Theoretical Computer Science 258 (2001) 269–297

Theoretical Computer Science

www.elsevier.com/locate/tcs

Proofs as computations in linear logic

Giorgio Delzanno[a,*], Maurizio Martelli[b]

[a]*c/o DISI-Università di Genova, Via Dodecaneso 35, I-16146 Genova, Italy*

[b]*Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Via Dodecaneso 35, I-16146 Genova, Italy*

Received May 1997
Communicated by G. Levi

Abstract

The notions of *uniform proof* and of *resolution* represent the foundations of the proof-theoretic characterization of logic programming. The class of Abstract Logic Programming Languages nicely captures these concepts for a wide spectrum of logical systems. In the logic programming setting, however, the structure of the formulas, e.g. Horn clauses and hereditary Harrop formulas, plays a crucial role in discriminating between programming and theorem proving. In the paper, and in the framework of the proofs as computations interpretation of linear logic, we present an extension of hereditary Harrop formulas and a corresponding logical system which are the foundations of the logic programming language . The starting point of this study is Forum (Miller, Theoret. Comput. Sci. 165 (1) (1996) 201–232), a presentation of higher-order linear logic in terms of uniform proofs. A subset of its formulas have been isolated and proved to be well-suited to encode descriptions of various programming paradigms. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Extensions of logic programming; Linear logic; Higher-order logic

1. Introduction

The notion of Abstract Logic Programming Language (ALPL) [26] gives the guidelines for designing well-founded logic programming languages. In an Abstract Logic Programming Language formulas are partitioned into *clauses* and *goals* and an abstract interpreter is given in terms of a goal-driven proof system, i.e. where the search for a proof is guided by the structure of the goal to be proved. Moreover, resolution steps (backchaining) are used to simplify atomic goal formulas using clauses

* Corresponding author.
E-mail addresses: giorgio@disi.unige.it (G. Delzanno), martelli@disi.unige.it (M. Martelli).

0304-3975/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0304-3975(00)00024-4

nondeterministically selected from the program. Besides the logic of Horn Clauses [30] and of hereditary Harrop formulas [27], several fragments of Girard's linear logic [15] have been isolated and proved to be ALPLs. Examples are *LO* [2], Lolli [18], and Forum [25]. Furthermore, Lygon [17] has been proved to be the largest fragment of linear logic enjoying uniform provability, in the sense stated in [29]. All these languages adhere to the so called *proofs as computations* interpretation of linear logic, an interpretation in which sequents represent instant configurations of a computation and where the logical rules describe its evolution. The result of a computation is the computed answer substitution associated to the proof.

These aspects have been extensively studied in Forum [7, 10, 25], a powerful specification language complete with respect to higher-order linear logic. Many aspects of Forum, however, rely on full linear logic and do not have an immediate operational reading according to 'traditional' logic programming languages. For instance, Forum allows negative occurrences of formulas like $A \multimap \bot$ (or $A^{\bot}$); they introduce a further level of nondeterminism in the proofs as they can introduce new goals any time during a proof. Though these features can be useful to specify, for instance, reactive systems (to simulate occurrences of external events) it can be difficult to accommodate them into an executable logic programming language cf. [19, 22]. Is there a weaker setting (or a subset of Forum) which is still capable of representing most common computational mechanisms but which is closer to a programming paradigm? To answer this question it seems important to study the structure of formulas, trying to isolate a subset which naturally extends the operational aspects of existing logic programming languages. We are looking for a logic based on a distinction between clauses and goals, and such that the corresponding proof system assigns a clear operational meaning to each of its connectives (i.e. connectives as programming constructs). The proof system should comprise only one single left rule acting as a backchaining rule. These requirements represent further restrictions to the notion of ALPL, however, they are still reasonable when trying to design a programming paradigm.

In the paper, we propose a paradigm integrating the logic of hereditary Harrop formulas [26] with the process view of sequents and proofs introduced by Kobayashi and Yonezawa in [20] and revised by Miller in [23]. In our view of proofs as computation we aim at representing the evolution of the state of a 'system' so as to observe its possible 'final states'. In this setting multi-conclusion clauses [1] play a central role for both synchronization and resource management. The resulting fragment of Forum, called *ehhf* (for *extended hereditary Harrop formulas*) [8], enjoys a clear methodology to write and execute specifications based on linear logic. In the logic *ehhf* it is possible to combine the power of higher-order logic programming languages like λProlog [27] with the view of proofs as computations via rewriting steps suggested in [25]. Taking advantage of previously developed results in proof theory of linear logic [1, 14, 25, 29], we will focus on the problems related to provability in a higher-order setting. Specifically, in the paper we will prove the completeness of provability in *ehhf* with respect to provability in Forum. This result is inspired by the result of completeness for hereditary Harrop formulas with respect to provability in higher-order intuitionistic logic [26]. The logic of *ehhf* turns out to be a powerful prototyping language with an operational reading of the resulting programs which is not far from the familiar logic programming languages.

In the paper, we will also present a different higher-order formulation of the considered fragment of Forum where we allow variables to range over $\mathscr{D}$-formulas, a special class of Forum formulas we will use to denote logic programs, hence the name *ehhf*$_D$. The two logics, *ehhf* and *ehhf*$_D$, coincide in their first-order formulation, whereas they differ in the way quantification is introduced in their higher-order formulation. The logic of *ehhf*$_D$ is interesting from a practical point of view, since it provides examples of interchangeability between meta- and object-level suitable to encode call-patterns, dynamic modification and mobility of code [10, 5, 3]. In the paper, we will prove a completeness result for provability in *ehhf*$_D$ with respect to provability in Forum. In the following we will assume the reader to be familiar with linear logic and its presentation (Forum) given by Miller in [24, 25].

1.1. Related work

Andreoli's focusing proofs [1] set the foundations for studying extensions of logic programming based on linear logic. The language *LO* proposed by Andreoli and Pareschi in [2] and its superset LinLog [1] introduced the use of multi-conclusion clauses to manage collections of resources while viewing sequents as description of processes and proofs as execution of concurrent processes.

Kobayashi and Yonezawa [20] and Miller [23] proposed an alternative model in which sequents modeled collections of processes instead of a single process. In this setting the connectives were considered as process constructors and, in particular, the multiplicative disjunction $\invamp$ as a primitive for parallelism. Lincoln and Saraswat [21] and Perrier's CPL [28] discussed a dual representation (i.e. $\otimes$ as primitive for parallelism) in which linear implication acts as an *ask* primitive. A similar approach has been considered in FILL [13] in which two-sided sequents are considered in the context of a multiple-conclusion formulation of Intuitionistic linear logic. By the duality of linear logic connectives, many aspects of the previously mentioned works can be rephrased in presentations of linear logic closer to a logic programming perspective, as discussed below.

Hodas and Miller in [18] and Pym and Harland in [29] focused on uniform-proof formulations of linear logic in which embedding Horn clauses and hereditary Harrop formulas so as to extend standard logic programming with resource management. An intelligent management of resources has been studied in order to define an implementation of the language (the I/O model in [18]). Lolli does not provide what Andreoli and Pareschi [2] called *local* parallelism, i.e. the connective $\invamp$ cannot occur in a goal. On the other hand multi-conclusion sequent-calculi, as shown in [13, 16, 20, 23] allow us to operationally model concurrent agents by exploiting multi-set rewriting. None of the works on uniform fragments of linear logic have considered extensions to the higher-order case.

Forum [25] can be considered as the natural extension of Lolli to a multi-conclusion sequent calculus. A Forum sequent has the form $\Sigma : \Gamma; \Delta \longrightarrow \Omega; \Upsilon$ where Γ and Υ are sets of unbounded-use formulas (Γ contains !-formulas and Υ contains ?-formulas). Forum (cut-free) proofs are uniform in the following sense: the right-hand side of sequents must be reduced to a multi-set of atomic formulas before applying left introduction rules. In [7, 25], several application of Forum to the specification of resource sensible logics and programming paradigms have been described.

Implementations of a first-order formulation of Forum based on the I/O model of [18] have been recently proposed by Hodas and Polakow in [19] and by López and Pimentel in [22]. In [19], the authors also point out the need of restricting the set of formulas in order to turn Forum into an executable logic programming language. Our proposal, that extends our preliminary formulations of *ehhf* presented in [10], is a preliminary step in this direction.

1.2. Structure of the paper

The language *ehhf* is formally introduced in Section 2. In the same section we briefly discuss the 'operational' view we have in mind, and the relationship of *ehhf* with hereditary Harrop formulas. In Section 3, we discuss the technical details of the completeness proof of *ehhf* w.r.t. Forum. In Section 4, we present an alternative higher-order extension of the first-order version of *ehhf*, called $ehhf_D$, which differs from the usual treatment of higher-order features of logic programming (e.g. [26]) in both the considered language and the technical details of the completeness proof. In Section 5, we suggest a methodology to write *ehhf*-executable specifications. Finally, in the remaining part of the paper we present two examples in which we employ the two higher-order extensions discussed in the paper. In Section 6, we specify the operational semantics of an imperative language with functions using the core language *ehhf* in a continuation passing style. In Section 7, we show how $ehhf_D$ can be used to describe complex systems in which it is necessary to predicate over *programs*, *objects* or *agents*. Finally, Section 8 includes some final remarks and future perspectives of the work.

2. Proofs as computations via rewriting

Before formally introducing the logic of *ehhf* we will try to give some hints on the operational interpretation of its formulas. As shown in [2, 20, 16], provability in linear logic shares many aspects with rewriting. In the following example, we will use Forum syntax [25], with the following exception: here and in the rest of the paper we will use $\circ\!-$ for negative occurrences of $\multimap$ (e.g. at the top level in a clause). A set of rewrite rules of the form s_i *rewrites in* t_i can be encoded as a linear logic theory, say Γ, consisting of formulas of the form $s_i \circ\!- t_i$.[1] The query 'does s rewrite in t?' can be

[1] For the sake of simplicity, we consider only ground terms, and we do not consider *context* rules. See [9] for a detailed encoding of term rewriting systems in linear logic.

solved by checking provability for the Forum sequent $\Gamma;\ \longrightarrow t \multimap s$: if the sequent is provable in Forum (hence provable with a uniform proof [2, 18, 25]) then s rewrites in some steps in t, and, vice versa, if s rewrites in some steps in t then the sequent above is provable in Forum. For instance, let $\Gamma = \{s \circ\!\!- r, r \circ\!\!- t\}$. Consider now the following proof for *s rewrites in t* (built using Forum rules):

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; s \longrightarrow s; \cdot}\,\mathit{initial} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma; r \longrightarrow r; \cdot}\,\mathit{initial} \quad \dfrac{}{\Gamma; \mathbf{t} \longrightarrow t; \cdot}\,\mathit{initial}}{\Gamma; \mathbf{t} \xrightarrow{r \circ\!\!- t} r; \cdot}\,\multimap_l}{\Gamma; \mathbf{t} \longrightarrow r; \cdot}\,\mathit{decide}!}{\Gamma; \mathbf{t} \xrightarrow{s \circ\!\!- r} s; \cdot}\,\multimap_l}{\Gamma; \mathbf{t} \longrightarrow s; \cdot}\,\mathit{decide}!}{\Gamma; \cdot \longrightarrow \mathbf{t} \multimap s; \cdot}\,\multimap_r
$$

Note that an application of rule $\multimap_l$ corresponds to a *backchaining step* [25, 19], in which the head of the clause $s \circ\!\!- t$ matches the current goal s. This correspondence can be extended to conditional rewriting rules, i.e. rules of the form *if c then s rewrites in t*. They find a natural counterpart in the following type of formulas: $c \Rightarrow (s \circ\!\!- t)$. In fact, when applied in a backchaining step, they behave as follows:

$$
\dfrac{\dfrac{\cdots}{\Gamma; \cdot \longrightarrow c; \cdot} \qquad \dfrac{\dfrac{}{\Gamma; s \longrightarrow s; \cdot}\,\mathit{initial} \quad \dfrac{\cdots}{\Gamma; \mathbf{r} \longrightarrow t; \cdot}}{\Gamma; \mathbf{r} \xrightarrow{s \circ\!\!- t} s; \cdot}\,\multimap_l}{\Gamma; \mathbf{r} \xrightarrow{c \Rightarrow (s \circ\!\!- t)} s; \cdot}\,\Rightarrow_l
$$

Note that the condition c must be proved in a sequent with empty bounded context. In the previous examples each backchaining step models the *evolution* of the configuration represented by a sequent. Note also that the root sequents have a *residue*, namely **r** and **t** in the previous figures, occurring in the bounded context of the sequent. The residue is used in the final step of this process to *match* the final configuration.

The view of proofs as conditional rewriting introduced in the previous example can be formalized and successively generalized to more interesting examples by considering the following class of Forum formulas:

$$\forall x_1 \ldots . \forall x_1 .\ (G_1 \,\&\, \ldots \,\&\, G_n \Rightarrow (H \circ\!\!- G)),$$

in which the goals $G_1, \ldots, G_n$ represent the conditional part of the rewrite rule *H rewrites in G*. Specializing the Forum proof-system to this class of formulas, we obtain a proof system with a single backchaining rule that describes the operational meaning of the considered type of clauses. As mentioned before we will consider multiple-headed clauses, hence allowing H to be a disjunction of atomic formulas. In order to provide constructs to control the rewriting process, in addition to $\multimap$ and $\Rightarrow$ we allow & and $\forall$ to occur in goal formulas. We leave extensions of the fragment with other connectives (e.g. introducing $\otimes$ in the goals as in [18, 17]) for future works (see also discussion in Section 8).

The appealing aspects of the resulting formalism is that it provides a clear way to study integration between rewrite-based specification languages and logic programming languages. In fact, we can naturally integrate logic programming aspects allowing the conditional part of the clauses to be an invocation of predicates defined by linear logic programs (in the sense introduced for instance in [17, 18]). We will turn back to this point after a more precise introduction of the language and a discussion of its properties. In the following section, we will introduce the fragment of Forum we are interested in, namely *ehhf*. We anticipate here that, in Section 4, we will introduce another fragment, namely $ehhf_D$, which differs from *ehhf* only in its higher-order formulation. Since the first order formulations of the two fragments coincide, our informal introduction of the operational behavior of the formulas we are interested in applies to *ehhf* as well as to $ehhf_D$.

2.1. Extended hereditary Harrop formulas

As discussed in the previous section, we aim at finding a general characterization of the considered conditional-rules by considering a subset of Forum formulas. In the rest of the paper, a signature Σ will denote a list of elements of the form $c:\tau$ where c is a constant symbol and τ its type. Σ-terms are applications and lambda-abstractions built over the symbols in Σ according to their types.[2] Each connective is defined by a nonlogical constant having as target type o. The linear connectives we will consider are $⅋$, $\&$, $\circ\!-$ ($-\!\circ$), having type $o \to o \to o$; a family of indexed symbols $\forall_\tau$ for each type τ having type $(\tau \to o) \to o$ (the type is omitted when it is clear from the context), and the logical constants $\top$ and $\bot$ with type o. The class of formulas we are interested in is given by the following grammar:

$$\mathscr{D} ::= \mathscr{D} \,\&\, \mathscr{D} \mid \forall x.\mathscr{D} \mid H \circ\!- \mathscr{G} \mid \mathscr{G} \Rightarrow \mathscr{D} \mid H.$$

$$\mathscr{G} ::= \mathscr{G} \,\&\, \mathscr{G} \mid \mathscr{G} ⅋ \mathscr{G} \mid \forall x.\mathscr{G} \mid \mathscr{D} \Rightarrow \mathscr{G} \mid \mathscr{D} -\!\circ \mathscr{G} \mid A_\mathrm{r} \mid \bot \mid \top .$$

$$H ::= H ⅋ H \mid A_\mathrm{r}.$$

Here A_r is any *rigid* atomic formula with type o, i.e. $A_\mathrm{r} = p(t_1, \ldots, t_n)$ where t_i has type τ_i for $i : 1, \ldots, n$ and p is a constant symbol having type $\tau_1 \to \cdots \to \tau_n \to o$. Note that the formula $G_1 \Rightarrow \cdots \Rightarrow G_n \Rightarrow (H \circ\!- G)$ is equivalent to the formula $G_1 \,\&\, \cdots \,\&\, G_n \Rightarrow (H \circ\!- G)$, thus $\mathscr{D}$-formulas provide the desired class of clauses. First order formulas are defined as formulas built from non-logical symbols of type $\tau_1 \to \cdots \to \tau_n \to \tau$, such that τ_i for $i : 1, \ldots, n$ is a primitive type distinct from o and τ is a primitive type, and such that universal quantification is restricted to primitive types distinct from o. The language *ehhf* standing for extended hereditary Harrop formulas, results by considering the set of formulas generated by the productions $\mathscr{D}$ and $\mathscr{G}$, i.e. $\mathscr{D}$-clauses (formulas) and $\mathscr{G}$-formulas (goals), respectively. In the rest of the paper, the term *formula* will

[2] Given a set of primitive types containing o, functional types are built using a constructor $\to$ (i.e. $\tau \to \sigma$ if τ and σ are types). Furthermore, we will consider signatures without empty types.

Axioms and Structural rules

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

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

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

Search Rules

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

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

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

Backchaining

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

$$G \Rightarrow (A_1 \mathbin{⅋} \ldots \mathbin{⅋} A_n \circ\!- B) \in \langle D \rangle, \{A_1,\ldots,A_n\} \equiv \mathscr{A}.$$

Fig. 1. The *ehhf* proof system: in ($\forall_r$), $y:\tau \in \Sigma$ and $\Sigma' = y:\tau, \Sigma$.

denote an *open* formula, i.e. with possible occurrences of free variables. Furthermore, given a (open) formula F, $\forall F$ will denote the closed formula $\forall x_1 \cdots \forall x_n.F$ obtained by adding a universal quantifier $\forall x_i$ for each free variable x_i in F.

According to the Forum syntax [25], the left-hand side of a sequent can be partitioned into two parts: the set of unbounded-use clauses, say Γ, and the multi-set of bounded-use clauses, say Δ. We will use $\cdot$ to denote the empty multi-set, and $\uplus$ to denote multi-set union. With 'unbounded' we mean that all formulas in Γ are prefixed by the exponential !, i.e. a copy of them is always available during a derivation. In this paper, we simplify the presentation of Forum sequents as follows: we write a Forum sequent $\Sigma : \Gamma; \Delta \longrightarrow \mathscr{A}, F, \Omega; \cdot$ (i.e. with empty ?-context) as $\Gamma; \Delta \longrightarrow_{\Sigma} \Omega'$ where Ω' is the multi-set $\{F\} \uplus \mathscr{A} \uplus \Omega$, i.e. without loss of generality, in the right-hand side of sequents we consider multi-sets of formulas instead of lists (see [24, 25]). *ehhf*-sequents have the following form: $\Gamma; \Delta \longrightarrow_{\Sigma} \Omega$ or $\Gamma; \Delta \xrightarrow{D}_{\Sigma} \mathscr{A}$, where Γ is a set of $\mathscr{D}$-formulas, Δ is a multi-set of $\mathscr{D}$-formulas, Ω is a multi-set of $\mathscr{G}$-formulas, D is a $\mathscr{D}$-formula and $\mathscr{A}$ a multi-set of atomic formulas. All the formulas are defined over the signature Σ. The set of Forum rules can be specialized to this class of formulas as shown in Fig. 1. In the higher-order case all the rules are modulo λ-conversion (in [26] a special λ rule is part of the system), so as to consider terms in normal form only. In the rest of the paper we will use $t = t'$ to denote that t λ-converts to t'. Note that the left rules collapse into one single backchaining rule [25, 29, 19], namely *bc*. The axiom *initial* is an extension of the *initial* axiom of Forum to (universally quantified) clauses consisting of linear disjunctions of atomic formulas.[3] It can also be seen as an extension of the base case of resolution for classical logic (i.e. resolution with a unit clause). In these rules $\langle D \rangle$ represents the set of instances of a clause D. The set $\langle D \rangle$ is defined as follows. Given a multi-set Δ of $\mathscr{D}$-clauses over a signature Σ, $\langle \Delta \rangle$ is the minimum set of 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 $\langle A \rangle = \{A\}$, otherwise. The considered class of formulas is an extension of hereditary Harrop formulas [26], which are defined as follows:

$$\mathscr{D}_i \ ::= \ true \mid A \mid \mathscr{D}_i \wedge \mathscr{D}_i \mid G_i \supset D_i \mid \forall x.\mathscr{D}_i$$

$$\mathscr{G}_i \ ::= \ true \mid A \mid \mathscr{G}_i \wedge \mathscr{G}_i \mid \mathscr{G}_i \vee \mathscr{G}_i \mid \mathscr{D}_i \supset \mathscr{G}_i \mid \forall x.\mathscr{G}_i \mid \exists x.\mathscr{G}_i.$$

The standard embedding of intuitionistic logic into linear logic given by Girard [15] is defined as follows: $true^0 = \mathbf{1}$, $A^0 = A$ for A atomic, $(\mathscr{D}_1 \wedge \mathscr{D}_2)^0 = \mathscr{D}_1^0 \,\&\, \mathscr{D}_2^0$, $(G \supset D)^0 = \,!(G^0) \multimap D^0 = (G^0) \Rightarrow D^0$, $(\forall x.\mathscr{D})^0 = \forall x.\mathscr{D}^0$, $(\exists x.\mathscr{D})^0 = \exists x.\mathscr{D}^0$, $(\mathscr{D}_1 \vee \mathscr{D}_2)^0 = \mathscr{D}_1^0 \oplus \mathscr{D}_2^0$. Furthermore, let Δ^0 be $\{D^0 \mid D \in \Delta\}$. If we disregard $\vee$ and $\exists$ and we modify the Girard's encoding so as to map *true* into $\top$, then the logic of hereditary Harrop formulas can be embedded into the considered language as stated in the following proposition, where, according to [26], we use $\vdash_I$ to denote sequents of intuitionistic logic.

Proposition 1. *Let Γ be a set of D_i-formulas and G a G_i-formula, both Γ and G without occurrences of $\vee$ and $\exists$, then $\Gamma \vdash_I G$ is provable in intuitionistic minimal logic if and only if Γ^0; $\longrightarrow_\Sigma G^0$ is provable in ehhf.*

Proof. The proof is similar to the one given in [18]. $\square$

Note that Horn formulas are a fragment of hereditary Harrop formulas. Their encoding into *ehhf* gives, e.g., clauses of the form $\mathscr{G}_\mathrm{h} \Rightarrow A$ where $\mathscr{G}_\mathrm{h}$ is either a conjunction of atomic formulas or the constant $\top$. As stated in the previous proposition, the rule *bc* provides a natural way to combine logic programming and the proofs as rewriting interpretation of linear clauses we sketched at the beginning of this section. In fact, the conditional part of a $\mathscr{D}$-clause (to be proved in the empty context) can be considered as a goal invocation in a traditional logic programming language, e.g. Prolog (based on Horn Clauses) or λProlog (based on hereditary Harrop formulas). In this sense, they can be viewed as guards for the application of the linear clauses. We will turn back to this point in Section 5. It is also important to point out that the encoding of hereditary Harrop formulas given in [29, 18] are based on $\otimes$ and thus they could

[3] The rule *initial* can be decomposed into a sequence of applications of $\forall_l$ and $\mathbin{⅋}_l$ terminated by instances of Forum *initial* axioms.

provide a finer grained manipulation of resources. Our purpose, however, is to handle resources in the main thread of the rewriting and, specifically, in the right-hand side of sequents. Multi-headed clauses will be used to achieve this aim.

In the following section, we will discuss the soundness and completeness of the considered proof system (remember that we consider a higher-order formulation) w.r.t. Forum. In the rest of the paper, $\vdash_F$ will denote provability in Forum, $\vdash_L$ will denote provability in full linear logic, and $\vdash_e$ will denote provability in *ehhf*, i.e. the system of Fig. 1. Furthermore, $\delta \vdash_x s$ will denote the existence of a proof δ for the sequent s in the proof-system x (one of *ehhf*, Forum, full linear logic).

3. Completeness with respect to Forum

In the first-order case the correspondence between Forum provability and *ehhf* provability follows by observing that the proof-system in Fig. 1 results by specializing the set of Forum rules to the class of *ehhf*-formulas. However, in order to capture interesting computational models, such as objects, processes, and agents, in Section 2 we have considered a language based on the Simply Typed λ-calculus. In this setting it is possible to express formulas of any order, and to quantify over variables having target type o. As a consequence the Sub-Formula Property [12] (which holds in cut-free first-order sequent calculus for classical, intuitionistic and linear logic) does not hold anymore. This can affect the completeness of *ehhf*-provability w.r.t. Forum. More precisely, when trying to prove an *ehhf*-sequent in Forum, the application of an instance of the rule $\forall_l$ may introduce formulas that are not in *ehhf*. For instance, consider the following Forum proof of an initial sequent consisting of *ehhf*-formulas:

$$\dfrac{\dfrac{\dfrac{\overset{\delta}{\Gamma;\ \xrightarrow{\;p\;} p} \qquad \overset{\gamma}{\Gamma;\cdot \longrightarrow (\psi \multimap \phi)}}{\Gamma;\cdot \xrightarrow{\;p \multimapinv (\psi \multimap \phi)\;} p}\ {\multimap_l}}{\Gamma; \forall_o x.p \multimapinv x \longrightarrow p}\ \mathit{decide}}{\Gamma; \forall_o x.p \multimapinv x \longrightarrow p}\ \forall_l$$

Here the formulas ϕ and ψ can be generic Forum formulas (outside *ehhf* as well) and γ an arbitrary Forum proof for the sequent $\Gamma; \psi \longrightarrow \phi$. As for higher-order hereditary Harrop formulas [26], it is possible to find a proof in *ehhf* by instantiating the variable x with $\top$ instead of $\psi \multimap \phi$, i.e. a proof where all terms introduced by $\forall_l$ are *ehhf*-terms. Actually, it is not necessary to replace each term of type o introduced by a $\forall_l$ over a variable x, with $\top$. In fact, it is enough to *transform* the sub-terms with implication at the top level into $\top$, as we will show in the following. For this purpose we need to define a restricted universe of terms.

Definition 2 (*Restricted universe of terms*). Let $\mathscr{T}$ be the set of terms built over a given signature Σ, and the connectives &, $\invamp$, $\forall$, $\top$, and $\bot$ (i.e. terms of $\mathscr{T}$ have no implications).

In the rest of the section, *formulas over* $\mathscr{T}$ will denote formulas with nested subterms defined over $\mathscr{T}$. Since (at the top-level within a formula) variables of type o can only occur in goal position, restricting the set of terms to $\mathscr{T}$ avoids the introduction of formulas which are not *ehhf*-terms. The following properties hold.

Proposition 3. *If* $s, t \in \mathscr{T}$ *then* $t[s/x] \in \mathscr{T}$. *If* $\forall x.F$ *is a* $\mathscr{G}(\mathscr{D})$*-formula then* $F[t/x]$ *is still a* $\mathscr{G}(\mathscr{D})$*-formula.*

The proofs are by induction on the structure of terms and formulas.

Let $\mathscr{T}_F$ be the set of all possible Forum terms over Σ. In order to transform a Forum proof of an *ehhf*-sequent into an *ehhf*-proof we will define a normalization from $\mathscr{T}_F$ to $\mathscr{T}$ of the terms occurring in the original proof.

Definition 4 (*Restriction map*). The map $\cdot^+ : \mathscr{T}_F \rightsquigarrow \mathscr{T}$ is defined by induction on terms in normal form as follows: $x^+ = x$ if x is a variable; $c^+ = c$ if c is a constant different from $\multimap$ and $\Rightarrow$; $\multimap^+$ and $\Rightarrow^+$ are equal to $\lambda x.\lambda y.\top$; $(\lambda x.t)^+ = \lambda x.t^+$; and $(st)^+ = (s^+ t^+)$.

In the rest of the section, $\overline{[s/x]}$ will denote a substitution consisting of the sequence of pairs $[s_1/x_1]\ldots[s_n/x_n]$, where the s_i's are *closed* terms, and the variables x_i's are all different from each other (i.e. in $t\overline{[s/x]}$ substitutions cannot be composed but just applied to t). Under this hypothesis, the following proposition is true.

Proposition 5. *Let* t *be a term* (*in normal form*) *with free variables* $x_1 \ldots x_n$, *then* $(t[s_1/x_1]\ldots[s_n/x_n])^+ = t^+[s_1^+/x_1]\ldots[s_n^+/x_n]$ (*equality modulo* λ*-conversion*). *Furthermore, if* $t \in \mathscr{T}$ *then* $(t[s_1/x_1]\ldots[s_n/x_n])^+ = t[s_1^+/x_1]\ldots[s_n^+/x_n]$.

Note that the previous fact holds even in case t is flexible (i.e. $t = (hu)$ and h is a variable). For instance, $((ha)[\lambda y.(y \multimap b)/h])^+ = (\lambda y.(y \multimap b)a)^+ = (a \multimap b)^+ = \top$ and $((ha)[(\lambda y.(y \multimap b))^+/h]) = (ha)[\lambda y.(y \multimap b)^+/h] = (ha)[\lambda y.\top/h] = \top$. Thus, the mapping commutes with λ-conversion.

In order to formalize the idea of 'proofs with terms in $\mathscr{T}$' we introduce the following definition.

Definition 6 ($\mathscr{T}$*-proofs*). A $\mathscr{T}$-proof π for a sequent s is a Forum proof for s in which for each occurrence of $\forall_l$ that introduces a new term t, $t \in \mathscr{T}$.

Since we are interested in studying a proof in Forum of a sequent in *ehhf*, we need to characterize all sequents originated by an *ehhf*-sequent applying Forum rules (and in particular applying the rule $\forall_l$).

Definition 7 (*Quasi ehhf-sequent*). A quasi *ehhf*-sequent is a Forum sequent either of the form $\Gamma; \Delta \longrightarrow_\Sigma \Omega$, or $\Gamma; \Delta \xrightarrow{F}_\Sigma \mathscr{A}$ where Γ is a set of $\mathscr{D}$-formulas over $\mathscr{T}$, Δ denotes a multi-set of Forum formulas $D\overline{[s/x]}$ where $\forall D$ is a $\mathscr{D}$-formula over $\mathscr{T}$, and Ω ($\mathscr{A}$) denotes a multi-set of formulas $G\overline{[s/x]}$ where $\forall G$ is a $\mathscr{G}$-formula over $\mathscr{T}$.

The application of the normalization to a quasi *ehhf*-sequent s, namely $\mathrm{s}^\bullet$, is defined by applying the normalization to each of its components: if $E = F\overline{[s/x]}$ then we use $E^\bullet$ to denote $F\overline{[s^+/x]}$, and if Δ is a multi-set of formulas, then $\Delta^\bullet$ is the multi-set $\{F^\bullet \mid F \in \Delta\}$. In a Forum proof for an *ehhf*-sequent with terms in $\mathscr{T}$, it is possible to isolate a partial proof tree consisting of quasi *ehhf*-sequents. Formally, we have the following definition.

Definition 8 (*Quasi proof tree*). Given a Forum proof π of a quasi *ehhf*-sequent s, the quasi proof tree of π is the maximal partial proof tree of π rooted with s and consisting of quasi *ehhf*-sequents only.

Note that a quasi proof tree may have nonaxiom leaves. For instance, if the root s is obtained by applying a rule with *non*quasi *ehhf*-sequents as premises, then s is a leaf of the corresponding quasi proof tree of π. Furthermore, by definition, given a Forum proof of a quasi *ehhf*-sequent there exists a unique, non-empty, quasi proof tree of π. We will use this property to prove the following lemma.

Lemma 9. *Let* s *be the quasi ehhf-sequent* $\Gamma; \Delta \longrightarrow \Omega$. *Then, if* s *has a proof in Forum, then* $\mathrm{s}^\bullet = \Gamma; \Delta^\bullet \longrightarrow \Omega^\bullet$ *has a* $\mathscr{T}$*-proof.*

Proof. Let π be the Forum proof for s. The proof is by induction on the length of the quasi proof tree of π.

- If the length is 1 and the axiom is an instance of $\top_r$ then the thesis immediately holds. If the axiom is an instance of *initial*, Δ are Ω singletons, consisting of $A\overline{[t/x]}$ and $B\overline{[s/y]}$, respectively. By hypothesis A is an atomic $\mathscr{D}$-formula over $\mathscr{T}$ (hence rigid) and B is an atomic $\mathscr{G}$-formulas over $\mathscr{T}$ (possibly flexible) and $A\overline{[t/x]} = B\overline{[s/y]}$ (modulo λ-conversion). By Fact 5, $(A\overline{[t/x]})^+ = A\overline{[t^+/x]}$ and $(B\overline{[s/y]})^+ = B\overline{[s^+/y]}$. Thus, it follows that $A\overline{[t^+/x]} = B\overline{[s^+/y]}$.
- If the length of the proof for s is greater than 1, the proof is by cases on its last rule.
  - The last rule is $\multimap_l$ with principal formula $F = H \circ\!- G$. We first observe that by hypothesis $F = D\overline{[s/x]}$, and $\forall D$ is a $\mathscr{D}$-formula. Since variables of type o are not $\mathscr{D}$-formulas, D and F must have the same top-level connective, i.e. $D = H' \circ\!- G'$ and $H = H'\overline{[s/x]}$ and $G = G'\overline{[s/x]}$. Since, $\delta_1 \vdash_F \Gamma; H, \Delta_1 \longrightarrow \Omega_1$ and $\delta_2 \vdash_F \Gamma; \Delta_2 \longrightarrow G, \Omega_2$, by inductive hypothesis, $\gamma_1 \vdash_F \Gamma; H^\bullet, \Delta_1^\bullet \longrightarrow \Omega_1^\bullet$ and $\gamma_2 \vdash_F \Gamma; \Delta_2^\bullet \longrightarrow G^\bullet, \Omega_2^\bullet$, and both γ_1 and γ_2 are $\mathscr{T}$-proofs. The thesis follows by applying again the $\multimap_l$ rule selecting as principal formula: $H^\bullet \circ\!- G^\bullet$, i.e. $(H \circ\!- G)^\bullet$, obtaining a $\mathscr{T}$-proof for the initial sequent.
  - The last rule is $\forall_l$ with principal formula $F = \forall x.F'\overline{[s/y]}$, where $\forall F'$ is a $\mathscr{D}$-formula (x does not occur free in $s_1 \ldots s_n$). Now we have that $\delta \vdash_F \Gamma; F'\overline{[s/y]}[t/x], \Delta \longrightarrow \Omega$ and, by inductive hypothesis, there exists a $\mathscr{T}$-proof γ s.t. $\gamma \vdash_F \Gamma; F'\overline{[s^+/y]}[t^+/x], \Delta^\bullet \longrightarrow \Omega^\bullet$. The thesis follows by applying again the $\forall_l$ rule selecting as principal formula: $(F'\overline{[s^+/y]}[t^+/x])$. Since t^+ is a $\mathscr{T}$-term the resulting proof is still a $\mathscr{T}$-proof.

– Let the last rule be an instance of $\multimap_r$ with principal formula $G[\overline{t/x}] = A \multimap B$. If G is an implication $A' \multimap B'$, the thesis immediately follows by applying the inductive hypothesis. Otherwise, if G is a flexible atomic formula then $(G[\overline{t^+/x}])$ converts to $(A \multimap B)^+ = \top$ (i.e. one of the terms introduced by $\forall_l$ is equal to $A \multimap B$) and the resulting sequent is a Forum axiom. A similar argument applies if the last rule is a right-introduction rule for $\Rightarrow$.

– Let the last rule be an instance of $\mathbin{\rotatebox[origin=c]{180}{\&}}_r$, with principal formula $G[\overline{t/x}] = A_1 \mathbin{\rotatebox[origin=c]{180}{\&}} A_2$. If G is the $\mathscr{G}$-formula $A_1' \mathbin{\rotatebox[origin=c]{180}{\&}} A_2'$, the thesis immediately follows from the inductive hypothesis. If G is a flexible atomic formula we can write $G[\overline{t/x}]$ as $y_1 \mathbin{\rotatebox[origin=c]{180}{\&}} y_2[A_1/y_1, A_2/y_2]$, where the $\mathscr{G}$-formula y_i is a 'dummy' variable for $i:1,2$. The premise of the last rule corresponds then to the sequent $\Gamma; \Delta \longrightarrow y_1[A_1/y_1], y_2[A_2/y_2], \Omega$. By induction hypothesis, $\xi \vdash_F \Gamma; \Delta \longrightarrow y_1[A_1/y_1], y_2[A_2/y_2], \Omega$, thus there exists γ s.t. $\gamma \vdash_F \Gamma; \Delta^\bullet \longrightarrow y_1[A_1^+/y_1], y_2[A_2^+/y_2], \Omega^\bullet$. Applying the rule $\mathbin{\rotatebox[origin=c]{180}{\&}}_r$ we obtain δ s.t. $\delta \vdash_F \Gamma; \Delta^\bullet \longrightarrow A^+ \mathbin{\rotatebox[origin=c]{180}{\&}} B^+, \Omega^\bullet$. Now, $A^+ \mathbin{\rotatebox[origin=c]{180}{\&}} B^+ = (A \mathbin{\rotatebox[origin=c]{180}{\&}} B)^+ = (G[\overline{t/x}])^+$ and, by hypothesis, G is a flexible atomic formulas such that its nested sub-terms are $\mathscr{T}$-terms, i.e. G is a $\mathscr{T}$-term. Hence, $(G[\overline{t/x}])^+ = G[\overline{t^+/x}]$. It follows that $\delta \vdash_F \Gamma; \Delta^\bullet \longrightarrow G[\overline{t^+/x}], \Omega^\bullet$.

A similar argument applies to the other cases. □

The following corollary holds.

Corollary 10 (Completeness w.r.t. $\mathscr{T}$-provability). *Let* s *be the ehhf-sequent* $\Gamma; \Delta \longrightarrow_\Sigma \Omega$ *with terms in* $\mathscr{T}$. *If* s *is provable in Forum then there exists a* $\mathscr{T}$*-proof for* s.

The above restriction to the set of terms $\mathscr{T}$ allows to state a completeness theorem between provability for *ehhf* and Forum w.r.t. the subset of formulas considered in *ehhf*. As a consequence of this corollary, we have the following theorem.

Theorem 11 (Completeness w.r.t. Forum). *Let* s *be the ehhf-sequent* $\Gamma; \Delta \longrightarrow_\Sigma \Omega$ *with terms in* $\mathscr{T}$, *then* s *is provable in ehhf if and only if* s *is provable in Forum.*

Proof. Based on Corollary 10, we can restrict ourself to consider $\mathscr{T}$-provability, which preserves well-formed *ehhf*-sequents. To conclude, we note that for sequents closed under instantiation the *ehhf*-system is nothing but a specialization of Forum to *ehhf*-formulas (as in the first-order case). The backchaining rule is then an instance of the one given in [25] also presented for a different formulation of linear logic in [29]. □

4. Another higher-order fragment

In the previous section, we have presented a (higher-order) logic programming language that provides:
– quantification over predicate variables;

– flexible $\mathscr{G}$-formulas (but rigid atomic $\mathscr{D}$-formulas);
– a universe of terms restricted to formulas without implications.

As in λProlog such an extension provides many useful programming examples such as abstract syntax for representing languages with binding constructs, and continuation passing. However, the restrictions on the language rule out other interesting *meta-programming* examples. For instance, it is not possible to write a clause like

$$compile(S,T)\,\&\,(T \Rightarrow execute(G)) \Rightarrow compile\_and\_execute(S,G),$$

i.e. a compile-and-execute routine parametric over the input program S. This formula is not a $\mathscr{D}$-formula in that T is not rigid and it occurs in $\mathscr{D}$-position. Furthermore, since the universe of terms is restricted to formulas without implication, even if we admitted such an occurrence of T, we could only instantiate T with trivial programs (i.e. consisting of a collection of facts).

In this section, we will present a slightly different higher-order formulation of the first-order version of *ehhf* in which we will admit quantification over *program* variables (such as T before) as well as terms with implications. Unfortunately, in order to obtain a completeness result we will have to rule out λ-terms from the language. The resulting extension can be seen as an enrichment of the first-order formulation of *ehhf* with terms of type o, we will call it $ehhf_D$ to emphasize that universal quantification is extended to variables ranging over $\mathscr{D}$-formulas. More formally, the higher-order language $ehhf_D$ is defined as follows.

Definition 12 (*$\mathscr{T}_D$-terms and formulas*). The set of terms $\mathscr{T}_D$ is the smallest set such that:

- A variable x of type τ (possibly o) is a $\mathscr{T}_D$-term of type τ;
- a rigid term A_r of type σ of the form $(h\ t_1\ \dots\ t_n)$ is in $\mathscr{T}_D$ whenever t_i: τ_i is in $\mathscr{T}_D$ for $i:1,\dots,n$, and h is a constant with type $\tau_1 \to \cdots \to \tau_n \to \sigma$;
- a $\mathscr{D}$-formula, as defined by the following grammar, is a $\mathscr{T}_D$-term of type o:

$$\mathscr{D} ::= \forall x.\mathscr{D} | \mathscr{D}_1\,\&\,\mathscr{D}_2 \,|\, \mathscr{G} \Rightarrow \mathscr{D} \,|\, H \circ\!- \mathscr{G} \,|\, H$$

$$\mathscr{G} ::= \mathscr{G}_D \,|\, \mathscr{G} \parr \mathscr{G} \,|\, \mathscr{G}\,\&\,\mathscr{G} \,|\, \mathscr{D} \multimap \mathscr{G} \,|\, \mathscr{D} \Rightarrow \mathscr{G} \,|\, \forall x.\mathscr{G} \,|\, A_r \,|\, \bot \,|\, \top$$

$$H ::= H \parr H \,|\, A_r$$

$$\mathscr{G}_D ::= \mathbf{v} \multimap A_r \,|\, \mathbf{v} \Rightarrow A_r,$$

where $\mathbf{v}$ is a variable of type o, and A_r a rigid atomic formula built over $\mathscr{T}_D$-terms.

Thus, $ehhf_D$-terms of type o can be either variables or well-formed $\mathscr{D}$-formulas (i.e. we allow nested occurrences of terms with implications). Note that $\forall x.x$ is not a $\mathscr{D}$-formula, whereas $\forall x.(x \multimap a) \Rightarrow b$ is (i.e. variables of type o must occur within an implicational goal). Also note that the first-order formulation of $ehhf_D$ coincides with the first-order formulation of *ehhf*.

As in the previous section, our aim is to show that a Forum proof of an $ehhf_D$-sequent can be carried out using $ehhf_D$-terms only. To achieve this aim we need to define a mapping $v : \mathscr{T}_F \rightsquigarrow \mathscr{T}_D$ which preserves provability. We first define the set of formulas arising by substituting arbitrary terms in *ehhf*$_D$-formulas.

Definition 13 (*Quasi $\mathscr{D}$- and $\mathscr{G}$-formulas*). A quasi $\mathscr{D}$-($\mathscr{G}$-)formula E is obtained from a $\mathscr{D}$-($\mathscr{G}$-)formula F instantiating the free variables in F with arbitrary Forum-terms.

Quasi $\mathscr{D}$- and $\mathscr{G}$-formulas can be characterized by the following grammar:

$$\mathbf{D} ::= \forall x.\mathbf{D} \mid \mathbf{D} \& \mathbf{D} \mid \mathbf{G} \Rightarrow \mathbf{D} \mid \mathbf{H} \circ\!- \mathbf{G} \mid \mathbf{H}.$$

$$\mathbf{G} ::= \mathbf{G} \parr \mathbf{G} \mid \mathbf{G} \& \mathbf{G} \mid \mathbf{F} \rightsquigarrow \mathbf{A} \mid \mathbf{D} \rightsquigarrow \mathbf{G} \mid \forall x.\mathbf{G} \mid \mathbf{A} \mid \bot \mid \top .$$

$$\mathbf{H} ::= \mathbf{H} \parr \mathbf{H} \mid \mathbf{A},$$

where $\rightsquigarrow$ is one of $\multimap$, $\Rightarrow$, $\mathbf{F}$ is an arbitrary Forum formula, and $\mathbf{A}$ is an atomic formula with arbitrary (Forum) subterms. The mapping v should map all non-*ehhf*$_D$-terms into a fixed *ehhf*$_D$-term. For terms of type $\tau \neq o$ we can simply choose a special constant $c : \tau$. For terms of type o this simple idea will not work. In fact, terms of type o may occur in formula position and may be selected in backchaining steps. Following [6] the solution is to inspect the Forum proof of a *ehhf*$_D$-sequent and to keep trace of all the possible occurrences of variables of type $\tau \rightarrow \cdots \rightarrow o$ in $\mathscr{G}$-position in the sequents occurring in the proof tree. Based on this information we will build a formula that will succeed in any context where a variable of type o may occur. We first need to define the following set of formulas. Given a Forum proof π,

$\rho(\pi) = \{A$ s.t. A is a (rigid) atomic formula and either $\boldsymbol{v} \multimap \boldsymbol{A}$ or $\boldsymbol{v} \Rightarrow \boldsymbol{A}$ is a subformula of a formula occurring in π and $\boldsymbol{v}$ is a variable of type $o\}$.

The formula used as normal form for non-*ehhf*$_D$-formulas is then defined as follows.

Definition 14 (*$\mathscr{T}_D$-normal form*). Let π be a Forum proof and c be a *new* constant of type o, then

$$\mu_\pi = \begin{cases} \&_{A \in \rho(\pi)} C_A & \text{if } \rho(\pi) \neq \emptyset, \\ c & \text{otherwise,} \end{cases}$$

where $C_A = \forall_{\tau_1} x_1 . \ldots . \forall_{\tau_n} x_n . \, p(x_1, \ldots, x_n) \circ\!- \top$ whenever $A = p(t_1, \ldots, t_n)$, $t_i : \tau_i$ for $i : 1, \ldots, n$.

Given a Forum proof π, the normalization map is then defined as follows:

Definition 15 (*$\mathscr{T}_D$-normalization*). Let t be a term in λ-normal form. The $\mathscr{T}_D$-normalization of t w.r.t. a proof π, written $v(t)$, is defined inductively as follows:
- if t is a typed constant or a variable, then $v(t) = t$;
- if $t = (h \; t_1 \; \ldots \; t_n)$, a rigid term, them $v(t) = (h \; v(t_1) \; \ldots \; v(t_n))$;
- if t has type o, it is not a variable, and it is not a $\mathbf{D}$-formula, then $v(t) = \mu_\pi$;

- if t has type o and t is a **D**-formula (not in atomic form), then $v(t) = v^{-}(t)$ where v^{-} is the mapping defined by the following mutual recursion:

$$v^{-}(\mathbf{G} \Rightarrow \mathbf{D}) = v^{+}(\mathbf{G}) \Rightarrow v^{-}(\mathbf{D}), \quad v^{+}(\mathbf{D} \multimap \mathbf{G}) = v(\mathbf{D}) \multimap v^{+}(\mathbf{G}),$$

$$v^{-}(\mathbf{H} \circ\!- \mathbf{G}) = v^{-}(\mathbf{H}) \circ\!- v^{+}(\mathbf{G}), \quad v^{+}(\mathbf{D} \Rightarrow \mathbf{G}) = v(\mathbf{D}) \Rightarrow v^{+}(\mathbf{G}),$$

$$v^{-}(\mathbf{D} \,\&\, \mathbf{D}) = v^{-}(\mathbf{D}) \,\&\, v^{-}(\mathbf{D}), \quad v^{+}(\mathbf{G} \,\&\, \mathbf{G}) = v^{+}(\mathbf{G}) \,\&\, v^{+}(\mathbf{G}),$$

$$v^{-}(\forall_\tau v.\mathbf{D}) = \forall_\tau v.v^{-}(\mathbf{D}), \quad v^{+}(\mathbf{G} \parr \mathbf{G}) = v^{+}(\mathbf{G}) \parr v^{+}(\mathbf{G}),$$

$$v^{-}(\mathbf{H} \parr \mathbf{H}) = v^{-}(\mathbf{H}) \parr v^{-}(\mathbf{H}), \quad v^{+}(\forall_\tau v.\mathbf{G}) = \forall_\tau v.v^{+}(\mathbf{G}),$$

$$v^{-}(\mathbf{A}) = v^{+}(\mathbf{A}) = v(\mathbf{A}) \text{ whenever } \mathbf{A} \text{ is atomic.}$$

– $v(t) = c_\tau$ where $t : \tau$ and c_τ is a fixed constant with type τ, in all the other cases.

Note that in the base case for $v^{+}(\mathbf{A})$, i.e. when $\mathbf{A}$ is an atomic formula, we have to normalize the subterms of $\mathbf{A}$, namely $v^{+}(\mathbf{A}) = v(\mathbf{A})$. Remember that the language *ehhf*$_D$ allows variables in $\mathscr{D}$-position in the scope of an implicational goal. We can capture occurrences of non-$\mathscr{D}$-formulas as soon as they appear in a proof through the following subcases in the definition of $v^{+} : v^{+}(\mathbf{D} \multimap \mathbf{G}) = v(\mathbf{D}) \multimap v^{+}(\mathbf{G})$ and $v^{+}(\mathbf{D} \Rightarrow \mathbf{G}) = v(\mathbf{D}) \Rightarrow v^{+}(\mathbf{G})$. The following proposition holds.

Proposition 16. *Let t be a term* (*in normal form*) *with free variables $x_1 \ldots x_n$, s.t. $t \in \mathscr{T}_D$ then $v(t[s_1/x_1]\ldots[s_n/x_n]) = t[v(s_1)/x_1]\ldots[v(s_n)/x_n]$ and if t is a $\mathscr{D}$-formula then $v^{-}(t[s_1/x_1]\ldots[s_n/x_n]) = t[v(s_1)/x_1]\ldots[v(s_n)/x_n]$.*

Now, according to what we did in Section 3, we first introduce the notion of *quasi ehhf*$_D$-sequent, i.e. a sequent either of the form $\Gamma; \Delta \longrightarrow_\Sigma \Omega$ or $\Gamma; \Delta \xrightarrow{F}_\Sigma \mathscr{A}$ such that Γ and Δ are multi-set of **D**-formulas, F is a **D**-formula, and Ω is a multi-set of **G**-formulas. The application of the normalization to a quasi *ehhf*$_D$-sequent s, namely s$^\bullet$, is defined by applying the normalization to each of its components: if $E = F\overline{[s/x]}$ then we will use $E^\bullet$ to denote $F\overline{[v(s)/x]}$, and if Δ is a multi-set consisting of formulas of the form $F\overline{[s/x]}$, then $\Delta^\bullet$ is the multi-set $\{F\overline{[v(s)/x]} \mid F\overline{[s/x]} \in \Delta\}$. Finally, given a Forum proof π of a quasi *ehhf*$_D$-sequent s we define the *quasi proof tree* of π as the maximal partial proof tree rooted with s and consisting of quasi *ehhf*$_D$-sequents only. Then, the following lemma holds.

Lemma 17. *Let* s *be the quasi ehhf$_D$-sequent $\Gamma; \Delta \longrightarrow \Omega$. Then, if* s *has a proof in Forum, then* s$^\bullet = \Gamma^\bullet; \Delta^\bullet \longrightarrow \Omega^\bullet$ *has a $\mathscr{T}_D$-proof.*

Proof. The proof is by induction on the length of the quasi proof tree for s. The proof of the basis is similar to that of Lemma 9. If the length of the proof for s is greater than one, the proof is by cases on the last rule. If the last rule is a left-introduction rule the thesis follows applying the inductive hypothesis to the premises. Let us assume that the last rule of the proof π of s is an instance of Forum $\multimap_r$ with principal formula

$A \multimap B$, i.e., $s = \Gamma; \Delta \longrightarrow A \multimap B, \Omega$, and the premise $\Gamma; \Delta, A \longrightarrow B, \Omega$ is provable in Forum. By hypothesis, $A \multimap B = D\overline{[s/x]} \multimap G\overline{[s/x]}$ where $D \multimap G$ is a $\mathscr{G}$-formula. By definition of $\mathscr{G}$-formulas either D is a $\mathscr{D}$-formula or D is a variable of type o.

i. if D is a $\mathscr{D}$-formula a proof for the sequent $\Gamma^\bullet; \Delta^\bullet \longrightarrow (A \multimap B)^\bullet, \Omega^\bullet$ can be reconstructed by applying $\multimap_r$ to the sequent $\Gamma^\bullet; \Delta^\bullet, A^\bullet \longrightarrow B^\bullet, \Omega^\bullet$ which is provable by inductive hypothesis.

ii. If D is a variable then $D\overline{[s/x]} = \phi$ must be a closed term, i.e. $D = x_i$ and $[\phi/x_i]$ for some i is a component of $\overline{[s/x]}$. Furthermore, by definition of $\mathscr{D}$-formulas, $B = G\overline{[s/x]}$ must be a rigid atomic formula, say $p(\bar{u})$.

ii.a. if ϕ is a quasi $\mathscr{D}$-formula then $\nu(\phi)$ is a closed $\mathscr{D}$-formula by Fact 16 thus it is possible to apply the inductive hypothesis and conclude as in case (i).

ii.b. If ϕ is not a quasi $\mathscr{D}$-formula, then $\nu(\phi) = \mu_\pi$. As a consequence, $A^\bullet = \nu(\phi) = \mu_\pi$ and it remains to show that $\Gamma^\bullet; \Delta^\bullet, \mu_\pi \longrightarrow B^\bullet, \Omega^\bullet$ is provable. Now, $B^\bullet$ is an atomic formula and by hypothesis only quasi $\mathscr{G}$-formulas over $\multimap, \Rightarrow, \forall, \&, ⅋, \bot$ and $\top$ may occur in Ω. Notice that the right-introduction rules for these connectives are all deterministic (*synchronous* [1]) and, when exhaustively applied to the formulas in Ω, they yield a set of premises $\Gamma_i^\bullet; \Delta_i^\bullet, \mu_\pi \longrightarrow B^\bullet, \mathscr{A}_i^\bullet$ for $i : 0 \ldots k$ in which the right-hand sides have been reduced to multi-sets of atomic formulas containing $\mathscr{B}^\bullet$ and all left-hand sides contains μ_π (since synchronous rules cannot split but only copy the bounded contexts in the lower sequent to all the premises). In case $k = 0$ an instance of $\top_r$ occurred in Ω and thus the thesis immediately follows. In case $k > 0$, for each premise we simply need to focus the proof on the formula μ_π. In fact, since $B = p(\bar{s})$, by definition $\top \multimap p(\bar{x})$ is a conjunct of μ_π such that $\bar{x}$ are variables universally quantified in μ_π. Thus, we can build the following Forum proof:

$$\dfrac{\dfrac{}{\Gamma_i^\bullet; \xrightarrow{C} B^\bullet}\,(initial) \qquad \dfrac{}{\Gamma_i^\bullet; \Delta_i^\bullet \longrightarrow \top, \mathscr{A}_i^\bullet}\,\top_r}{\Gamma_i^\bullet; \Delta_i^\bullet \xrightarrow{\top \multimap C} B^\bullet, \mathscr{A}_i^\bullet}\,\multimap_l$$

$$\vdots\ \forall_l + \&_l$$

$$\dfrac{\Gamma_i^\bullet; \Delta_i^\bullet \xrightarrow{\mu_\pi} B^\bullet, \mathscr{A}_i^\bullet}{\Gamma_i^\bullet; \Delta_i^\bullet, \mu_\pi \longrightarrow B^\bullet, \mathscr{A}_i^\bullet}\,decide \quad .$$

where, to turn the sequent $\Gamma_i^\bullet; \xrightarrow{C} B^\bullet$ into an instance of the *initial* axiom we simply need to select the right conjunct from μ_π and to instantiate its variables with $\bar{s}$ so that C coincides with $p(\bar{s})$.

The proofs for the remaining left-rules and for the structural rules apply similar arguments. □

Note that the requirement that only *synchronous* connectives occur in $\mathscr{G}$-formulas is necessary to prove the case ii.b. Based on Lemma 17 we can state the following results.

Theorem 18 (Completeness of $ehhf_D$-provability). *Let* s *be the* $ehhf_D$*-sequent* $\Gamma; \Delta \longrightarrow_{\Sigma} \Omega$. *Then,* s *is provable in* $ehhf_D$ *if and only if* s *is provable in Forum.*

Proof. If there exists a Forum-proof for s, then by Lemma 17 there exists a $\mathscr{T}_D$-proof δ for s. The thesis now holds by observing that the *ehhf*-proof system is obtained by restricting Forum to the considered class of formulas. □

5. Practice

In this section, we will introduce a methodology to write executable specifications in *ehhf* and $ehhf_D$. According to the view of proofs as computations via rewriting, state-based computations can be represented by transition rules of the form OLD_STATE *rewrites in* NEW_STATE. The current state is then described by a term $cons(t_1, \ldots, t_n)$ where *cons* is a constructor, and the t_i's represent the values of the state components. Note that rewriting is applied modulo congruence rules, whereas it is customary in logic programming to consider nested terms as *uninterpreted* data structures. In [2], Andreoli and Pareschi introduce a method in linear logic programming to relax this constraint while preserving the above mentioned view. The idea is to split the state of a computation into a disjunction of its components, namely $t_1 \parr \cdots \parr t_n$, so that each component can now be considered both as an interpreted (i.e. defined by a predicate) or an uninterpreted term. Multi-conclusion clauses can be used to aggregate the components. For instance, the rewriting rule $cons(t_1, t_2)$ *rewrites in* $cons(s_1, s_2)$, which describes an update of the state represented through the constructor *cons*, can be written as the linear logic clause: $t_1 \parr t_2 \multimapinv s_1 \parr s_2$. The application of the previous rule (in a *bc* step) to a sequent whose right-hand side is $\{t_1, t_2\}$ yields a new sequent whose right-hand side is $\{s_1, s_2\}$ as desired. In the rest of the paper, we will focus on a representation of *instantaneous configurations* based on multi-sets of formulas: each uninterpreted component will denote a *value*, whereas each interpreted component will denote an *operation* whose semantics is specified by a $\mathscr{D}$-formula. Manipulation of values and operations is performed by the use of multi-headed clauses.

5.1. Refining the syntax of the sequents

In the rest of the paper, it will be convenient to introduce an explicit set of predicate symbols Σ_R used as constructors for values of a configuration (i.e. to represent database facts, values associated to locations, objects, etc.). Also, we will consider sequents in which the right-hand side is partitioned into two multi-sets of formulas: $\Gamma; \Delta \rightarrow_{\Sigma} \Omega \parallel \Theta$ with the intended meaning that Θ, the *values* of the configuration, is the multi-set of *all* atomic formulas ranging over Σ_R occurring in the right-hand side. Each time a value is produced during the proof it will be implicitly moved into the Θ-component of the current configuration. It is important to remark that a sequent $\Gamma; \Delta \rightarrow_{\Sigma} \Omega \parallel \Theta$ is logically equivalent to the *ehhf*-sequent $\Gamma; \Delta \longrightarrow_{\Sigma} \Omega \uplus \Theta$. The multi-set $\Omega \uplus \Theta$ will be

$$\cfrac{\cfrac{}{\Gamma, \Phi; \cdot \xrightarrow{\Phi}_{\Sigma} \cdot \,\|\, A_1, \ldots, A_n}\ \textit{initial}}{\Gamma, \Phi; \cdot \longrightarrow_{\Sigma} \cdot \,\|\, A_1, \ldots, A_n}\ \textit{decide!}$$

Fig. 2. The derived rule (*verify*).

used to represent the *global state* of a computation. We refer the reader to Section 8 for comparison with other approaches such as [2, 20].

To simplify the presentation of the examples, we will further refine the syntax of sequents. In our first example relating deduction and rewriting (Section 2) the idea was to *guess* the target term s in order to complete the proof of *t rewrites in s*. To simplify the notation we will assume that the guessed *final state* of the derivation is already part of Γ in the initial sequents. Formulas denoting final states (the residue we mentioned in the first example of Section 2) are defined as follows.

Definition 19 (*$\mathscr{R}$-formulas*). An $\mathscr{R}$-formula is a $\mathscr{D}$-formula of the form

$$\forall (A_1^1 \mathbin{⅋} \cdots \mathbin{⅋} A_{n_1}^1) \,\&\cdots\&\, (A_1^k \mathbin{⅋} \cdots \mathbin{⅋} A_{n_k}^k),$$

where A_j^i is an atomic formula over Σ_R for i: $1, \ldots, k$ and j: $1, \ldots, n_i$.

We will refer to a backchaining step over this type of formulas as the *verify*-rule of Fig. 2. In this rule, Φ is an $\mathscr{R}$-formula and the A_i's are Σ_R-atomic formulas. Note that, given the simple form of the considered $\mathscr{R}$-formulas, Ω and Δ must necessarily be empty (see definition of *initial* in Fig. 1). In the following, we will use the notation $\Gamma[\Phi]; \Delta \longrightarrow_\Sigma \Omega \,\|\, \Theta$ to isolate an $\mathscr{R}$-formula occurring in Γ. These conventions have the following operational counterpart. Consider Φ as a meta-variable. The computation is guided by the simplification of the goals in Ω until the state of the sequents is reduced to a multi-set of values. At this point Φ can be unified with this set of values and the resulting $\mathscr{R}$-formula returned as an answer to the initial query. The algorithm for the reconstruction of Φ is given in Fig. 3, where in $s \rightsquigarrow s_1$ **and** s_2, s stands for the lower sequent of the considered rule and the s_i's for its premises. Here we have used an abstract syntax notation to represent formulas and we assume that $=$ corresponds to meta-level unification.[4] The formula $\circ$ indicates that the leaf of the considered branch is not a (*verify*) axiom, in this sense they can be considered as part of conditional branches, for which (as the conditional part of the rule *bc*) we need no Φ-answer.

We can exploit the powerful structure of $\mathscr{D}$-formulas to naturally integrate state-based computations with other logical features by an appropriate use of the conditional part. Let us assume that *logic programs* (in the sense stated by the embedding of hereditary Harrop formulas given in Proposition 1) are part of the unbounded context of sequents. Thus, given a rule $G_1 \& \cdots \& G_n \Rightarrow (H \multimapinv G)$ we can now view the conditions G_i either

[4] As customary in λ Prolog, $(pi\ \lambda x.F(x))$ is an abstract syntax representation for $\forall x.F(x)$.

$(verify):\ \Gamma[\Phi];\ \rightarrow_{\Sigma} \cdot \parallel \Theta$	$\rightsquigarrow\ \Phi = \parr\, \Theta$
$(initial):\ \Gamma[\Phi];\ \rightarrow_{\Sigma} \mathscr{A} \parallel \Theta$	$\rightsquigarrow\ \Phi = \circ$
$(\top_r):\ \Gamma;\Delta \longrightarrow_{\Sigma} \top, \Omega$	$\rightsquigarrow\ \Phi = \circ$
$(\forall_r)\ \Gamma[\Phi];\Delta \longrightarrow_{\Sigma} \forall_{\tau}x.A, \Omega$	$\rightsquigarrow\ \Gamma[\Psi];\Delta \longrightarrow_{\Sigma} A[y/x], \Omega \quad \Phi = (pi\ (\Psi\ y))$
$(\&_r):\ \Gamma[\Phi];\Delta \longrightarrow_{\Sigma} A_1\ \&\ A_2, \Omega$	$\rightsquigarrow\ \Gamma[\Phi_1];\Delta \longrightarrow_{\Sigma} A_1, \Omega$ **and** $\Gamma[\Phi_2];\Delta \longrightarrow_{\Sigma} A_2, \Omega \quad \Phi = \Phi_1 \& \Phi_2$
$(bc):\ \Gamma[\Phi];\Delta \xrightarrow{D}_{\Sigma} \mathscr{A}, \mathscr{A}'$	$\rightsquigarrow\ \Gamma; \cdot \longrightarrow_{\Sigma} G$ **and** $\Gamma[\Phi];\Delta \longrightarrow_{\Sigma} B, \mathscr{A}'$
$others:\ \Gamma[\Phi];\Delta \longrightarrow_{\Sigma} \Omega$	$\rightsquigarrow\ \Gamma'[\Phi];\Delta' \longrightarrow_{\Sigma} \Omega'$

Fig. 3. Reconstructing $\mathscr{R}$-formulas by using unification for the $[\cdot]$ component.

as further rewriting branches or simply as invocations of goals defined over logic programs being part of Γ.

As an example, in the following section we will give an *ehhf*-specification of the semantics of *small*, a simple imperative language. In all examples, when clear from the context, we will omit the type of the constants, of the quantifiers and the signatures in the sequents.

6. Specification of the semantics of an imperative language

The imperative language *small* has the following features: variables of integer type, expressions, and the statements: assignment, skip, if-then-else and while. In order to describe the semantics of this language we will consider *ehhf*-sequents as instantaneous configurations of the execution of a program. To represent the state of the data variables we will use atomic formulas of the form $var(x, v)$ where x is (the name of) a variable and v is its current value. In Fig. 4, we show the embedding of the different syntactic components of *small* into *ehhf*: each syntactic component is represented using a predicate whose semantics is defined by an *ehhf*-theory, we will call $\mathscr{M}$. We anticipate here that the execution of a (terminating) *small* program P will be fully described by a proof of the sequent $\mathscr{M}[\Phi]; \cdot \longrightarrow P^*$ where P^* is the goal formula obtained from P via the encoding of Fig. 4, and Φ is the $\mathscr{R}$-formula denoting the final state of the execution. The intermediate configurations will have the form $\mathscr{M}[\Phi]; \cdot \rightarrow_{\Sigma} \Omega_{\text{stat}} \parallel \Theta_{\text{vars}}$, where Θ_{vars} will represent the current state of the data variables, and Ω_{stat} the list of statements to be executed. Now, we will discuss the rules of the theory $\mathscr{M}$.

6.1. Semantics via an ehhf-executable specification

The semantics of expressions is specified by a predicate *eval* defined as follows:

$$eval(\bar{n}, \bar{n}) \multimapinv \top. \quad \textit{for each numeral } \bar{n}.$$

$$eval(X, V) \parr var(X, V) \multimapinv \top.$$

$$eval(E_1 = E_2, V) \multimapinv eval(E_1, V_1)\ \&\ eval(E_2, V_2)\ \&\ test(V_1, V_2, V).$$

$$
\begin{array}{ll}
(var\ x_1,\ldots,var\ x_n)^\star & ::= var(x_1,\bar{0}) \mathbin{\rotatebox[origin=c]{180}{\&}} \ldots \mathbin{\rotatebox[origin=c]{180}{\&}} var(x_n,\bar{0}),\\
skip^\star & ::= \bot,\\
(x := E)^\star & ::= assign(x,E^\star),\\
(if\ E\ the\ S_1\ else\ S_2)^\star & ::= cond(E^\star,S_1^\star,S_2^\star),\\
(while\ E\ do\ C)^\star & ::= while(E^\star,C^\star),\\
(S_1;S_2)^\star & ::= seq(S_1^\star,S_2^\star),\\
(program\ Dec\ in\ S)^\star & ::= Dec^\star \mathbin{\rotatebox[origin=c]{180}{\&}} S^\star.
\end{array}
$$

Fig. 4. The translation of *small* into *ehhf*.

$$eval(E_1 + E_2, V) \circ\!- eval(E_1, V_1)\ \&\ eval(E_2, V_2)\ \&\ sum(V_1, V_2, V).$$

$$\ldots\ Similarly\ for,\ E_1 - E_2,\ E_1 * E_2, etc.$$

The auxiliary predicates *test* and *sum*, both with arity 3, define equality and addition for integer values, e.g., *test* binds its third argument to *1* if the first two arguments are equal and to *0*, otherwise. We use the connective & in the body of the definition of *eval* in order to create different branches of a proof with a copy of the current bounded resources Θ_{var}. This allows us to evaluate each subexpression using a copy of the current state (used, e.g. by the second rule). The use of $\top$ in the base case of the definition of *eval* makes the evaluation of an expression to succeed independently from the current state of the bounded contexts (i.e. even when they are non-empty). The predicate *seq* is used to enforce sequentiality in the evaluation of the statements. In the style of continuation passing, we will evaluate the first argument of the predicate *seq*, with type $stat \to stat \to o$, accumulating the remaining statements in the second argument (the continuation). Formally, we have the following:

$$seq(seq(S_1, S_2), R) \circ\!- seq(S_1, seq(S_2, R)).$$

$$seq(\bot, R) \circ\!- R.$$

In the following, we will write S instead of $seq(S, \bot)$. Furthermore, we assume $\Theta = \{var(x, v) \mid x$ has value v$\}$. The semantics of each statement is then defined by cases on the first argument of *seq*.

For the assignment statement we have the following rule, we refer to as the *assign* rule:

$$seq(assign(X, E), R) \mathbin{\rotatebox[origin=c]{180}{\&}} var(X, V_1) \circ\!-$$

$$(var(X, V_1) \mathbin{\rotatebox[origin=c]{180}{\&}} eval(E, V_2))\&(var(X, V_2) \mathbin{\rotatebox[origin=c]{180}{\&}} R).$$

The semantics of assignment is expressed by the following derived *ehhf* scheme:

$$\frac{\mathscr{M}[\Phi]; \cdot \to_\Sigma eval(e, v) \parallel \Theta \quad \mathscr{M}[\Phi]; \cdot \to_\Sigma \cdot \parallel \Theta'}{\mathscr{M}[\Phi]; \cdot \to_\Sigma assign(x, e) \parallel \Theta}$$

where $\Theta = \{var(x,w)\} \uplus \Theta''$ and $\Theta' = \{var(x,v)\} \uplus \Theta''$. The expanded scheme derived by applying the rule *assign* is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathscr{M}[\Phi];\cdot \rightarrow_\Sigma eval(e,v) \parallel var(x,w),\Theta''}{\mathscr{M}[\Phi];\cdot \rightarrow_\Sigma (eval(e,v) \,⅋\, var(x,w)) \parallel \Theta''}\,⅋_r \quad \mathscr{M}[\Phi];\cdot \rightarrow_\Sigma \cdot \parallel var(x,v),\Theta''}{\mathscr{M}[\Phi];\cdot \rightarrow_\Sigma (eval(e,v) \,⅋\, var(x,w)) \& var(x,v) \parallel \Theta''}\,\&_r}{\mathscr{M}[\Phi];\cdot \xrightarrow{assign} assign(x,e) \parallel var(x,w),\Theta''}\,bc}{\mathscr{M}[\Phi];\cdot \rightarrow_\Sigma assign(x,e) \parallel var(x,w),\Theta''}\,decide!$$

The use of & in the body of the clause *assign* is used to create a copy of the current state, that, together with the old value of x, is used to evaluate e.

For the conditional statement we have the rule:

$$seq(cond(E,S_1,S_0),R) \circ\!- eval(E,1) \;\&\; seq(S_1,R).$$

$$seq(cond(E,S_1,S_0),R) \circ\!- eval(E,0) \;\&\; seq(S_0,R).$$

Finally, for the while-loop statement we have the rule:

$$seq(while(E,S),R) \circ\!- eval(E,1) \;\&\; seq(S,seq(while(E,S),R)).$$

$$seq(while(E,S),R) \circ\!- eval(E,0) \;\&\; R.$$

As for the assignment, we can specialize *bc* so as to obtain a transition system defining the operational semantics of the language *small*. The following lemma states an interesting property of the previous specification. With an abuse of notation, in the lemma we will use Φ to denote either an $\mathscr{R}$-formula $A_1 ⅋ \cdots ⅋ A_n$ or the corresponding multi-set $\{A_1,\ldots,A_n\}$.

Lemma 20 (Concatenation). *Let* s_1 *and* s_2 *be the encoding of two small statements. If the sequents* $\mathscr{M}[\Phi];\cdot \rightarrow_\Sigma s_1 \parallel \Theta$ *and* $\mathscr{M}[\Psi];\cdot \rightarrow_\Sigma s_2 \parallel \Phi$ *are provable in ehhf then* $\mathscr{M}[\Psi];\cdot \rightarrow_\Sigma seq(s_1,s_2) \parallel \Theta$ *is provable in ehhf.*

Proof. Since the sequent $\mathscr{M}[\Phi];\cdot \rightarrow_\Sigma s_1 \parallel \Theta$ is provable in *ehhf*, by construction of $\mathscr{M}$, it follows that there exists Θ' s.t. $\mathscr{M}[\Phi];\cdot \rightarrow_\Sigma \cdot \parallel \Theta'$ is provable, too. By the soundness of *ehhf*, the two sequents $\mathscr{M};\Phi \longrightarrow \Theta'$ and $\mathscr{M},\Psi;\cdot \longrightarrow s_2,\Phi$ are provable in Forum, i.e. in full linear logic.[5] Then, by applying the *cut*-rule of linear logic to $\mathscr{M};\Phi \longrightarrow \Theta'$ and to $\mathscr{M},\Psi;\cdot \longrightarrow s_2,\Phi$ we obtain a proof for the sequent $\mathscr{M};\Psi \longrightarrow s_2,\Theta'$. By the completeness of *ehhf* w.r.t. full linear logic, we also know that the sequent $\mathscr{M}[\Psi];\cdot \rightarrow_\Sigma s_2 \parallel \Theta'$ is provable in *ehhf*. Thus, to build a proof for the sequent $\mathscr{M}[\Psi];\cdot \rightarrow_\Sigma seq(s_1,s_2) \parallel \Theta$ we simply have to mimic each step of the proof

[5] For simplicity, we use the same notation used for Forum, i.e. $\Gamma;\Delta \longrightarrow \Omega$, to denote sequents of linear logic [15].

for $\mathscr{M}[\Psi];\cdot \rightarrow_\Sigma \mathrm{s}_1 \,\|\, \Theta$ as far as we obtain the sequent $\mathscr{M}[\Psi];\cdot \rightarrow_\Sigma \mathrm{s}_2 \,\|\, \Theta'$ that we know to be provable in *ehhf*. $\square$

The previous property can be read as a form of *cut* rule for concatenation of statements: Φ is used as auxiliary formula (intermediate state) to compute the final state of the concatenation of s_1 and s_2.

6.2. A simple form of function declarations

Functional terms can be directly represented in *ehhf* using λ-expressions. Let us enrich *small* with declarations of functions over integers of the form $f(X_1,\ldots,X_n)=E$. This declaration can be compiled into the lambda term $\lambda x_1.\ldots.\lambda x_n.E^*$, where E^* is the encoding of the expression E, so as to syntactically represent the scope of the parameters of the function f. Note that in higher-order logic programming languages based on the Simply Typed Lambda Calculus, like λProlog, lambda terms do not have an active computational role. The semantics of a function must be formalized by the logical component of our language (in this case by the *eval* predicate). To make things work, we need to store this definition in the environment and to modify the *eval* predicate in order to cope with function invocations. The first task is accomplished by adding pairs of the form $var(f,\lambda x_1.\ldots.\lambda x_n.E^*)$ to Θ_{vars}. In this pair the first argument is the name of the function and the second is its definition. To handle function invocations we use a special predicate $app(F,X_1,\ldots,X_n)$ whose semantics is defined as follows:

$$eval(app(F,X_1,\ldots,X_n,V) \parr var(F,D) \multimapinv$$
$$var(F,D) \parr ((eval(X_1,V_1)\ \&\ \ldots\ eval(X_n,V_n)\ \&\ eval(D(W_1,\ldots,W_n),V)).$$

Intuitively, to evaluate the invocation of F we first evaluate the parameters $X_1,\ldots,X_n$. Then, we keep on evaluating the expression which results by applying D to the values associated with the parameters (note that $D(W_1,\ldots,W_n)$ is a flexible term).

6.3. Proving properties of a program

In a sequent $\mathscr{M}[Post];\cdot \rightarrow_\Sigma Pre \,\|\, P^\star$ it is possible to encode a general condition over the initial state, *Pre*, and over the final state, *Post*, trying to prove the correctness of the program P w.r.t. the specified assertions. Differently from the previous sections, it will be necessary to consider more general forms of state formulas (i.e. not simply a disjunction of atoms). Let $s^n(0)$ be the expansion of a numeral $\bar{n}$. We will consider the following simple properties: $\Phi_1(x)$ =‘the variable x has a value greater than or equal to zero’, and $\Phi_2(x)$ =‘the variable x has a value greater than zero’. It is possible to use the following two $\mathscr{R}$-formulas as templates of states satisfying the previous properties: $\Phi_1(x) \equiv \forall_r y.var(x,s(y))\,\&\,var(x,0)$, and $\Phi_2(x) \equiv \forall_r y.var(x,s(y))$. Note that Φ_1 is modeled by a combination of $\forall$ and &: the first connective is used to generalize the value of the variable named x, whereas the second connective is used to prove the property for all possible cases. Consider now the assignment $S = assign(x,x+1)$.

Then, we have the following proof:

$$\cfrac{\cfrac{\cfrac{\delta_1}{\mathscr{M}[\Phi_2(x)]; \cdot \rightarrow_{\Sigma'} \cdot \parallel var(x, s(s(d)))}}{\mathscr{M}[\Phi_2(x)]; \cdot \rightarrow_{\Sigma'} S \parallel var(x, s(d))}\,bc \quad \cfrac{\cfrac{\delta_2}{\mathscr{M}[\Phi_2(x)]; \cdot \rightarrow_{\Sigma'} \cdot \parallel var(x, s(0))}}{\mathscr{M}[\Phi_2(x)]; \cdot \rightarrow_{\Sigma'} S \parallel var(x, 0)}\,bc}{\cfrac{\mathscr{M}[\Phi_2(x)]; \cdot \rightarrow_{\Sigma'} S \parallel var(x, 0) \,\&\, var(x, s(d))}{\mathscr{M}[\Phi_2(x)]; \cdot \rightarrow_{\Sigma} \Phi_1(x) \parr S \parallel \cdot}\,\forall_r + \parr_r}\,\&_r$$

In order to encode more complex properties it might be necessary to consider a set of simplification rules for predicates involved in the *pre* and *post* formulas. As an example, consider the two properties

$$\Phi_1(x, y) \equiv \forall V, W.var(x, V) \parr var(y, W) \parr V < W$$
$$\Phi_2(x, y) \equiv \forall V, W.var(x, V) \parr var(y, W) \parr V \leqslant W.$$

In order to handle the predicates $<$ and $\leqslant$ we need to enrich the theory $\mathscr{M}$ with further simplification rules like

$$var(X, s(V)) \parr var(Y, W) \parr V < W \multimapinv var(X, V) \parr var(Y, W) \parr V \leqslant W,$$

when proving a sequent like $\mathscr{M}[\Phi_2(x, y)]; \cdot \rightarrow_{\Sigma} S \parallel \Phi_1(x, y)$. Similar rules, combined with Lemma 20, allow one to compositionally verify properties of a concatenation of statements. Furthermore, given a pair of *pre* and *post* conditions (expressed as $\mathscr{R}$-formulas) the proof can be carried out automatically.

In [26], Miller proposed to use Forum for proving equivalences of programs. Given two programs P_1 and P_2, the idea is to prove both $P_1^*; \cdot \longrightarrow P_2^*$ and $P_2^*; \cdot \longrightarrow P_1^*$. *ehhf* (and in general any restriction of Forum) turns to be unsuitable for this task. In fact, to express this type of sequents we would need no distinction between clauses and goal formulas (note that P_1^*, possibly a very complicated formula, occurs in both sides of a sequent). On the other hand, such a freedom in the syntax of the sequents may cause problems when trying to automatically generate the proofs.

In this section, we have shown how the combination of linear logic and higher-order features typical of λProlog can interact in a natural way in the setting of *ehhf*. This combination provides many operational features which can be useful to specify the internal behavior of processes (i.e. an imperative program). In the following section, we will focus on the description of systems composed of many processes.

7. Meta programming

In this section, we will show that the fragment of Forum $ehhf_D$ can be used as a formal language to write specification of complex systems where it is necessary to handle *programs*, *processes* [9, 3], *objects* [5, 6, 11] or *agents* [4] at the object level. To justify our claim we will present the specification of a transaction-based system, involving data-management, multi-threaded computations and mobility of code.

We will start our example by modeling a simple database system supporting the following set of operations: *retrieve*(A), *add*(A), *delete*(A) with the meaning induced by the corresponding names. We will read an *ehhf*-sequent $\Gamma;\Delta \rightarrow_\Sigma \Omega \,\|\, \Theta$ as the snapshot of the database during the execution of a (collection of) transaction(s): Γ is the set of meta-rules describing the semantics of the primitives and of the user-defined rules, Δ is the multi-set of active *clauses*, Ω is a multi-set representing the current collections of transactions, and Θ represents the database, i.e. a multi-set of atomic formulas having form $db(A)$ with A a given atomic datum (fact). We will use $tr(id,T)$ to denote a transaction with identifier *id*. T, its code, is a term of the form $seq(B_1,\ldots,seq(B_n,\perp)\ldots)$. We anticipate here that we will employ the constructor *seq* to execute *each* transaction sequentially. User-defined rules are encoded in *ehhf* as follows: $tr(I,seq(A,R)) \multimapinv tr(I,seq(B_1,\ldots,seq(B_n,R)\ldots))$, where A is the head (definition) of the rule, the B_i's are atomic formulas that form its body, and R is a universally quantified variable.

The semantics of the basic database operations is given by the following simple *ehhf* rules inspired by previous works like [2, 18, 20]:

$$tr(I,seq(retrieve(A),R)) \,⅋\, db(A) \multimapinv tr(I,R) \,⅋\, db(A).$$

$$tr(I,seq(delete(A,R))) \,⅋\, db(A) \multimapinv tr(I,R).$$

$$tr(I,seq(add(A),R)) \multimapinv tr(I,R) \,⅋\, db(A).$$

It is also possible to define operations requiring a global test over the current state. An example is the deletion of all facts matching a given value A. We call *deleteall* the predicate implementing this operation. The semantics of *deleteall* is defined as follows:

$$match(A,B) \Rightarrow$$
$$tr(I,seq(deleteall(A),R)) \,⅋\, db(B) \multimapinv tr(I,seq(deleteall(A),R)).$$
$$tr(I,seq(deleteall(A),R)) \multimapinv not\_exist(A) \,\&\, tr(I,R).$$

where the predicate *not_exist* is defined as follows

$$not\_match(A,B) \Rightarrow not\_exist(A) \,⅋\, db(B) \multimapinv not\_exist(A).$$
$$not\_exist(A).$$

Note that the base case for the predicate *not_exist* is given as a unit clause and remember that in *ehhf* a unit clause can be applied to a goal formula only when the remaining bounded contexts are empty (see axiom *initial*). The following scheme will clarify the meaning of the previous definitions:

$$\dfrac{\dfrac{\dfrac{}{\Gamma;\cdot \rightarrow_\Sigma not\_exist(a) \,\|\, \cdot}\,bc \quad \vdots\, bc \quad}{\Gamma;\cdot \rightarrow_\Sigma not\_exist(a) \,\|\, \Theta \qquad \Gamma;\cdot \rightarrow_\Sigma tr(i,r) \,\|\, \Theta}\,\&_r}{\dfrac{\Gamma;\cdot \rightarrow_\Sigma not\_exist(a) \,\&\, tr(i,r) \,\|\, \Theta}{\Gamma;\cdot \rightarrow_\Sigma tr(i,seq(deleteall(a),r)) \,\|\, \Theta}\,bc}$$

The predicate *not_exist* consumes all the facts which do not match with a until an empty context is reached. If this subproof fails (i.e. a matches with some fact b in Θ) we can select the other clause for *deletall* in backtracking, i.e.:

$$\dfrac{\Gamma; \cdot \rightarrow_\Sigma match(a,b) \parallel \cdot \qquad \dfrac{\cdots}{\Gamma; \Delta \rightarrow_\Sigma tr(i, seq(deleteall(a), r)) \parallel \Theta}\,bc}{\Gamma; \Delta \rightarrow_\Sigma tr(i, seq(deleteall(a), r)) \parallel db(b), \Theta}\,bc$$

This way, we guarantee that *deleteall* will be carried out only when all facts matching a will be removed from the current state. In the previous rules we assume that the two relations *match* and *not_match* are computable, e.g. via a finite table.

In the following section, we will extend the model to concurrently executing transactions assigning them an interleaving semantics.

7.1. Concurrent transactions

Each transaction can be viewed as a separate thread of the execution of the system communicating with the other threads and with the data stored in the database. As explained in Section 5 we can employ $\parr$ as primitive to compose the representations of a collection of transactions $T_1, \ldots, T_n$, i.e. $tr(i_1, T_1) \parr \cdots \parr tr(i_n, T_n)$. The interleaving execution of the queries is induced by the nondeterminism implicit in a backchaining steps (multiple choices of the clause to apply and of the multi-set to rewrite). It remains to introduce primitives for communication and synchronization between transactions. The simplest form of synchronization can be achieved using the primitives $wait(A)$ and $wake(A)$, respectively, to suspend and re-activate a transaction. Their semantics is given via the following clause:

$$\begin{array}{l} tr(I, seq(wait(A), R)) \parr tr(J, seq(wake(A), S)) \multimapinv \\ \quad tr(I, R) \parr tr(J, S). \end{array}$$

Since each transaction is executed sequentially, to exploit the extended concurrent setting we must be able to execute subtasks in parallel. For this purpose, we introduce the primitive $fork(A, B)$ to create two concurrent sub-transactions that execute the tasks A and B, respectively. The semantics is given via the following clause:

$$\begin{array}{l} tr(I, seq(fork(A, B), R)) \multimapinv \\ \quad \forall x.\forall y.\forall z.\forall w. \\ \qquad tr(x, seq(A, seq(wake(z), \bot))) \parr tr(y, seq(B, seq(wake(w), \bot))) \\ \qquad\quad \parr tr(I, seq(wait(z), seq(wait(w), R))). \end{array}$$

The two *private* subtransactions x and y are created by hiding their names using universal quantification. These subtransactions will execute A and B before passing the control back to the parent-transaction (by using the synchronization primitives *wait*/*wake*).

$$\begin{array}{cl}
\Gamma;\Delta \rightarrow_{\Sigma} tr(i,m',t'),\Omega \parallel \Theta & \\
\hline
\Gamma;\Delta \xrightarrow{m}_{\Sigma} tr(i,m,seq(a,r)),\Omega \parallel \Theta & bc \\
\hline
\Gamma;m,\Delta \rightarrow_{\Sigma} tr(i,m,seq(a,r)),\Omega \parallel \Theta & decide \\
\hline
\Gamma;\Delta \rightarrow_{\Sigma} m \multimap tr(i,m,seq(a,r)),\Omega \parallel \Theta & \multimap_r \\
\hline
\Gamma;\Delta \xrightarrow{(call)}_{\Sigma} tr(i,m,seq(call(a),r)),\Omega \parallel \Theta & bc \\
\hline
\Gamma;\Delta \rightarrow_{\Sigma} tr(i,m,seq(call(a),r),\Omega \parallel \Theta & decide! \\
where\ tr(i,m,seq(a,r)) \circ\!\!- tr(i,m',t') \in \langle m \rangle. &
\end{array}$$

Fig. 5. Specialization of *bc* to the rule *call*.

7.2. Code mobility

The above language can be made more interesting by taking into consideration mobility of code. In the new model each transaction will be represented as $tr(i,M,T)$ where the new component M is the *code* defining the operations (methods) of the transaction i. This view is an abstraction of object-oriented systems in which a unit of data (which in our case is also a unit of execution) encapsulates its set of methods together with the private data. Based on the previous extension, transaction i with code M_i is able to perform the following operations:
- $call(A)$ to invoke a method A defined in M_i;
- $remote(j,A)$ to invoke an operation A defined in transaction j;
- $local(j,M,A)$ to locally invoke an operation A by first copying the code M from transaction j.

To give more details, we first need to fix a format for the code M in a transaction $tr(i,M,T)$. Taking advantage of the higher-order nature of $ehhf_D$, we define M as an $ehhf_D$-theory itself. M will be a conjunction of $\mathscr{D}$-formulas of the form $\forall\bar{x}.\forall I.\forall R.tr(I,M,seq(A,R)) \circ\!\!- tr(I,M',seq(B_1,\dots,seq(B_n,R)))$, where A is the method interface and $B_1,\dots,B_n$ is the body. Such a method can update the transaction code M rewriting it into M' (e.g. extensions, overriding of methods). A constant method (i.e. that returns a fixed value) corresponds to an attribute: overriding it corresponds to update the value of the attribute. The specification of the semantics of *call* is defined by the following $ehhf_D$-clause:

$$tr(I,M,seq(call(\mathscr{A}),R)) \circ\!\!-$$
$$M \multimap tr(I,M,seq(A,R)).$$

The resulting inference rule is shown in Fig. 5. Note that we use a variable of type o, namely M, in $\mathscr{D}$-position within a goal so as to *fire* the methods encapsulated in the transaction I. Universal quantification in goal formulas can be used in order to protect the code of each transaction (see e.g. [9, 20, 23]). The semantics of *remote* is given via the following $ehhf_D$-clause:

$$tr(I, M, seq(remote(J, A), R)) \parr tr(J, N, S) \circ\!- $$
$$tr(I, M, R) \parr tr(J, N, seq(call(A), S)).$$

Differently from *call*, in the clause *remote* we have to synchronize the two transactions I and J, enforcing transaction J to execute the task A on behalf of transaction I. Finally, the semantics of *local* is given via the following $ehhf_D$-clause:

$$tr(I, M, seq(local(J, N, A), R)) \parr tr(J, N, S) \circ\!-$$
$$(N \multimap tr(I, M, seq(A, R)) \parr tr(J, N, S).$$

In this case the execution thread of transaction J is not modified, however, we fire the code N of J in order to execute A in I. Though, in the previous example we have abstracted away many details from the specification of the transaction-based system, we feel that it gives a clear picture of the descriptive power of $ehhf_D$. Many other operational mechanisms can be modeled by similar ideas as we shown in [3, 4, 11].

8. Conclusions

The main contribution of this paper is the definition of two complete fragments of higher-order linear logic, called *ehhf* and $ehhf_D$, which incorporate features of higher-order logic programming and of linear logic programming. According to the logic of higher-order hereditary Harrop formulas, in *ehhf* we allow quantification over variables ranging over goal formulas. On the other hand, in $ehhf_D$ we have introduced quantification over variables ranging over programs. Though, it seems difficult to accommodate the features of the two fragments into one language without losing completeness, their integration can be admitted at the *interpreter level*, as for instance in some of the implementations of λProlog where it is possible to specify programs which are actually non-hereditary Harrop formulas. For these two fragments we have provided a methodology to write specifications (Section 5) and the logical foundation (i.e. a proof theoretical presentation) to execute them (Section 2).

Concerning linear logic programming, the novelty of our approach with respect to previous works like Lolli [18] and Lygon [17] is in the different operational interpretation of proofs. In particular, in our framework we try to characterize computations as rewriting steps so as to *observe* their final state; in this sense the final state is the answer to the initial query. As opposite in Lolli and Lygon the result of a computation is a computed answer substitution associated with the free variables of the initial query. Another difference is in the set of connectives allowed by the different languages. Specifically, in *ehhf* while providing multi-headed clauses we restrict goal formulas to have occurrences of synchronous connectives only. To explain the reason, let us consider the connective $\otimes$. Occurrences of $\otimes$ in goal formulas are used in linear logic programming languages like Lolli to manipulate the resources which are stored in the left hand-side of sequents. In our approach, we use a dual representation in which the resources are stored in the right-hand side of sequents (as in some of the examples in [26] and in previous approaches like [2, 20]). To handle them, we employ multi-headed clauses: when applied they determine a nondeterministic splitting of the resources in the right-hand side of sequents. We have motivated our choices through several examples of executable *ehhf*- and *ehhf*$_D$-specifications of state-based systems. We leave as future work a possible extension of *ehhf* with occurrences of $\otimes$ in goal formulas. This might require more complex proofs for the completeness of higher-order logic allowing quantification over formulas like in *ehhf*$_D$.

A preliminary version of this work appeared in [10] where we defined a subset of the language *ehhf*, we called F &O, including a backchaining rule for multi-headed clauses. In [10, 11], we applied these ideas to model aspects of object-oriented programming. In [5, 6], Bugliesi et al. investigated in the foundation of an object calculus based on a fragment of Forum consisting of Horn Clauses enriched with the linear implication in goal formulas. In our paper, we followed the ideas in [6] to prove the completeness result for the richer fragment *ehhf*$_D$.

Acknowledgements

The authors would like to thank Dale Miller, Didier Galmiche, Michele Bugliesi, and Marco Bozzano for many useful conversations and suggestions.

References

[1] J.M. Andreoli, Logic Programming with focusing proofs in linear logic, J. Logic Comput. 2 (3) (1992) 297–347.

[2] J.M. Andreoli, R. Pareschi, Linear objects: logical processes with built-in inheritance, New Generation Comput. 9 (1991) 445–473.

[3] M. Bozzano, G. Delzanno, M. Martelli, A linear logic specification of chimera, Proc. DYNAMICS'97, a satellite workshop of ILP's '97, 1997.

[4] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, F. Zini, Logic programming & multi-agent systems: a synergic combination for applications and semantics, in: K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren (Eds.), The Logic Programming Paradigm: a 25-Year Perspective, Series: Artificial Intelligence, Springer, Berlin, 1999, pp. 5–32.

[5] M. Bugliesi, G. Delzanno, L. Liquori, M. Martelli, A linear logic calculus of objects, In: Proc. Joint Internat. Conf. Symp. on Logic Programming MIT Press, Cambridge, MA, 1996, pp. 67–81.

[6] M. Bugliesi, G. Delzanno, L. Liquori, M. Martelli, Object calculi in linear logic, J. Logic Comput. 10 (2000).

[7] J. Chirimar, Proof theoretic approach to specification languages, Ph.D. Thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[8] G. Delzanno, Logic & object-oriented programming in linear logic, Ph.D. Thesis, Università of Pisa, Dipartimento di Informatica, March 1997.

[9] G. Delzanno, Specification of term rewriting in linear logic, in: D. Galmiche (Ed.), Proc. Workshop on Proof-Search in Type-Theoretic, Vol. 17 of Electronic Notes in Theoretical Computer Science, Lindau, Germany, July 5, 1998, Elsevier, Amsterdam, available at the URL www.elsevier.nl/locate/tcs.

[10] G. Delzanno, M. Martelli, Objects in forum, in: Proc. Internat. Logic Programming Symposium, MIT Press, Cambridge, MA, 1995, pp. 115–129.

[11] G. Delzanno, D. Galmiche, M. Martelli, A specification logic for concurrent object-oriented programming, Math. Struct. Comput. Sci. 9 (3) (1999) 253–286.
[12] J.H. Gallier, Logic for Computer Science, Harper & Row, New York, 1986.
[13] D. Galmiche, E. Boudinet, Proofs, concurrent objects and computations in a FILL framework, in: Proc. Workshop on Object-Based Parallel and Distributed Computation (OBPD 95), Lecture Notes in Computer Science, Vol. 1107, 1995, 148–167.
[14] D. Galmiche, G. Perrier, On proof normalization in linear logic, Theoret. Comput. Sci. 135 (1) (1994) 67–110.
[15] J.Y. Girard, Linear logic, Theoret. Comput. Sci. 50 (1987) 1–102.
[16] A. Guglielmi, Abstract logic programming in linear logic-independence and causality in a first order calculus, Ph.D. Thesis, Department of Computer Science, University of Pisa, 1995.
[17] J.A. Harland, D. Pym, M. Winikoff, Programming in Lygon: an overview, in: M. Wirsing, M. Nivat (Eds.), Algebraic Methodology and Software Technology, Lecture Notes in Computer Science, Vol. 1101, July 1996, Springer, Munich, Germany, pp. 391–405.
[18] J. Hodas, D. Miller, Logic programming in a fragment of intuitionistic linear logic, Inform. and Comput. 110 (2) (1994) 327–365.
[19] J.S. Hodas, J. Polakow, Forum as a logic programming language, in: J.-Y. Girard, M. Okada, A. Scedrov (Eds.), Linear Logic 96 Tokyo Meeting, Vol. 3 of Electronic Notes in Theoretical Computer Science, Mita Campus, Keio University, Tokyo, Japan, March 28–April 2, 1996.
[20] N. Kobayashi, A. Yonezawa, Asynchronous communication model based on linear logic, Formal Aspects of Comput. 7 (1995) 113–149.
[21] P. Lincoln, V. Saraswat, Higher-order, linear, concurrent constraint programming, Manuscript, January 1993.
[22] P. López, E. Pimentel, A lazy splitting system for forum, in: M. Falaschi, M. Navarro, A. Policriti (Eds.), Proc. APPIA-GULP-PRODE 97 Joint Conf. on Declarative Programming, Grado, Italy, June, 1997, pp. 247–258.
[23] D. Miller, The π-calculus as a theory in linear logic: preliminary results, in: E. Lamma, P. Mello (Eds.), Proc. 1992 Workshop on Extension to Logic Programming, Lecture Notes in Computer Science, Vol. 660, Springer, Berlin, 1993, pp. 242–265.
[24] D. Miller, A multiple-conclusion meta-logic, Proc. 1994 Symp. on Logics in Computer Science, 1994, pp. 272–281.
[25] D. Miller, Forum: a multiple-conclusion specification logic, Theoret. Comput. Sci. 165 (1) (1996) 201–232.
[26] D. Miller, G. Nadathur, F. Pfenning, A. Scedrov, Uniform proofs as a foundation for logic programming, Ann. Pure Appl. Logic 51 (1991) 125–157.
[27] G. Nadathur, D. Miller, An overview of λ-Prolog, Fifth Internat. Symp. on Logic Programming (1988).
[28] G. Perrier, A model of concurrency based on linear logic, Proc. Conf. on Computer Science Logic 95 (1995).
[29] D.J. Pym, J.A. Harland, A uniform proof-theoretical investigation of linear logic programming, J. Logic Comput. 4 (2) (1994) 175–207.
[30] M.H. van Emden, R.A. Kowalski, The semantics of predicate logic as a programming language, J. ACM 1976 (23) 733–742.