correspond to the body of the invoked method. To handle such a recursive definition (an object contains its methods defined in terms of the object itself) we have to abstract from some of the components of the object-representation. To clarify, let us consider the following simple method:

```
call id (get V) || object id V M --> anti || object id V M.
```

We will assume that it retrieves the value stored in the object. On the other hand, the methods `M` must be unfolded only at the time of execution. To achieve this, we rewrite the method as follows:

```
pi V \ pi M \ (call id (get V) || object id V M --> anti || object id V M),
```

using quantification over variables of type o to leave the methods in the nested object-atoms unspecified. Therefore, we construct the object as follows:

```
object id s
  (pi V \ pi M \
     (call id (get V) || object id V M --> anti || object id V M)
   ).
```

Note that the third parameter of the `object` predicate is a (universally quantified) $\mathscr{D}$-clause. For simplicity we assume that the state denotation, `s` in the example, is a term of type o. For instance, the internal state `s` can simply be a tuple `(st A B ...)`, where `st` is any constructor with target type o.

6.3. *Method invocation*

In order to execute a method it is necessary to:

i) synchronize the object and the message;

ii) fire the methods, *i.e.*, move them to the top level on the left-hand side of sequents (methods are indeed clauses);

iii) execute a backchaining step over the $\mathscr{D}$-clauses defining the methods.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

To prevent anomalies we need to enrich the syntax of our representation (see Figure 6). The new predicate `send` is used to represent messages before step *i*, that is, messages that require the corresponding definition to be fired. The predicate `call` will be used instead to represent messages during steps *ii* and *iii*, that is, messages whose definitions have already been fired. Finally, to prevent the execution of a `call` from being delayed after step *ii*, we inhibit other receptions by *freezing* the considered object after step *i*. Methods then take on the following form:

```
call id Msg || frozen id V Ms --> msg1 |...| msgn || object id V' Ms'.
```

Note that, if we do not freeze objects before steps *ii* and *iii*, an interleaved execution of another message could change the structure of the original object while a copy of its old methods exists. In synthesis, `send` will be used as an *external* primitive to invoke methods, whereas `call` will serve to internally dispatch the message. Formally, such semantics is formalized through a single $\mathscr{E}_{hhf}$-rule.

Definition 6.1. (Self-application) The semantics of method-application (steps *i*, *ii* and *iii* of Section 6.3) is defined by the following clause:

```
send Id Msg || object Id St Ms --> (Ms -o call Id Msg) || frozen Id St Ms.
```

Step *i* is expressed by the head of the previous clause. Note that such a rule can be applied in *any* context, that is, with any number of messages and objects in the global state, provided the object `Id` exists. Step *ii* is achieved by the goal `(Ms -o call Id Msg)`: the methods are fired and they are *consumed* right after their execution. Finally, after adding `call Id Msg` to the current operations and freezing the object, we start up the execution of the method. In fact, by hypothesis, the methods in `Ms` *define* the predicate `call` for the object `Id`. Note that `Ms` is an &-conjunction of methods. The nondeterminism in the choice of one of its conjuncts (see rule *bc*) guarantees the selection of the right clause.

Example 6.2. Let us reformulate the `polygon` objects of example 6.1 in the new setting.

```
object id (s face rank perim) Ms
 where Ms =
  pi Face \ pi Rank \ pi Perim \ pi F \ pi R \ pi P \
   (Perim is Face*Rank  >>
     (call id (set Face Rank)  || object id (s F R P)  -->               (a)
                         anti   || object id (s Face Rank Perim)))
        &
     (call id (get Face Rank Perim)) || object  id (s Face Rank Perim)  -->  (b)
                                 anti || object  id (s Face Rank Perim)
```

Note that the methods incorporate the identifier of the object to which they belong. Below we shall denote such an object by $\mathrm{o}_{f,r,p}$ and the corresponding frozen object by $\mathrm{F}_{f,r,p}$, where f,r,p are the values of the attributes `Face`, `Rank`, `Perimeter`. Then, the query $\Gamma[\Phi];\varnothing \vdash_\Sigma (\mathit{send}\ \mathit{id}\ (\mathit{set}\ 2\ 3)) \parallel \mathrm{o}_{5,4,w}$ has the (partial) $\mathscr{E}_{hhf}$-proof tree of Figure 7. Note that the method `(b)` (contained in `Ms`) is first fired by the *send* rule and then *removed* by the *decide*$_2$ rule. Since the *right* instance of b is in $\langle Ms \rangle$, the *bc* step over `Ms` succeeds and it achieves the desired effect.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma;\varnothing \longrightarrow_\Sigma X\ is\ 2*3} \qquad \begin{array}{c}\vdots\\ \Gamma[\Phi];\varnothing \longrightarrow_\Sigma \Omega \parallel \mathrm{O}_{2,3,X},\Theta\end{array}}{\Gamma[\Phi];\varnothing \stackrel{Ms}{\longrightarrow}_\Sigma (call\ id\ (set\ 2\ 3)),\Omega \parallel \mathrm{F}_{5,4,w},\Theta}\ bc\ over\ (b)}{\Gamma[\Phi];Ms \longrightarrow_\Sigma (call\ id\ (set\ 2\ 3)),\Omega \parallel \mathrm{F}_{5,4,w},\Theta}\ decide_2}{\Gamma[\Phi];\varnothing \longrightarrow_\Sigma Ms \multimap (call\ id\ (set\ 2\ 3))\ \parr\ \mathrm{F}_{5,4,w},\Omega \parallel \Theta}\ \parr_r\ +\multimap_r}{\Gamma[\Phi];\varnothing \stackrel{(send)}{\longrightarrow}_\Sigma (send\ id\ (set\ 2\ 3)),\Omega \parallel \mathrm{O}_{5,4,w},\Theta}\ bc}{\Gamma[\Phi];\varnothing \longrightarrow_\Sigma (send\ id\ (set\ 2\ 3)),\Omega \parallel \mathrm{O}_{5,4,w},\Theta}\ decide_1$$

Fig. 7. A sample (piece of) proof: $X = 6$.

In the next section we shall study the proof-theoretic properties of the object-calculus resulting from the above described encoding.

6.4. *Aspects of concurrency*

In this section, we shall focus on the relation between provability and concurrency aspects of the proposed model. Specifically, our aim is to define an interleaving operational semantics for the object calculus introduced in the previous section, by means of a further refinement of our sequent-calculus presentation of $\mathscr{E}_{hhf}$.

Throughout the rest of the section, we shall consider Γ as the singleton consisting of the clause of Definition 6.1, Θ as a multiset of `object` and `frozen` atoms, and Ω as a multiset of `send` and `call` atoms. Furthermore, we say that a global state Θ is consistent if the identifiers of the object occurring in Θ are different from each other. We then introduce the following derived rule.

Definition 6.2. (The *send* rule) The *send* rule consists of the following sequence of rules:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma[\Phi];Meths,\Delta \vdash_\Sigma \mathtt{call\ id\ m},\Omega \parallel \mathtt{frozen\ id\ st\ Meths},\Theta}{\Gamma[\Phi];Meths,\Delta \vdash_\Sigma \mathtt{frozen\ id\ st\ Meths}\ \parr\ \mathtt{call\ id\ m},\Omega \parallel \Theta}\ \parr_r}{\Gamma[\Phi];\Delta \vdash_\Sigma Meths \multimap (\mathtt{frozen\ id\ st\ Meths}\ \parr\ \mathtt{call\ id\ m}\ ),\Omega \parallel \Theta}\ \multimap_r}{\Gamma[\Phi];\Delta \stackrel{(send)}{\longrightarrow}_\Sigma \mathtt{send\ id\ m}\ ,\Omega \parallel \mathtt{object\ id\ st\ Meths},\Theta}\ bc}{\Gamma[\Phi];\Delta \vdash_\Sigma \mathtt{send\ id\ m}\ ,\Omega \parallel \mathtt{object\ id\ st\ Meths},\Theta}\ decide_1$$

We observe that the right rules occurring immediately above the *bc* rule can always be permuted so as to obtain the previous configuration. Thus, without loss of generality, given a sequent $\Gamma[\Phi];\Delta \vdash_\Sigma \Omega \parallel \Theta$ we can restrict our attention to proofs in which each application of a backchaining step (over the clause of Definition 6.1) matches the previous derived rule. We denote such a derived rule as *send* rule.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

Lemma 6.3. (Permutability of the *send* rule) Let Γ consist of the single $\mathscr{E}_{hhf}$-clause of Definition 6.1, and Θ_o and Ω be multisets of objects and messages. Let δ be an $\mathscr{E}_{hhf}$-proof for $s = \Gamma[\Phi]\,;\Delta \vdash_\Sigma \Omega \parallel \Theta$, such that the last rule of δ is a *send* rule, applied to an object named *id* and introducing $Meths_{id}$ on the left-hand side of its premise. Finally, let us assume that Δ does not contain any other methods labelled by *id* and that each state Θ_i occurring in δ is consistent. Then, a proof γ for s exists, where, $Meths_{id}$ is consumed immediately after the application of the *send* rule that introduces it.

Proof. Let $send_{id}$ be the last occurrence of *send* applied to the object denoted by *id*. The proof is by induction on the number of rule applications between the $send_{id}$ rule and the corresponding *consumption* of the methods $Meths_{id}$. The proof of the base is trivial. If the distance is greater than zero, the proof is by cases on the rule R that occurs immediately above the instance of the $send_{id}$ rule taken into consideration. There are two cases.

— R is an instance $send_{id'}$ of the *send* rule and $id' \neq id$, since *id* is frozen by $send_{id}$. It is easy to see that two instances of the *send* rule (over different objects) permute. Therefore, after permuting $send_{id'}$ and $send_{id}$, we conclude by applying the induction hypothesis.
— R is an instance of the *bc* rule over the methods $Meths_{id'}$ of the object `frozen id'` (where, by hypothesis, $id' \neq id$) followed by *deterministic* application of the $\mathcal{B}_r$ rule, as shown in the following picture:

$$\dfrac{\dfrac{\begin{array}{c}\eta\\ \Gamma[\Phi]\,;Meths_{id},\Delta \vdash_\Sigma \texttt{call id}, m_1 \ldots m_n, \Omega \parallel \texttt{frozen id}, \texttt{object id}', \Theta\end{array}}{\Gamma[\Phi]\,;Meths_{id}, Meths_{id'},\Delta \vdash_\Sigma \texttt{call id}, \texttt{call id}', \Omega \parallel \texttt{frozen id}, \texttt{frozen id}', \Theta}\ \vdots\, R}{\Gamma[\Phi]\,;Meths_{id'},\Delta \vdash_\Sigma \texttt{send id}, \texttt{call id}', \Omega \parallel \texttt{object id}, \texttt{frozen id}', \Theta}\ send$$

where, $\texttt{frozen id}' \mathcal{B}\, \texttt{call id}' \circ\!\!- \texttt{object id}' \mathcal{B}\, m_1 \ldots m_n \in \langle Meths_{id'} \rangle$. In this case, we can re-arrange the proof as follows:

$$\dfrac{\dfrac{\begin{array}{c}\eta\\ \Gamma[\Phi]\,;Meths_{id},\Delta \vdash_\Sigma \texttt{call id}, m_1,\ldots,m_n, \Omega \parallel \texttt{frozen id}, \texttt{object id}', \Theta\end{array}}{\Gamma[\Phi]\,;\Delta \vdash_\Sigma \texttt{send id}, m_1 \ldots m_n, \Omega \parallel \texttt{object id}, \texttt{object id}', \Theta}\ send}{\Gamma[\Phi]\,;Meths_{id'},\Delta \vdash_\Sigma \texttt{send id}, \texttt{call id}', \Omega \parallel \texttt{object id}, \texttt{frozen id}', \Theta.}\ \vdots\, R$$

By applying the induction hypothesis on the subproof η, we conclude that γ satisfying the thesis exists.

Note that in each case the length of the resulting proof γ is equal to the length of δ. □

In order to emphasize the operational aspects of the $\mathscr{E}_{hhf}$-theory previously introduced, we define the following operational semantics.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

Definition 6.4. (Interleaving semantics) Let $\Omega, \Omega', \Theta, \Theta'$ be multisets as specified above. The relation $\rightsquigarrow_o$ is defined as follows.

$$\langle \Omega \mid \Theta \rangle \quad \rightsquigarrow_o \quad \langle \Omega' \mid \Theta' \rangle$$

iff `(send id n)`$\in \Omega$ and `(object id st ms)`$\in \Theta$,
`(call id m || object id st mts --> msgs|| object id st' mts')`$\in \langle$`mts`$\rangle_\Sigma$,
`msgs =` m_1 `| .. |` m_n,
$\Omega' = (\Omega \backslash \{$`(send id n)`$\}) \cup \{m_1, \ldots, m_n\}$,
$\Theta' = (\Theta \backslash \{$`(object id st ms)`$\}) \cup \{$`(object id st' ms')`$\}$.

With $\rightsquigarrow_o^\star$ the transitive closure of $\rightsquigarrow_o$, the following result holds.

Proposition 6.5. (Provability and operational semantics) Let Γ consist of the single $\mathscr{E}_{hhf}$-clause of Definition 6.1, Θ_o be a multiset of `object`-atoms, and Ω be a multiset of `send`-atoms. Then, $\Gamma[\Phi]; \varnothing \vdash_\Sigma \Omega \parallel \Theta_o$ has an $\mathscr{E}_{hhf}$-proof iff $\langle \Omega \mid \Theta_o \rangle \rightsquigarrow_o^\star \langle \varnothing \mid \Theta_f \rangle$ and $\Phi \multimap \Theta_f$ is provable in $\mathscr{E}_{hhf}$.

Proof. The proof is by induction on the length of a proof and by case analysis based on Lemma 6.3 □

We shall use this semantics (and its extensions) to simplify the description of the object behaviour. In the rest of this section we will enrich the basic model with new features.

6.4.1. *Classes.* Classes can be viewed as templates for the creation of *clones* of a given object. We can enrich the $\mathscr{E}_{hhf}$-theory Γ with a very simple type of *class* declaration. We first introduce the predicate `new` with type `Class -> o -> identifier -> o`. A class declaration then assumes the following form:

```
new classname InitState Id || anti --> anti || object Id InitState Mts.
```

Here `InitState` represents the initial values of the attributes of the object being created. An invocation of the *primitive*, that is, visible to any object, `new c s id` generates a new instance of the class named `c`, with an initial state `s`, and an identifier `id`. Below we enrich the possible types of messages with a `new`-atom.

Example 6.3. The `rpoly` class can be defined as:

```
new rpoly (st F R P) Id || anti --> anti || object Id (st F R P) Ms.
```

where `Ms` is defined as in the Example 6.2.

This model captures many other features that are typical of object-oriented programming. We shall discuss them next.

6.5. *Hiding*

As shown in Hodas and Miller (1989) and Miller (1989b), universal quantification can be used to create *local* data structures. In our setting there are three possible ways of using universal quantifications:

— before an object declaration to *hide data fields*;
— before an object declaration to provide *private methods*;
— before the body of a method to provide creation of *private objects*.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

The first two points require the following modification of the previously introduced `new` definition:

```
new classname InitState Id || anti -->
      pi X \( anti || object Id InitState' Mts ).
```

The following examples will illustrate how to exploit such features.

6.5.1. *Hiding of data fields.* In the model presented above, any object may freely access the attributes of the other objects. Let us consider the following rule:

```
break State || object Id State Mts --> all || anti.
```

Through unification, the predicate `break State` may break the encapsulation of any of the objects in the current state. In order to prevent such a violation, we shall employ the universal quantifier as in the example below.

Example 6.4. The `rpoly` class of Example 6.3 can be re-defined as:

```
new rpoly (st F R P) Id || anti -->
         pi st \ (anti || object Id (st F R P) Ms).
```

where `Ms` is defined as in example 6.2. Note that the quantification is defined over a predicate name, namely `st`. This is allowed by the syntax of $\mathscr{E}_{hhf}$. Such a higher-order quantification hides the occurrences of a constructor in the definition of the object.

For instance, the `break` predicate cannot be applied these objects since constants introduced with $\forall_r$ (for example, the one associated with `st` when the body of `new` is executed) cannot be bound to free variables outside the scope of the quantifier, see Miller (1989b) and Hodas and Miller (1989).

6.5.2. *Hiding of methods.* The same idea can be applied in order to define private methods. A method is private in o if it can be invoked only by other methods of o. To accomplish this it suffices to quantify over the names of the private methods. For instance, in the following declaration the method `pub` (with type `o`) is public while `m` (with type `o`) can be invoked only by other methods of the same object.

```
(new private St Id || anti -->
         (pi m \ (anti || object Id St (Ms m)))

where (Ms m) =
   pi S \ pi M \
    (call Id m    || frozen Id S M  --> ... || ...)
     &
    (call Id pub  || frozen Id S M  --> send Id m || object Id S M).
```

The third use of universal quantification will be illustrated in the following section.

6.5.3. *Creation of local objects* Universal quantification in the body of a method can be used to create data structures that are local to that method. In particular, we can employ this feature to create *local objects*. Consider the following general pattern:

```
cond >> call id msg || frozen id st ms --> pi x \ Body.
```

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

Now, if x is associated to the identifier of a newly created object, the object will automatically become local to the method. The meta-rule for the self-application is not affected by this modification. However, the *send* rule of Definition 6.2 must be enriched so as to account for possible applications of the $\forall_r$ rule. We redefine the relation $\rightsquigarrow_o$ according to this idea. The new relation is denoted by $\overset{\forall}{\rightsquigarrow}_o$. By a case analysis similar to the one in Proposition 6.5, it is easy to state the correspondance between $\overset{\forall*}{\rightsquigarrow}_o$-transitions and $\mathscr{E}_{hhf}$-proofs. We shall consider this feature in the example given in the next section.

6.6. *Expressiveness of the calculus*

To illustrate the power of the resulting *object* calculus, we note that a method can modify the object to which it belongs. For instance, the method

```
call kill Id  ||  frozen Id St Ms --> anti ||  anti.
```

deletes the object that receives it. Similarly, we can think of methods that *dynamically* modify the attributes or even the methods of the object.

We shall clarify this point with some examples.

Example 6.5. (Factorial objects) Let `factclass` be a class of objects that, upon reception of the message `(fact N Obj)` (`N` is an integer and `Obj` is an object identifier), send the factorial of `N`, that is, `N!=N*N-1*...`, to `Obj`. Below we present an implementation in which the same object can manage many requests at the same time. To implement this type of objects, it is necessary to define `slave`-objects that perform the actual computation of the factorial, allowing their masters to process the other possible incoming requests. Let us consider the following signature:

`factclass,slave, receiver:`	o.
`fact,st:`	int → identifier → o.
`eval:`	int → o.

The definition of `factclass`-objects is given as follows:

```
% Class of "factorial objects"
new factclass St Id || anti --> anti || object Id St Ms.

where Ms =

 (pi S \ pi M \ pi N\ pi N1 \ pi O\
        (((N =< 2) >>
              call Id (fact N O) || frozen Id S M -->
                      send O  (eval N)   || object Id S M)
         &
         ((N > 2, N1 is N - 1) >>
            call Id (fact N O) || frozen Id S M  -->
             (pi x \(new slave (st N O) x | send Id (fact  N1 x)))
                                               || object Id S M
         ))).
```

These objects have only one method (`fact N O`) defined by two clauses: if `N` is less or equal to 2, then `(eval N)`, the result of the computation is immediately sent to the object

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

`O`; in the other cases a new `slave` object `x` is created and the message `fact N-1 x` is sent to `x`. Slave and receiver objects are defined as follows:

```
% Class of "slave objects"
new slave St Id || anti --> anti || object Id St Ms

where Ms =

   (pi N \ pi N1 \ pi N2 \ pi Ms \ pi O \
         ((N2 is N1 * N) >>
                 call Id (eval N1) || frozen Id (st N O) Ms -->
                         send O  (eval N2) || anti)).

% Class "receiver"
new receiver St Id || anti --> anti || object Id St Ms

where  Ms =

   (pi N \ pi S \ pi M \
         call Id (eval N) || frozen Id S M  --> anti || frozen Id (eval N) M).
```

They both expect `(eval N)` messages: a slave object executes a part of the computation of `N!`, it sends the result to another slave object and then it disappears, whereas a receiver simply stores the result.

Some abbreviations are needed in order to describe possible derivations based on the previous definitions. Let $\mathrm{O}_{id}[\mathrm{V}, id']$ be a `slave`-object with identifier id and state (V, id'). Now, let $\Theta = \{\mathrm{R}_{id_r}, \mathrm{F}_{id_f}\}$, where R_{id_r} is a *receiver* and F_{id_f} is a *factclass* object. The following picture describes a $\overset{\forall}{\leadsto}_o$-derivation that starts from the state $\langle \Omega \mid \Theta \rangle$, where $\Omega = \{\texttt{send id}_\texttt{f}\ \texttt{(fact 4 id}_\texttt{r}\texttt{)}\}$, that is, we ask the object id_f to compute $4!$ and to send the result to id_r:

$$\langle \Omega \mid \Theta \rangle \overset{\forall}{\leadsto}_o \langle \texttt{new slave (st 4 id}_\texttt{r}\texttt{) x}, \texttt{send id}_\texttt{f}\ \texttt{(fact 3 x)} \mid \Theta \rangle \overset{\forall}{\leadsto}_o$$
$$\overset{\forall}{\leadsto}_o \langle \texttt{send id}_\texttt{f}\ \texttt{(fact 3 x)} \mid \mathrm{O}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o$$
$$\overset{\forall}{\leadsto}_o \langle \texttt{new slave (st 3 x) y)}, \texttt{send id}_\texttt{f}\ \texttt{(fact 2 y)} \mid \mathrm{O}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o$$
$$\overset{\forall}{\leadsto}_o \langle \texttt{send y (eval 2)} \mid \mathrm{O}_y[3, x], \mathrm{O}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \langle \texttt{send x (eval 6)} \mid \mathrm{O}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o$$
$$\overset{\forall}{\leadsto}_o \langle \texttt{send id}_\texttt{r}\ \texttt{(eval 24)} \mid \Theta \rangle \overset{\forall}{\leadsto}_o \langle \varnothing \mid \Theta' \rangle.$$

The final state Θ' is obtained by storing 24 in the *receiver*-object. x and y are newly introduced names for the local *slave*-objects. A `factclass`-object can process many *fact*-messages at the same time since it delegates the computation to its `slave`. For instance, let Θ be as before and add a new message $(\texttt{fact 3 id}_\texttt{r})$ to Ω. We obtain a computation like the following one:

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

$$
\begin{array}{c}
\langle \mathtt{send\ id_f\ (fact\ 4\ id_r), send\ id_f\ (fact\ 3\ id_r)} \mid \Theta \rangle \overset{\forall}{\leadsto}_o \\
\langle \mathtt{new\ \ slave\ (st\ 4\ id_r)\ x, send\ id_f\ (fact\ 3\ x), send\ id_f\ (fact\ 3\ id_r)} \mid \Theta \rangle \overset{\forall *}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ id_f\ (fact\ 3\ x), new\ \ slave\ (st\ 3\ id_r)\ x', send\ id_f\ (fact\ 2\ x')} \mid o_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{new}\ \ \mathit{slave}\ (\mathit{st}\ 3\ x)\ y), \mathtt{send\ id_f\ (fact\ 2\ y), send\ id_f\ (fact\ 2\ x')} \mid o_x[4, id_r], o_{x'}[3, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ x'\ (eval\ 2), send\ y\ (eval\ 2)} \mid o_y[3, x], o_x[4, id_r], o_{x'}[3, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\langle \mathtt{send\ id_r\ (eval\ 6), send\ x\ (eval\ 6)} \mid o_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ id_r\ (eval\ 24), send\ id_r\ (eval\ 6)} \mid \Theta \rangle.
\end{array}
$$

Here the identifiers x, y, x', y' are new names for the local objects of the two concurrent computations. Note that two messages are executed in interleaving. To conclude, we would like to go back to the problem of inheritance outlined at the beginning of this section.

6.7. *Inheritance at the object level*

Being in an object-based setting allows us to capture a very basic notion of inheritance. Let us extend the formalism above with two new *primitive* operations `extend` and `override` defined as follows:

```
extend Id NMs || object Id St Ms --> anti || object Id St (Ms & NMs).

override Id OMs NMs || object Id St OMs --> anti || object Id St NMs.
```

The meaning of the first clause should be clear: it extends the current set of methods with new ones. On the other hand, the second clause *overrides* the methods specified through the parameter `Ms` with the new definitions given in `OMs`.

Using such definitions, it is now possible to define hierarchies of classes on a delegation-based mechanism. Below we assume that methods are paired with their name and kept on a list $(name_1, Def_1) :: \ldots :: (name_n, Def_n) :: nil$. It is not difficult to modify the clause of Definition 6.1 in order to handle the new representation. We can rewrite the `rpoly` and `circle` classes of the example at the end of Section 6.1 as follows:

```
 new rpoly  (st Face Rank Perim) Id || anti -->
        anti || object Id (st Face Rank Perim) Ms.

where Ms is the list (set,M_set)::(get,M_get)::nil

 new circle (st Ray Perim) Id || anti -->
        new rpoly (st Ray infinity Perim) Id | override Id OMs RMs || anti

      OMS=(set,M1)::(get,M2)::nil

      RMs=(set,M_set_circle)::(get,M2)::nil
```

where for the sake of brevity, we omit the definitions of `M_set`, `M_get` and `M_set_circle`, they can easily be written by following the examples shown in the previous sections. Note that in the definition of `circle` we do not have to specify the code of the old and the new methods. In fact, we can distinguish them by the name. Therefore, `M1` and `M2` are simply two free variables that will be instantiated with the code of the corresponding methods.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

To summarize, in this section we have shown how to define a rich object-based concurrent calculus in higher order linear logic by means of numerous examples. Without going into a detailed description, we simply mention that other features such as *classes* as *special meta-objects* can also be derived. Finally, note that the presented encoding integrates logical features peculiar to linear logic (for example, state-based computations) with features introduced in previous works in higher order logic programming (for example, the use of universal quantification) in a natural way.

7. Related work

The central point in the approach integrating OO and LP is the definition of a logical counterpart of objects, and, in particular, of objects with an internal state. The various approaches and the different ways of representing objects are presented in Brogi *et al.* (1991). Here we shall recall some of them.

In Login (Aït-Kaci and Nasr 1986), an extension of Prolog with order-sorted type structures, objects are viewed as records, represented by *typed variables*, containing their attributes. This is only a partial view of encapsulation (*i.e.*, methods are not part of the object). In McCabe (1992), objects are viewed as logical *theories*, and methods as clauses. In this approach the attributes of the objects cannot be updated. Using the same idea, Miller (1989a) uses an extension of logic programming with embedded implication to define modules with hidden structures. Hodas and Miller (1989) uses embedded implications to model objects that can be modified monotonically (*i.e.*, by adding attributes and definitions). Universal quantification is used to hide data structures.

Chen and Warren (1988) views objects as non-logical variables whose value can be modified by means of an assignment construct. On the other hand, Conery (1988) views objects as atomic formulae. In this approach Horn Clauses are extended to multiple-headed clauses. The evolution of an object (an atomic formula) is determined by executing a clause, *i.e.*, rewriting the object and the message into a new object. The semantics of this approach finds a natural counterpart in the linear logic setting, as has been discussed in the course of the paper. In Concurrent Prolog (Shapiro 1989), goals are considered as a collection of concurrent processes, each of which is represented by an atomic formula, and shared variables are used as communication channels. In Vulcan (Kahn *et al.* 1987), a concurrent object-oriented language based on Concurrent Prolog program clauses are seen as rewriting rules. State updates are modelled by rewriting the atoms representing *state* components. This is the same idea as is used in Polka, an OO presentation of Parlog (Clark and Gregory 1986).

A similar approach has been taken in the specification of OO paradigms in the setting of term rewriting systems, such as, for instance, in the Conditional Rewriting Logic (Meseguer 1990).

Natural extensions to the previously mentioned ideas have been developed in the setting of linear logic programming. As we discussed in the paper, *LO*, the linear logic language proposed in Andreoli and Pareschi (1991), is one of the most significant approaches in the field. *LO* extends previous work relating traditional logic programming and object-orientation (for example, Conery (1988) and McCabe (1992)) in different ways. Specifically,

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

an object in *LO* is an an aggregation of data fields (represented by disjunctions of atomic formulae), and a class is defined by collections of multiple-conclusion clauses synchronizing objects and messages. Here a sequent represents a single object and a proof represents the evolution of a collection of objects (spread over different branches of a proof-tree). In our setting, objects are higher-order atomic formulae embedding their methods, defined, in turn, as multi-headed clauses. In addition, sequents represent collections of objects. Our view captures the essence of object-based languages, *i.e.*, encapsulation and privacy of data (private methods and data fields), as well as dynamic overriding of methods. Not all these features were captured by the *LO* object-model.

From a general point of view, our approach is strictly related to the work in Kobayashi and Yonezawa (1994a) and Kobayashi and Yonezawa (1994b), where the authors present a higher-order process calculus (*i.e.*, where a process can be sent as a value to another process) modelled in higher-order classical linear logic. They have extended this idea in Kobayashi and Yonezawa (1994b) to model a concurrent object-based calculus and its type inference system. In contrast with our approach, they modelled the basic computational mechanism of the calculus (*i.e.*, method invocation) through the addition of a fixpoint operator to represent recursive object expressions. Object expressions are unfolded at the time of the method execution. In our approach, method invocations are modelled by exploiting the combined power of the linear implication (to fire methods) as well as of higher-order quantification used in encoding objects (*i.e.*, methods are viewed as terms or formulae depending on the context in which they occur). It is our opinion that our work gives a more complete view of the expressiveness of linear logic programming languages and, in particular, of the uniform-proof based approach.

Bugliesi *et al.* (1996) proposes an extension of logic programming with objects. In this setting objects are nothing but (higher-order) record terms that can be handled by pure programs with some special primitive whose semantics finds a counterpart in a fragment of Forum (namely Horn-clauses enriched by $\multimap$ goals). In this approach there is no notion of *global state* of the computation and, in addition, concurrent executions are not taken into consideration.

To conclude, we would like to point out that the present paper extends the preliminary works Delzanno and Martelli (1995) and Delzanno and Martelli (1996), where the rule for method invocation based on backchaining was introduced, but inheritance and overriding were not discussed, and Boudinet and Galmiche (1996), where alternative models to uniform-proofs based on FILL were proposed as foundations of linear logic specification languages.

8. Conclusions

The proof-theoretic analysis of linear logic has led to interesting specification languages based on the *proofs-as-computations* metaphor. In this paper, we have illustrated the key points of this refinement process through an overview of the various fragments, proof systems, and proof-search strategies proposed in the literature. In order to introduce a concrete language to be used to specify advanced aspects of programming, we have adopted the notion of *goal-directed* provability, which is at the root of logic programming.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

Specifically, we have used the fragment of Forum (Miller 1996) called $\mathscr{E}_{hhf}$(Delzanno 1997; Delzanno and Martelli 1998) as a basis to provide a simple and readable form of linear logic executable specifications. This language is the platform for the specification of the basic features of concurrent objects presented in the final section of the paper. Concerning this point, we have tried to fill the gaps of previous approaches relating logic programming, linear logic and object-orientation, as we discussed in the previous section.

Proof theory is the essential *tool* needed to study all these aspects. First of all, it gives a theoretical justification for the development of refined proof-systems such as Forum and $\mathscr{E}_{hhf}$. Furthermore, when restricted to a particular logic model, for example, the encoding of the object-based model in Section 6, it provides the means for proving important computational properties, such as the equivalence of a proof in the logical model and of an execution in the computational model. As remarked in McDowell and Miller (1997), further research in this direction might yield interesting results in the automated analysis of complex programming systems.

Appendix A. Proof systems for linear logic

The language of LL consists of a set of *terms* defined over a denumerable set of variables, a countable set of *atoms* and the following set of formulae built over the *logical operators* $\{0, 1, \bot, \top, ()^{\bot}, !, ?, \otimes, \parr, \&, \oplus, \forall, \exists\}$:

$$F ::= \mathbf{0}|\mathbf{1}| \bot |\top|a|a^{\bot}|?F|!F|F \otimes F|F \parr F|F\&F|F \oplus F|\forall xF|\exists xF.$$

The negation of the formulas is defined by the following equalities:

$$\begin{array}{rcl}
A^{\bot\bot} & = & A \\
\mathbf{1}^{\bot} & = & \bot \\
\bot^{\bot} & = & \mathbf{1} \\
\top^{\bot} & = & \mathbf{0} \\
\mathbf{0}^{\bot} & = & \top \\
(A \otimes B)^{\bot} & = & A^{\bot} \parr B^{\bot} \\
(A \parr B)^{\bot} & = & A^{\bot} \otimes B^{\bot} \\
(A\&B)^{\bot} & = & A^{\bot} \oplus B^{\bot} \\
(A \oplus B)^{\bot} & = & A^{\bot}\&B^{\bot} \\
(\forall xA)^{\bot} & = & \exists xA^{\bot} \\
(\exists xA)^{\bot} & = & \forall xA^{\bot} \\
(!A)^{\bot} & = & ?A^{\bot} \\
(?A)^{\bot} & = & !A^{\bot}.
\end{array}$$

In this section we present two formulations of linear logic: a one-sided sequent calculus and a two-sided sequent calculus. The laws of negation can be used to go from one to the other.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

A.1. *One-sided sequent calculus*

$$\frac{}{\vdash A, A^{\perp}}\ Id \qquad \frac{\vdash A, \Gamma \quad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}\ Cut$$

$$\frac{\vdash \Delta}{\vdash ?A, \Delta}\ w? \qquad \frac{\vdash ?A, ?A, \Delta}{\vdash ?A, \Delta}\ c?$$

$$\frac{\vdash A, \Gamma_1 \quad \vdash B, \Gamma_2}{\vdash A \otimes B, \Gamma_1, \Gamma_2}\ \otimes \qquad \frac{\vdash A, B, \Gamma}{\vdash A \mathbin{⅋} B, \Gamma}\ ⅋ \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1} \qquad \frac{\vdash \Gamma}{\vdash \perp, \Gamma}\ \perp$$

$$\frac{\vdash A, \Gamma \quad \vdash B, \Gamma}{\vdash A \& B, \Gamma}\ \& \qquad \frac{\vdash A, \Gamma}{\vdash A \oplus B, \Gamma}\ \oplus_1 \qquad \frac{\vdash B, \Gamma}{\vdash A \oplus B, \Gamma}\ \oplus_2 \qquad \frac{}{\vdash \top, \Delta}\ \top$$

$$\frac{\vdash A, ?\Gamma}{\vdash !A, ?\Gamma}\ ! \qquad \frac{\vdash A, \Gamma}{\vdash ?A, \Gamma}\ ? \qquad \frac{\vdash A[y/x], \Gamma}{\vdash \forall x A, \Gamma}\ \forall \qquad \frac{\vdash A[t/x], \Gamma}{\vdash \exists x A, \Gamma}\ \exists$$

In $\forall$ rule, y is not free in Γ and in A if y is different from x.

A.2. *Two-sided sequent calculus*

Two-sided sequents have the form $\Gamma \vdash \Delta$ where Γ and Δ are multisets of linear formulae, that is, formulae built on the linear primitives $0, 1, \perp, \top, ()^{\perp}, \otimes, ⅋, \&, \oplus, \forall, \exists, ?, !$.

For a complete presentation of this system, refer to Girard (1987).

$$\frac{}{A \vdash A}\ id \qquad \frac{\Gamma \vdash A, \Delta \quad A, \Lambda \vdash \Theta}{\Gamma, \Lambda \vdash \Delta, \Theta}\ cut$$

$$\frac{\Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}\ !w_L \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, ?A}\ ?w_R \qquad \frac{!A, !A, \Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}\ !c_L \qquad \frac{\Gamma \vdash \Delta, ?A, ?A}{\Gamma \vdash \Delta, ?A}\ ?c_R$$

$$\frac{A, B, \Gamma \vdash \Delta}{A \otimes B, \Gamma \vdash \Delta}\ \otimes_L \qquad \frac{\Gamma \vdash \Delta, A \quad \Lambda \vdash \Theta, B}{\Gamma, \Lambda \vdash \Delta, \Theta, A \otimes B}\ \otimes_R \qquad \frac{}{\vdash \mathbf{1}}\ 1_R \qquad \frac{\Gamma \vdash \Delta}{\mathbf{1}, \Gamma \vdash \Delta}\ 1_L$$

$$\frac{A, \Gamma \vdash \Delta \quad B, \Lambda \vdash \Theta}{A \mathbin{⅋} B, \Gamma, \Lambda \vdash \Delta, \Theta}\ ⅋_L \qquad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \mathbin{⅋} B}\ ⅋_R \qquad \frac{}{\perp \vdash}\ \perp_L \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \perp}\ \perp_R$$

$$\frac{\Gamma \vdash \Delta, A \quad B, \Lambda \vdash \Theta}{A \multimap B, \Gamma, \Lambda \vdash \Delta, \Theta}\ \multimap_L \qquad \frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \multimap B}\ \multimap_R$$

$$\frac{\Gamma \vdash \Delta, A}{A^{\perp}, \Gamma \vdash \Delta}\ \perp_L \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, A^{\perp}}\ \perp_R$$

$$\frac{}{\Gamma \vdash \Delta, \top}\ \top_R \qquad \frac{}{\mathbf{0}, \Gamma \vdash \Delta}\ 0_L$$

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

$$\frac{A, \Gamma \vdash \Delta}{A \& B, \Gamma \vdash \Delta}\ \&_L \qquad \frac{B, \Gamma \vdash \Delta}{A \& B, \Gamma \vdash \Delta}\ \&_L \qquad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta B}{\Gamma \vdash \Delta, A \& B}\ \&_R$$

$$\frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \oplus B, \Gamma \vdash \Delta}\ \oplus_L \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \oplus B}\ \oplus_R \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \oplus B}\ \oplus_R$$

$$\frac{A, \Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}\ !_L \qquad \frac{!\Gamma \vdash A, ?\Delta}{!\Gamma \vdash !A, ?\Delta}\ !_R \qquad \frac{!\Gamma, A \vdash ?\Delta}{!\Gamma, ?A \vdash ?\Delta}\ ?_L \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, ?A}\ ?_R$$

$$\frac{A[t/x], \Gamma \vdash \Delta}{\forall x A, \Gamma \vdash \Delta}\ \forall_L \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, \forall x A}\ \forall_R \qquad \frac{A, \Gamma \vdash \Delta}{\exists x A, \Gamma \vdash \Delta}\ \exists_L \qquad \frac{\Gamma \vdash \Delta, A[t/x]}{\Gamma \vdash \Delta, \exists x A}\ \exists_R$$

Appendix B. Permutability for the linear connectives

The following table illustrates the possible permutations of the rule in the fragment considered in Hodas and Miller (1994).

$t_2 \backslash t_1$	$\multimap_L$	$\otimes_R$	$\multimap_R$	$!_R$	$\&_R$	$\&_L$	$\oplus_R$	$!_L$	$w!_L$	$c!_L$	$\forall_R$	$\forall_L$	$\exists_R$
$\multimap_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$\otimes_R$	p	np	np	np	np	p	np	p	p	p	np	p	np
$\multimap_R$	np	np	np	np	np	p	np	p	p	p	np	p	np
$!_R$	np	np	np	np	np	np	np	np	p	p	np	np	np
$\&_R$	np	np	np	np	np	np	np	np	np	np	np	np	np
$\&_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$\oplus_R$	p	np	np	np	np	p	np	p	p	p	np	p	np
$!_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$w!_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$c!_L$	np	np	p	p	p	p	p	p	p	p	p	p	p
$\forall_R$	p	np	np	np	np	p	np	p	p	p	np	np	np
$\forall_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$\exists_R$	p	np	np	np	np	p	np	p	p	p	np	p	np

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

Acknowledgments

The authors would like to thank the anonymous referees for their fruitful comments and recommendations.

References

Abadi, M. and Cardelli, L. (1996) *A Theory of Objects*, Monographs in Computer Science, Springer-Verlag.

Abramsky, S. (1993) Computational Interpretations of Linear Logic. *Theoretical Computer Science* **111** 3–57.

Agha, G. (1986) Actor: A Model of Concurrent Computation in Distributed Systems, The MIT Press.

Aït-Kaci, H. and Nasr, R. (1986) LOGIN: A Logic Programming Language with Built-in Inheritance. *Journal of Logic Programming* **3** (3) 185–215.

Alexiev, V. (1994) Applications of Linear Logic to Computation: An Overview. *Bulletin of the IGLP* **2** (1) 77–107.

Andreoli, J. M. (1992) Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* **2** (3) 297–347.

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

Boudinet, E. and Galmiche, D. (1996) Proofs, Concurrent Objects and Computations in a FILL Framework. Proceedings of the Workshop on Object-based Parallel and Distributed Computation, OBPDC'95. *Springer-Verlag Lecture Notes in Computer Science* **1107** 148–167.

Brogi, A., Lamma, E. and Mello, P. (1991) Objects in a Logic Programming Framework. In: Voronkov, A. (ed.) Proceedings of the 1st Russian Conference on Logic Programming. *Springer-Verlag Lecture Notes in Computer Science* **592** 102–113.

Bugliesi, M., Delzanno, G., Liquori, L. and Martelli, M. (1996) A Linear Logic Calculus of Objects. In: *Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn*, The MIT Press 67–81.

Chen, W. and Warren, D. H. (1988) Objects as Intensions. In: Kowalski, R. A. and Bowen, K. A. (eds.) *Proceedings of 5th International Conference on Logic Programming*, The MIT Press 404–419.

Chirimar, J. (1995) *Proof Theoretic Approach to Specification Languages*, Ph. D. thesis, Department of Computer and Information Science, University of Pennsylvania.

Clark, K. L. and Gregory, S. (1986) PARLOG: Parallel Programming in Logic. *ACM TOPLAS* **8** (1) 1–49.

Conery, J. S. (1988) Logical Objects. In: Kowalski, R. A. and Bowen, K. A. (eds.) *Proceedings of 5th International Conference on Logic Programming*, The MIT Press 420–434.

Delzanno, G. (1997) *Logic & Object-Oriented Programming in Linear Logic*, Ph. D. thesis, Università of Pisa, Dipartimento di Informatica.

Delzanno, G. and Martelli, M. (1995) Objects in Forum. In: *Proceedings of the International Logic Programming Symposium*, The MIT Press 115–129.

Delzanno, G. and Martelli, M. (1996) Objects in a Higher Order Linear Logic Setting. In: *Proceedings of the Proof Theory and Concurrent Object-Oriented Programming pre-ECOOP 96 Workshop*, Linz, Austria, July 1996. (Also in *Special-Issue in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP 96*, D-Punkt.)

Delzanno, G. and Martelli, M. (1998) Proofs as Computations in Linear Logic. Technical Report DISI-TR-98-12, DISI, Dipartimento di Informatica, Università di Genova.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

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

Galmiche, D. (1994) Canonical Proofs for Linear Logic Programming Frameworks. In: *Workshop on Proof-theoretical Extensions of Logic Programming*, Santa Margherita Ligure, Italy.

Galmiche, D. and Perrier, G. (1994a) Foundations of Proof Search Strategies Design in Linear Logic. In: Logic at St Petersburg '94, Symposium on Logical Foundations of Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **813** 101–113.

Galmiche, D. and Perrier, G. (1994b) On Proof Normalization in Linear Logic. *Theoretical Computer Science* **135** (1) 67–110.

Girard, J. Y. (1987) Linear logic. *Theoretical Computer Science* **50** 1–102.

Guglielmi, A. (1995) *Abstract Logic Programming in Linear Logic - Independence and Causality in a First Order Calculus*, Ph. D. thesis, Department of Computer Science, University of Pisa.

Harland, J. A., Pym, D. and Winikoff, M. (1996) Programming in Lygon: An Overview. In: Wirsing, M. and Nivat, M. (eds.) Algebraic Methodology and Software Technology. *Springer-Verlag Lecture Notes in Computer Science* **1101** 391–405.

Hoare, C. A. R. (1978) Communicating Sequential Processes. *CACM* **21** (8) 666–677.

Hodas, J. (1994) *Logic Programming in Intuitionistic Linear Logic*, Ph. D. thesis, University of Pennsylvania, Department of Computer and Informaton Science.

Hodas, J. and Miller, D. (1989) Representing Objects in a Logic Programming Language with Scoping Constructs. In: Warren, D. H. and Szeredi, P. (eds.) *Proceedings of 7th International Conference on Logic Programming*, The MIT Press 511–526.

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

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

Hyland, M. and de Paiva, V. (1993) Full Intuitionistic Linear Logic (extended abstract). *Annals of Pure and Applied Logic* **64** 273–291.

Kahn, K., Tribble, E., Miller, M. and Bobrow, D. (1987) Vulcan: Logical concurrent objects. In: Wegner, P. and Shriver, B. (eds.) *Research Directions in Object-Oriented Programming*, The MIT Press.

Kobayashi, N. and Yonezawa, A. (1994a) Asynchronous Communication Model based on Linear Logic. *Formal Aspects of Computing* **3** 279–294.

Kobayashi, N. and Yonezawa, A. (1994b) Higher-order Concurrent Linear Logic Programming. In: Proceedings of Theory and Practice of Parallel Programming (TPPP'94). *Springer-Verlag Lecture Notes in Computer Science*.

Kobayashi, N. and Yonezawa, A. (1994b) Type-theoretic Foundations for Concurrent Object-Oriented Programming. In: *Proceedings of ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '94)* 31–45.

Lincoln, P. and Saraswat, V. (1993) Higher-order, Linear, Concurrent Constraint Programming (Unpublished manuscript, available at `parcftp.xerox.com/pub/ccp/lcc/hlcc.dvi` Xerox Co.)

Lincoln, P. and Shankar, N. (1994) Proof Search in First-order Linear Logic and Other Cut-free Sequent Calculi. In: *9th IEEE Symposium on Logic in Computer Science*, Paris, France 282–291.

McCabe, F. G. (1992) *Logic and Objects*, International Series in Computer Science, Prentice Hall.

McDowell, R. and Miller, D. (1997) A Logic for Reasoning with Higher-order Abstract Syntax. In: *Proceedings of LICS'97*, Warsaw 434–445.

Meseguer, J. (1990) A Logical Theory of Concurrent Objects. In: OOPSLA-ECOOP '90, Ottawa. *Sigplan Notices* **25** (10) 101–115.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press

Meseguer, J. and Marti-Oliet, N. (1991) From Petri nets to Linear Logic. *Math. Struct. in Comp. Science* **1** 69–101.

Miller, D. (1989a) A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming* **6** 79–108.

Miller, D. (1989b) Lexical Scoping as Universal Quantification. In: *Proceedings of 6th International Conference on Logic Programming*, The MIT Press 268–283.

Miller, D. (1993) The π-calculus as a Theory in Linear Logic: Preliminary Results. In: Lamma, E. and Mello, P. (eds.) Proceedings of the 1992 Workshop on Extension to Logic Programming. *Springer-Verlag Lecture Notes in Computer Science* **660** 242–265.

Miller, D. (1994) A Multiple-conclusion Meta-logic. In: *9th IEEE Symposium on Logic in Computer Science*, Paris, France 272–281.

Miller, D. (1996) Forum: A Multiple-conclusion Specification Logic. *Theoretical Computer Science* **165** (1) 201–232.

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

Nadathur, G. and Miller, D. (1988) An Overview of λ-Prolog. *Fifth International Symposium on Logic Programming*.

Perrier, G. (1995) A Model of Concurrency Based on Linear Logic. In: *Proceedings of the Conference on Computer Science Logic 95*, Paderborn, Germany.

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

Schellinx, H. (1991) Some Syntactical Observations on Linear Logic. *Journal of Logic and Computation* **1** (4) 537–559.

Shapiro, E. Y. (1989) The Family of Concurrent Logic Programming Languages. *Computing Surveys* **21** (3) 413–510.

Snyder, A. (1986) Encapsulation and Inheritance in Object-oriented Programming languages. In: *Proceedings of OOPSLA '86* 38–44.

Tammet, T. (1994) Proof Strategies in Linear Logic. *Journal of Automated Reasoning* **12** 273–304.

Wallen, L. A. (1990) *Automated Proof Search in Non-Classical Logics*, The MIT Press.

Wegner, P. (1987) Dimension of Object-based Language Design. In: *Proceedings of OOPSLA '87* 168–182.

https://doi.org/10.1017/S0960129599002789 Published online by Cambridge University Press