A Linear Logic Specification for Chimera

Marco Bozzano, Giorgio Delzanno and Maurizio Martelli

Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova,
Via Dodecaneso, 35,
Genova, I-16146, Italy
giorgio@disi.unige.it

Abstract. Forum [36], a powerful logic formalism based on Higher Order Linear Logic, is particularly suited to specify and reason about complex programs and systems. $\mathcal{E}_{hhf}$[12], a subset of Forum, models many interesting logic programming extensions towards O.O. and concurrent systems and can be viewed as a very high level logic programming specification language. The paper presents some results in this direction, namely the specification in $\mathcal{E}_{hhf}$ of Chimera, an Active, Object-Oriented and Deductive Database System.

Keywords: Linear Logic, Object-Oriented and Deductive Databases.

1 Introduction

Proof theory and automated deduction have provided relevant contributions to computer science, in particular in the fields of high-level programming languages and formal verification of software.

Many different logics have been proposed and used for these purposes. We will work with Linear logic [22] with the aim to use it as a theoretical foundation for modern and powerful specification languages. Specific features have been separately studied in previous works: object-orientation and concurrency in LO [3, 2] and ACL [29], deductive databases in [27], logic programming in [25], communicating processes in [35], imperative programming paradigms in [11, 38], and transition systems in [42]. The importance of such works is not limited to the development of new high-level languages. In fact, they can help in the analysis of declarative programs properties and in the verification of their correctness. Experimental results in this field have been obtained in the study of logical equivalence of concurrent processes modeled by Linear Logic [35, 30]. Further results are expected in the application of these techniques to logical characterizations of objects and agents.

In our current work [12] we are looking for a uniform representation of all above mentioned aspects within a unique framework. We have developed a formalism, called $\mathcal{E}_{hhf}$ [12], based on a subclass of proofs, sequents and formulas of Forum [38]. The interpretation of these restricted formalism is much closer to the real computational aspects present in the new high-level languages we are interested in.

Forum is a formulation of Higher-Order Linear Logic which has the very nice property of providing uniform proofs only. This class of proofs fully characterizes those systems which can be considered as Abstract Logic Programming languages (ALPL) [39]. Here, based on a sequent-calculus notation, two-sided sequents nicely capture the evolution of state transitions and the proof construction corresponds to the operational computation.

The paper presents a particular application of $\mathcal{E}_{hhf}$ to the specification of a deductive object-oriented database system called Chimera [10, 9, 23], which is an interesting example of integration among different programming paradigms. This is an interesting example to verify the power and the flexibility of the considered framework.

In the next sections we shall describe the data model of Chimera [10, 9, 23], developed as part of the ESPRIT Project Idea [8], and we shall give a logical semantics to it by defining an encoding in $\mathcal{E}_{hhf}$. Chimera has been chosen as a representative of modern database management systems (DBMS). These incorporate, for instance, objects modeling capabilities into deductive databases [5], triggers and constraints into object-oriented DBMS (OODBMS) [19, 20], or enhance SQL with object-oriented capabilities [16].

The next step of this research will be to precisely define which are the relations between Linear Logic and other formalisms developed to assign a semantics to transactions and updates, e.g. [6, 41, 28]. See also the final section for a brief comparison between our approach and some of the above mentioned ones.

In this work we have applied techniques already present in our previous studies on object-orientation and concurrency [13, 14]. These efforts allowed us to enrich our $\mathcal{E}_{hhf}$-based specification system, implemented using an experimental higher-order logic programming language called λProlog[1], with semantically well-founded primitives for writing object-oriented and database-oriented specifications.

Structure of the paper: In section 2 we shall introduce the proof as computation perspective in Linear Logic. In section 3 we shall give a brief overview of our language called $\mathcal{E}_{hhf}$, and show how to write a specification in such a setting. In section 4 we shall describe the main features of Chimera, and, finally, in section 5 we shall give a semantics for Chimera. We conclude the paper with some final remarks and comparisons with other works.

2 A Brief Overview of Linear Logic

Linear Logic (LL), introduced by Girard [22], is a logic of occurrences, i.e., a refinement of Classical Logic (CL) in which formulas can be consumed during the deduction process. This can be obtained eliminating the structural rules for Classical sequent calculus which allow one does to arbitrarily duplicate and reduce formulas on both sides of sequents. The resulting logical framework and in particular its proof-theoretical formalization revealed strong connections with many computational models such as the Petri nets [15, 21] and the π-calculus [4]. Proof theoretic investigations yielded both Intuitionistic and Classical formulations of LL [43] with interesting results on *cut-elimination* for sublogics based on fragments of connectives and on permutability of their rules [2, 18, 31].

For people not acquainted with these subject, in the following we shall describe the main features of LL under a proof-theoretical perspective. Some preliminary notions on sequent-calculus, λ-calculus, and CL are necessary for a good comprehension of the paper.

2.1 A Logic of Occurrences

Sequent calculus [17] has been introduced to derive presentations of logic formalisms, e.g., CL, suitable to develop automated theorem proving methodologies. Sequents are intuitively meta-representations of proofs, i.e., of the premises $P_1 \wedge \ldots \wedge P_m$ and of the thesis $T_1 \vee \ldots \vee T_n$, written as $P_1, \ldots, P_m \rightarrow T_1, \ldots, T_n$.

The rules captures the semantics of the connectives ($\wedge$, $\vee$, $\supset$, etc. in CL); they are divided in left-introduction and right-introduction rules which allow one to deal with their occurrences on both sides of sequents. To derive a theorem T from the set of hypothesis Γ, it is necessary to build a proof of the sequent $\Gamma \rightarrow T$ whose leaves are axioms such as $\Gamma, A \rightarrow A$ of CL (CL). This bottom up construction can be read as a simplification of the formulas in the initial sequent into simpler ones in order to reduce them to an instance of an axiom. Thus the rules for the connectives can be seen as operational rules guiding the proof search process.

As said before LL is obtained by CL eliminating the rules of contraction (e.g., $\frac{\Gamma \rightarrow \Delta, A, A}{\Gamma \rightarrow \Delta, A}$ concerning the right-hand side) and weakening (e.g., $\frac{\Gamma \rightarrow \Delta, A}{\Gamma \rightarrow \Delta}$ for the right-hand side). According to our previous interpretation, the former allows one to arbitrarily duplicate formulas in a proof, whereas the latter allows one to weaken the premises by introducing new formulas. This slight modification has a dramatic impact on a number of classical logical properties. In particular the additive and multiplicative formulations of the proof system (see [17]) are no more equivalent. This produces a sort of splitting of each classical connective into two new ones: an additive version and a multiplicative version, i.e., contexts are respectively copied and non-deterministically split when the formula with that connective is introduced. Thus, $\wedge$ is split into & (with) and $\otimes$ (tensor); $\vee$ is split into $\oplus$ (plus) and $\parr$ (par); $\supset$ into $\rightsquigarrow$ and $\multimap$ (lollipop); *true* into $\top$

[1] λProlog [39] enriches Prolog with λterms and intuitionistic logic connectives like implication and universal quantification in the goals.

(all), **1**; $false$ into **0**, $\bot$ (anti). Two particular modalities, namely ! and ?, are introduced to recover the expressiveness of CL (i.e., weakening and contraction can be applied over !- and ?-formulas).

In appendix A we have depicted the sequent proof system for Propositional LL. In this formulation sequents consist of *lists* or multisets of formulas since they cannot arbitrarily be duplicated unless they are prefixed by the modality ! (on the left) and ? (on the right) and thus each occurrence of the same formula does count in a derivation. [2]

According to recent works in the field of Logic Programming [39] and LL Programming [26, 36, 24] it is possible to interpret sequents as descriptions of instant configurations of a computation whose complete behavior is captured by a proof of the sequent itself. Under this perspective the left-hand side of a sequent is viewed as a set of program clauses (by choosing only particular classes of formulas such as Horn-Clauses in Intuitionistic Logic) while the right-hand side as the current goal to prove.

In this setting the rules for the connectives assume a more refined operational meaning which allows one to use them as programming constructs inside an ideal interpreter for the considered logic language. This metaphor, *proofs as computations*, has been investigated from many different point-of-views in LL as shown by the works in [3, 2, 24, 27, 29, 18]. The reader can refer to [1, 37] to have an extensive bibliography and survey on the topic. We will focus on a particular presentation of LL proof theory based on previous work on Logic Programming and non-standard logics [39].

2.2 Forum: Higher Order Linear Logic

Forum [36] can be considered as the point of convergence of Miller's previous works, for instance [39, 27, 34] on both Higher-Order and LL languages. This language is the result of the analysis of LL in terms of an extended notion of uniformity coping with multi-conclusion sequents. Forum is a *specification* language in which it is possible to define the semantics of many aspects of Programming Languages. In fact, among the motivations behind its introduction it is possible to find the need of powerful means to capture updates [26] and concurrency [34] in a purely logical setting.

The language is based on a fragment of LL with the following set of connectives: $\multimap$, $\Rightarrow$, &, $\parr$, $\bot$, $\top$, and the universal quantification $\forall$. For the sake of simplicity, we will omit the type in the quantification if it is clear from the context. The intuitionistic implication is not a proper linear connective but it is defined by the following logical equivalence (i.e. provable in linear logic) $(A \Rightarrow B) \equiv (!A \multimap B)$.

Uniformity of proofs is encoded in the proof system, since the left introduction rules are applicable if and only if the right-hand side of a sequent consists of atomic formulas. According to this definition of uniformity, Forum can be considered an ALPL formulation of LL.

LL sequents can be written by partitioning bounded and unbounded formulas (i.e., prefixed by !), in Forum a further component, a signature Σ, is added in order to keep track of the types of the constants. Thus, Forum sequents have the form $\Sigma : \Gamma; \Delta \rightarrow \Omega$ where Σ contains all the typed constants used in the formula in Γ (a set) , Δ and Ω (two multisets) over the above described set of linear connectives. Its interpretation in LL is $!\Gamma, \Delta \longrightarrow \Omega$. Two examples of Forum rules are:

$$\frac{\Sigma : \Gamma; A, \Delta_1 \rightarrow \Upsilon_1 \quad \Sigma : \Gamma; B, \Delta_2 \rightarrow \Upsilon_2}{\Sigma : \Gamma; A \parr B, \Delta_1, \Delta_2 \rightarrow \Upsilon_1 \uplus \Upsilon_2}\ \parr_l \qquad \frac{\Sigma : \Gamma; \Delta \rightarrow \Upsilon, A, B, \Gamma}{\Sigma : \Gamma; \Delta \rightarrow \Upsilon, A \parr B, \Gamma}\ \parr_r$$

Here the idea is that left rules can be applied if and only if the right hand side (i.e., $\Upsilon_1 \uplus \Upsilon_2$) is a set of atomic formulas, thus to prove a sequent it is necessary to first reduce its right-hand side, then to apply the left-rules, and so on (as prescribed by the notion of *uniformity*). Non-determinism in proof-search is due to different possible choices in context splitting as for Υ_1 and Υ_2 in the above depicted rule.

In [38] Miller has proved the equivalence of Forum with LinLog [2] and thus with full Higher-Order Linear Logic. The previous cited result of completeness between Forum and higher-order LL suggests to consider Forum as a proof theoretical foundation for Logic programming languages defined over particular classes of formulas, as in the case of First Order Logic for Horn Clauses (Prolog [44]), and Intuitionistic Logic for hereditary Harrop formulas (λProlog [40]). In our work [12] we have tried to isolate a particular

[2] Operations over multisets will be denoted by: $\uplus$ for the union and $\setminus$ for the difference.

subset of Forum enjoying the good properties that the subset of higher-order hereditary Harrop formulas does [39]. We called the considered class of formulas $\mathcal{E}_{hhf}$ standing for "extended hereditary Harrop formulas". Restricting the class of formulas (providing formulas corresponding to program clauses and to goal formulas as in standard logic programming) allows one to consider a subclass of proofs which enjoys good properties from an operational perspective. For instance, in the selected language we restrict non-determinism to clause selection and instantiation (unification) as in classical logic programming. Although this could seem a strong restriction, to be in a multi-conclusion setting (and thus enjoying multi set rewriting) provides with a number of different interpretations of sequents and proofs, suitable to encode many different kinds of applications.

3 Executable Specifications in Linear Logic

To avoid the high level of non-determinism of proof searching in Forum it is necessary to consider subclasses of Forum formulas closer to a programming perspective. One of these classes, namely $\mathcal{E}_{hhf}$ defined in [12], revealed enough expressiveness to be used as foundation of a system for executing logical specifications. In the following we will introduce this formalism.

SEQUENT-CALCULUS. As previously mentioned we are interested in considering higher-order syntax for Linear Logic-based languages. One way to achieve this is to consider a universe of simply typed λ-terms instead of the usual presentation of terms and formulas. Under this perspective LL formulas correspond to λ-terms having a fixed *boolean* type o. We will use signatures, i.e. sets of pairs $c : \tau$, to denote the type τ of a constant symbol c. The formulas considered in the language $\mathcal{E}_{hhf}$ fit into two classes: *goals* and *clauses*.

In the following we will introduce the language considering a slight variation of Forum sequents, namely $\Gamma; \Delta \longrightarrow_\Sigma \Omega \parallel \Theta$. Here Γ is as a set of clauses, Δ is a multi-set of *clauses*; Ω is a multi-set of *goals*, and Θ is a multi-set of *atomic* formulas contained in $\Pi_{\Sigma_\mathcal{R}}$, i.e., the set of atoms $(f\ t_1\ \ldots\ t_n)$ s.t. $f \in \Sigma_\mathcal{R}$, a signature, fixed a priori, of *state constructors*. According to our previous discussion on the interpretation of proofs as computations and of sequents as instant configurations, we can consider Ω as a collection of *concurrently executing processes* and Θ as a collection of *information about the state of the configuration*. Γ will define the operations to be executed and Δ will contain operations that must be executed a limited number of times. This view will help in explaining the features of $\mathcal{E}_{hhf}$ and in particular the meaning of the rules depicted in Fig. 5 in Appendix B, obtained by Forum ones by applying permutability properties.

GOALS. The notion of *goal* typical of standard LP is extended here in order to provide primitives for concurrency and other complex operations. A goal G can be one of the following formulas:

- $G_1 \& G_2$: this formula can be used to split a proof into two branches (one for G_1 and the other for G_2) maintaining the same bounded context (see rule $\&_r$ in Fig. 5 in Appendix B). As a consequence if one branch represents a thread of computation we could use the other one to test conditions over the current state. We will come back to this point in the example at the end of this section.
- $G_1 \parr G_2$: in accord with the chosen interpretation of proofs, the rule $\parr_r$ suggests to consider this connective as a primitive for concurrency, i.e., G_1 and G_2 will be executed concurrently.
- $\forall x.G$: the side condition of rule $\forall_r$ suggests to use this connective to introduce a notion of data abstraction over components of the current configuration see [33];
- $D \multimap G$: the first type of implication, i.e., the linear implication, allows one to enrich the current multiset of bounded-use clauses with a new one, see rule $\multimap_r$. The newly introduced resource must be consumed during the rest of the proof (it will be apply in some backchaining step). Here we require that D is a clause at the moment of the execution of such a goal.
- $D \Rightarrow G$: the intuitionistic implication (rule $\Rightarrow_r$) allows one to enrich the current program with a new clause (introducing modularity in the logical setting [32]). This modification will persist during the rest of the computation. Again we require that D is a clause at the moment of the execution of such a goal.

– an atomic formula A: if A is an atomic formula in $\Pi_{\Sigma_\mathcal{R}}$ then it is moved into Θ since it is a representation of a state component; otherwise it corresponds to the elementary representation of a *process* or *operation* whose meaning will be defined by the clauses in the current program and in the current bounded context.
– $\bot$: it simply disappears from the current sequent; $\top$: stops a computation.

CLAUSES. They are &-conjunctions of formulas having the following form: $\forall((G_1 \& \ldots \& G_n) \Rightarrow D)$. Here D can be either $(A_1 \parr \ldots \parr A_n) \circ\!\!- G$ or simply $(A_1 \parr \ldots \parr A_n)$, where A_i is an atomic formula (at least one must not be in $\Pi_{\Sigma_\mathcal{R}}$), whereas G and G_i are goal formulas for $i : 1, \ldots, n$. Such clauses are an extension of the notion of definite clause of standard LP in that they enjoy multi-heads (useful to simulate multi-set rewriting) and more powerful connectives in the body G (a goal as described above).

Intuitively, a clause having form $C \Rightarrow (H \circ\!\!- G)$ can be read as *if C holds then the multiset H (i.e., consisting of the atoms occurring in H) in the current goal can be rewritten into G.* This has been formalized by the backchaining rule (bc) in Fig. 5 of Appendix B, which deals with linear clauses and by the so called guard rule (gr) which deals with intuitionistic clauses. As explained in Fig. 5, in bc step a clause D is selected either from Γ (the unbounded context) or Δ (the bounded one) and an instance of one conjoined formula in D is applied as a rewriting step.
The selection of one of such instances induces a non-determinism in the proof-search process which can be managed by introducing *backtracking* (i.e., the possibility to roll back to such choice points if the last choice has not yielded a successful proof).
Backchaining over clauses having form $A_1 \parr \ldots \parr A_n$ corresponds to an application of a final step of the rewriting step which consists in a sort of matching of the component-atoms A_i against the current right-hand side of the sequent as shown by rule *initial*.
The absorb rule allow one to use an unlimited number of times the clauses in Γ in *bc, initial* and *gr* steps.
A further type of formulas (called $\mathcal{R}$-formulas) can occur in Γ. They have the form: $\forall S_1 \& \ldots \& S_n$ where S_i is a $\parr$-disjunction of atomic $\Pi_{\Sigma_\mathcal{R}}$-formulas. They are used to observe the final state of a computation as shown by the rule *verify*. Here Φ *matches* Θ stands for $\Phi \to \Theta$ is cut-free provable (i.e., the proof is a simple sequence of $\forall_l$, $\parr_l$, $\&_l$ applications). Since Φ can be guessed in the initial sequent, in the sequel we will also use the notation $\Gamma[\Phi]; \Delta \longrightarrow_\Sigma \Omega \parallel \Theta$.

AN EXAMPLE. Let us consider the following informal specification: *We want to specify the behavior of a set of counters which can be modified randomly by using an operation (inc Z). Furthermore, we want to specify an operation to create a counter if and only if there are no other counters already active.*

We can pass to a formal specification in $\mathcal{E}_{hhf}$ as described below. Let Σ be a signature with the symbols $new, inc, howmany, no\_counters, one\_new$ with adequate types and s.t. $(countert) \in \Pi_{\Sigma_\mathcal{R}}$, for each t. The following clauses implement the first sentence of the specification and they will be included in Γ:

$$new \circ\!\!- counter\ 0.$$
$$add\ X\ Z\ Y \Rightarrow (inc\ Z \parr counter\ X \circ\!\!- counter\ Y).$$

the former simply rewrites the atomic goal *new* into the a new *counter* initialized to 0 (there is no guard in this clause), whereas the latter *synchronize* a *inc*-message and a counter consuming and rewriting them into an updated counter (the guard simply executes the increment). The *add* predicate is just defined by a Prolog-like predicate which

$$add\ X\ 0\ X.$$
$$add\ (s\ X)\ Y\ (s\ Z) \Leftarrow add\ X\ Y\ Z.$$

In the following we will use $0, 1, \ldots$ for $0, (s\ 0), \ldots$. The second sentence can be specified by using an auxiliary predicate *howmany* which returns the number of existing counters occurring in Θ:

$$add\ N\ 1\ Z \Rightarrow (howmany\ N\ M \parr counterX \circ\!\!- howmany\ Z\ M).$$
$$howmany\ N\ N.$$

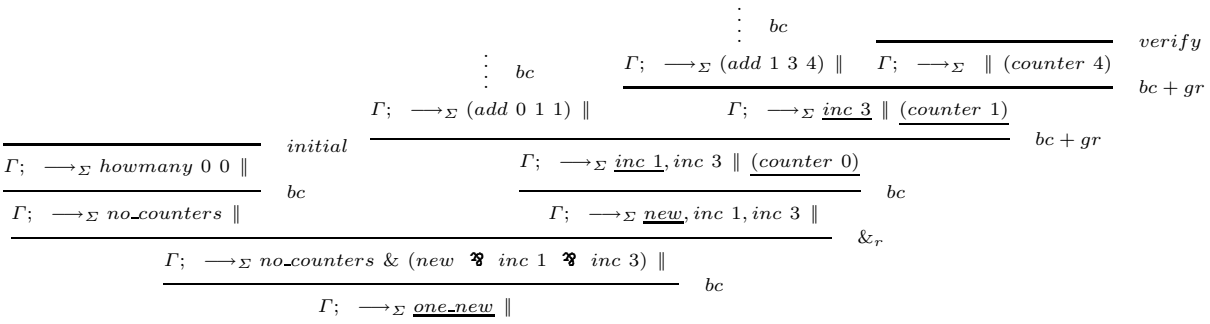

Fig. 1. A sample derivation: V is binded to (*counter* 4).

This program simply consumes the counter-resources in Θ until no counter exists, returning the value of an accumulator (the first parameter of *howmany*). Notice that the side condition of the *initial* ensures that the base case will be applied if and only if the remaining contexts are empty. Thus, the required operation can be specified executing the test *howmany* 0 0 over the current state by using &:

no_counters $\circ\!-$ *howmany* 0 0.
one_new $\circ\!-$ *no_counters* & (*new* ⅋ *inc* 1 ⅋ *inc* 3).

In Figure 1 we have depicted an example of sample proof applying the rules of Fig. 5. Notice that the two *inc* operations could be executed in any order, since the Ω component of a sequent is a multi-set: in this case their execution yields always the same final state. A backtracking mechanism should be considered in an effective implementation in order to consider all the different choices of clauses in a bc-step.

The main thread of the computation corresponds to the right branch which models the evolution of the state of the counter, whereas, the left branches are used here to test the conditional parts of the rules. For instance, In the rule $\&_r$ of the figure the condition *no_counter* is executed with a copy of the state (empty in this case). This ensures that *one_new* is executed if and only if there are no other active *counters*.

A final observation about the structure of the proof: the left branches are terminated by *initial* axioms matching facts in Γ with atomic goals in Ω; the right branch is terminated by an instance of the *verify* axiom which matches a particular formula consisting only of $\Pi_{\Sigma_\mathcal{R}}$ predicates with the final state of the computation. This formula should be included in Γ, however, it can be reconstructed by observing the final state, and it can be used to test if the given program correctly satisfies the specification (e.g., we would like to test if just one counter has been created without specifying its final value by including $\forall x.(counter\ x)$ in Γ). According to our previous notation we have then proved the sequent $\Gamma, \forall x.(counter\ x);\ \longrightarrow_\Sigma one\_new \parallel \emptyset$.

As previously mentioned, in the above sketched scheme the non-determinism is due to the choice of the clause to apply (which induces a choice on the multiset of atoms in the right-hand side that must be rewritten), and, in the effective implementation, to higher-order unification. In this sense the chosen class of formulas can be considered as an extension of hereditary Harrop formulas (λ-Prolog) with the possibility to handle resources and concurrent actions.

In section 5, we shall see how this interpretation of linear proofs can be successfully exploited to give an operational semantics for deductive database languages. For this purpose we will study the data model Chimera since it enjoys many interesting features of database languages as we will describe in the following section.

4 An Overview of Chimera

The data model of Idea, called Chimera [9, 10, 23], is an object-oriented, deductive, active data model in that: it provides all concepts commonly ascribed to object-oriented data models (such as object identity, complex objects and user defined operations, classes, inheritance); it provides capabilities for defining deductive rules (used to define views and integrity constraints, to formulate queries, to specify methods, to compute derived information); finally, it supports a powerful language for defining triggers. In this section we shall briefly describe the main features of Chimera.

OBJECTS AND VALUES. Chimera is a *class based* object oriented language for DBMS with *multiple inheritance*. *Objects* are abstractions of real world entities, and are distinguished from each other by means of unique *object identifiers* (OIDs). The *state* of an object can be viewed as a collection of attributes (i.e., functions mapping an object to a uniquely defined value). Objects can be manipulated by means of *operations* (or *methods*), which are *guarded* procedures. An operation is defined by means of passive rules of the form `Head:Guard -> Body`. Objects are divided into *persistent* and *temporary* ones, depending on whether they survive a database session or not.

CLASSES. The notion of *class* emphasizes the membership of objects in a common set of instances; classes are defined, populated, and deleted explicitly. *Class attributes* are functions mapping an entire class to a unique value. *Class operations* manipulate an entire class rather than individual instances (they can, for example, manipulate class attributes). Object classes may be recursively specialized into subclasses, resulting in a taxonomic hierarchy of arbitrary depth. A subclass inherits all attributes and operations from its super-classes, but it may redefine their implementation and add new ones.

Example 4.1 The real world entities 'person' and 'employee' could be modeled in Chimera by the following definitions:

```
define object class person
  attributes     name:string(20)
                 income:integer
                 profession:string(10)
  operations     changeIncome(in Amount:integer)
  c-attributes   lifeExpectancy:integer
  c-operations   changeLifeExpectancy(in Delta:integer,
                   out NewValue:integer)
end
```

```
define object class employee
  super-classes person
  attributes     emplNr:integer
                 mgr:employee
                 salary:integer
  c-attributes   maximumSalary:integer
end
```

The code for the method *changeIncome* of the class *person*, which is assumed to raise a person's income, can be

```
changeIncome(Amount):
  integer(New),New=Self.income+Amount ->
  modify(person.income,Self,New)
```

where **modify**$(clname.attr, id, v)$ is the primitive for attribute modification, i.e. changes the value of the attribute *attr* of the object identified by *id* (belonging to the class *clname*) to v. □

TRIGGERS. *Triggers* or *active rules* are a means for introducing specific reactions to particular events relevant to the database. Such events include database specific operations (queries and updates), operation calls and constraint violations. The execution of reactions takes place depending on the evaluation of particular conditions on the database state, expressed in trigger definitions; further, execution order may be guided by priorities assigned to triggers. When one of the events of an active rule occurs, we say that the active rule is *triggered*. At given points of time, *active rule processing* is started: every triggered rule in considered and, if the condition associated with it is satisfied, the reaction is executed. Triggers can be defined as either *immediate* or *deferred*, depending on whether they are executed as soon as they are triggered or at the end of the user transaction that has caused the triggering.

Example 4.2 An example of trigger might be the following:

```
define trigger adjustSalary for employee
  events     create
             modify(salary)
  condition  Self.salary> Self.mgr.salary
  actions    modify(employee.salary, Self, Self.mgr.salary)
end
```

This trigger is activated by the creation or salary modification of an object of the class 'employee', and checks to see if that object's salary exceeds the respective manager's one; in this case the object's salary is lowered. □

QUERIES AND UPDATES. A *query* in Chimera consists of a formula F and a target list T. The formula F is evaluated over the current state of the database, returning bindings for the variables in T. *Updates* in Chimera support object creation (primitive **create**) and deletion (**delete**), object migration from one class to another (**specialize**, **generalize**), state change (**modify**) or change of persistence status of objects (**make_persistent**). Updates can also be achieved through methods, because a method can change the state of an object. Finally, Chimera supports a conventional notion of *transaction*, with user-controlled *commit* and *rollback* primitives.

Example 4.3 Typical queries in Chimera look like

```
select(X.name where employee(X), X.salary < 5000)
select(X.name,Y.brand where person(X), car(Y), Y.owner=X).
```

□

5 A Linear Logic Semantics for Chimera

In this section we will show how the various features of Chimera can be modeled using the language $\mathcal{E}_{hhf}$. We shall not deal explicitly with deductive rules, as, of course, this kind of rules are freely available in the context of a logical sequent calculus. All the details of the presented work can be found in [7]. In the following, we shall use the notation $\forall F$ to indicate the universal closure of a formula F.

We have chosen to model the *concurrent* execution of multiple user transactions, each of which has a sequential code. Therefore, the right hand side of sequents will contain an encoding of transactions together with a representation of the database state, while the left hand side will contain the unbounded resources, i.e. clauses, and the bounded ones, i.e. methods (as we shall see, a method must be activated before it can be used). As mentioned in section 3, the formulas we shall use in the following paragraphs for writing clauses have the general form

$$(C_1 \ \& \ \ldots \ \& \ C_q) \Rightarrow (A_1 \parr \ldots \parr A_n \circ\!\!- B_1 \parr \ldots \parr B_m);$$

the informal meaning of such a formula is that, provided the conditions $C_1 \ldots C_q$ hold, the objects $A_1 \ldots A_n$ can be rewritten into $B_1 \ldots B_m$.

REPRESENTING COMPUTATIONS. We can think of a computation in a DBMS as the *parallel* execution of some user *transactions* submitted to the system. A transaction can be modeled by the term **trans**(*code*), where *code* (the transaction's code) can be represented as a sequence of statements using the non-logical symbol $''{;}''$ having type $o \to o \to o$ (this technique is known as *continuation passing*). Hence, sequents will assume the following form:

$$\Gamma; \Delta[\Phi] \to_\Sigma \mathbf{trans}(code_1), \ldots, \mathbf{trans}(code_n) \parallel \Theta,$$

where, as said before, Θ is a representation of the database state, i.e. a collection of objects, while $code_i$ can be a sequence like $op_1; \ldots; op_n; \bot$. Γ will contain all the unbounded resources (clauses), and Δ, as we shall see, the code for the activated methods.

An example of semantics is given by the clause for managing transaction termination: $\forall$ **trans**$(\bot) \circ\!\!- \bot$. This clause is used whenever the code of a transaction becomes empty ($\bot$), i.e. when all instructions have been executed, and simply destroys the transaction itself.

OBJECT AND CLASS REPRESENTATION. Let's now see how a hierarchy of classes can be modeled in our system (for the sake of simplicity, we shall limit ourselves to the case of single inheritance). We can represent the information on classes by clauses (in the unbounded context of a sequent) of the form

$$class(clname, attrs, meths, supclname),$$

where *attrs* and *meths* represent the attributes and methods of the class *clname*, while *supclname* is the superclass. The root class will be indicated as *topclass*.

The database state will be represented by means of a collection of object terms (appearing in the state part of a sequent) of the form

$$\mathbf{object}(clname, id, state),$$

where *id* is the *object identifier*, *clname* is the class to which the object belongs, and *state* is a record which associates the attribute names with the respective values. In the following, we shall use for records the syntax $\langle field_1 : v_1, \ldots, field_n : v_n \rangle$.

Example 5.1 We can represent the information on the classes *person* and *employee* (see example 4.1) by the following clauses (the encoding of methods, given by the terms *meths* and *meths*$'$, will be discussed in the following paragraph):

$class(person, [name, income, profession], meths, topclass).$
$class(employee, [name, income, profession, emplNr, mgr, salary], meths', person).$

An object of type *person* could be, for instance, a term such as

$\mathbf{object}(person, id, \langle name : Smith, income : 10000, profession : employee \rangle).$

□

METHOD REPRESENTATION. The set of methods of a class can be modeled by a term like

$$\lambda \text{ID}.\ \text{METH}_1\ \&\ \ldots\ \&\ \text{METH}_n,$$

where the abstraction on ID, as we shall see, is used to define the semantics of method invocation (it lets a method code be parametric over the identifier of an object). A user transaction can invoke the execution of a method sending an appropriate message to an object, with a statement such as **send**$(id, message)$, where *message* specifies the name of the method and possibly some parameters. We can define the semantics of **send** by means of the following clause:

$\forall$ *class*(CLNAME, ATTRS, METHS, SUPCLNAME) $\Rightarrow$
  trans(**send**(ID, MESSAGE); R) $\otimes$ **object**(CLNAME, ID, STATE) $\circ\!\!-$
    ((METHS ID) $\multimap$ **trans**(**call**(ID, MESSAGE); R) $\otimes$ **frozen**(CLNAME, ID, STATE)).

The semantics of method invocation and execution is achieved by means of two different steps. In the first step, given by the above clause, the **send** primitive is turned into a **call** primitive and the method code is activated in the *bounded* context using the linear implication (this implies that the code will be removed from the context after its use). In the second step, the method is effectively executed applying the code activated in the previous step. In the above clause, a mechanism of *object freezing* is used in order to prevent an object from responding to more than one method invocation at the same time (this can be problematic because an object, during its life, can migrate along the class hierarchy). You should also note that the method code is applied to the object identifier before being activated. Actually, the activation involves the code for all the methods of a class, but only the right one will be used (the right choice is allowed by the & conjunction, which, at the interpreter level, is managed through backtracking over the conjuncts). The predicate *class*, in this case, is used to get the method code of a class; the use

$$\dfrac{\dfrac{\dfrac{\Gamma;\Delta[\Phi] \rightarrow_\Sigma \mathbf{trans}(r';r),\Omega \parallel \mathbf{object}(c,id,s'),\Theta}{\Gamma;(meths\ id),\Delta[\Phi] \rightarrow_\Sigma \mathbf{trans}(\mathbf{call}(id,m);r),\Omega \parallel \mathbf{frozen}(c,id,s),\Theta}\ bc+\parr_r}{\mathsf{AUX}\quad \Gamma;\Delta[\Phi] \rightarrow_\Sigma (meths\ id) \multimap (\mathbf{trans}(\mathbf{call}(id,m);r) \parr \mathbf{frozen}(c,id,s)),\Omega \parallel \Theta}\ \multimap_r+\parr_r}{\Gamma;\Delta[\Phi] \rightarrow_\Sigma \mathbf{trans}(\mathbf{send}(id,m);r),\Omega \parallel \mathbf{object}(c,id,s),\Theta}\ bc$$

where AUX is the sequent $\Gamma;[\bot] \rightarrow_\Sigma class(c,\_,meths,\_) \parallel$

Fig. 2. Derivation scheme for method invocation and execution

of the intuitionistic implication $\Rightarrow$ in the previous clause implies that $class(\ldots)$ will be proved as a side condition (in fact, it does not involve the state of the database).

According to the previous description, the code of a method will have, basically, the following format:

$$\forall\ \mathbf{trans}(\mathbf{call}(\text{ID}, message);r) \parr \mathbf{frozen}(clname, \text{ID}, state) \circ\!\!-\ \mathbf{trans}(r';r) \parr \mathbf{object}(clname, \text{ID}, state')$$

where r' represents a sequence of statements added to the ones already present in r, and $state'$ is the state resulting from updating the old one. In this way, a method can update the state of an object and add new instructions to the transaction by which it was invoked. State updating, as it can be noted, is achieved through object rewriting.

Example 5.2 The method *changeIncome* of the class *person* can be coded like this:

$$\begin{array}{l}\lambda\text{ID}.\ \forall\ get(\text{STATE}.income, \text{INCOME})\ \&\\ \quad sum(\text{INCOME}, \text{AMOUNT}, \text{NEW})\ \&\\ \quad set(\text{STATE}.income, \text{NEW}, \text{STATE}')\ \Rightarrow\\ \quad\quad \mathbf{trans}(\mathbf{call}(\text{ID}, changeIncome(\text{AMOUNT})); \text{R}) \parr \mathbf{frozen}(person, \text{ID}, \text{STATE}) \circ\!\!-\\ \quad\quad\quad \mathbf{trans}(\text{R}) \parr \mathbf{object}(person, \text{ID}, \text{STATE}').\end{array}$$

The new state STATE′ is computed using the auxiliary predicates *get* (which retrieves the value of a record field), *sum* and *set* (which updates the value of a record field), and then the old object is properly rewritten. □

The general derivation scheme involving method invocation and execution is shown in figure 2.

OBJECT CREATION AND DELETION. Object creation can be modeled by the following clause, where **iddemon** represents a process, running in parallel with user transactions, that generates new identifiers for objects, and implemented as a simple counter:

$$\begin{array}{l}\forall\ class(\text{CLNAME}, \text{ATTRS}, \text{METHS}, \text{SUPCLNAME})\ \&\\ \quad buildrec(\text{ATTRS}, \text{VALUES}, \text{STATE})\ \Rightarrow\\ \quad\quad \mathbf{trans}(\mathbf{create}(\text{CLNAME}, \text{VALUES}, \text{ID}); \text{R}) \parr \mathbf{iddemon}(\text{ID}) \circ\!\!-\\ \quad\quad\quad \mathbf{trans}(\text{R}) \parr \mathbf{object}(\text{CLNAME}, \text{ID}, \text{STATE}) \parr \mathbf{iddemon}(s(\text{ID})).\end{array}$$

The auxiliary operation *buildrec* is used to build a record, given the field names and the respective values.

Example 5.3 We could create a new object of the class person by invoking the primitive **create** in the following way: $\mathbf{create}(person, [''Bond'', 100000, secret\ agent], \text{ID})$. □

Object deletion is very simple and can be managed by the clause

$$\forall\ \mathbf{trans}(\mathbf{delete}(\text{CLNAME}, \text{ID}); \text{R}) \parr \mathbf{object}(\text{CLNAME}, \text{ID}, \text{STATE}) \circ\!\!-\ \mathbf{trans}(\text{R}).$$

TRIGGERS. Triggers have the form

$$(name, events, condition, actions),$$

and can be associated to a class definition. Therefore, in order to store the information on triggers, we can modify the predicate *class* adding a new parameter *trlist* which contains the list of triggers associated with the respective class; the predicate *class* then becomes

$$class(clname, attrs, meths, supclname, trlist).$$

We shall consider here *deferred* triggers, i.e. triggers that are executed at the end of the user transaction that has caused rule triggering. In this case, the list of activated triggers can be collected, during transaction execution, in an additional parameter *activetr* added to transaction terms, which assume the format

$$\mathbf{trans}(code, activetr),$$

where *activetr* is the list of activated triggers (actually, their code). Trigger execution and transaction termination are then managed by the following rules:

$$\forall\ \mathbf{trans}(\bot, [\text{T}|\text{L}])\ \circ\!\!-\ \mathbf{trans}(\text{T}, \text{L}). \qquad \forall\ \mathbf{trans}(\bot, [])\ \circ\!\!-\ \bot.$$

All the clauses for the primitives of the DBMS must then be modified in order to take into account rule triggering (after the execution of a primitive it is necessary to check if there are any rules triggered, and in this case to update the list of activated ones). The clause for the primitive **modify** (the primitive for attribute modification), for instance, now becomes something like

$$\begin{array}{l}\forall\ set(\text{STATE.ATTR}, \text{V}, \text{STATE}')\ \&\\ \quad checktrigg(\text{CLNAME}, modify(\text{ATTR}), \text{T}')\ \&\\ \quad append(\text{T}, \text{T}', \text{T}'')\ \Rightarrow\\ \qquad \mathbf{trans}(\mathbf{modify}(\text{CLNAME.ATTR}, \text{ID}, \text{V}); \text{R}, \text{T})\ \wp\ \mathbf{object}(\text{CLNAME}, \text{ID}, \text{STATE})\ \circ\!\!-\\ \qquad\quad \mathbf{trans}(\text{R}, \text{T}'')\ \wp\ \mathbf{object}(\text{CLNAME}, \text{ID}, \text{STATE}').\end{array}$$

A number of extensions for trigger management can be added with little effort; these extensions include *trigger inheritance*, *immediate triggers*, and *priority triggers*.

QUERIES. For the sake of simplicity, we shall limit ourselves to queries on a single class, which can be written according to the following syntax:

$$\mathbf{select}(attrlist\ \mathbf{intolist}\ l\ \mathbf{from}\ clname\ \mathbf{where}\ cond),$$

whose informal meaning is to select all the objects belonging to the class *clname* which satisfy the condition *cond*, and to retrieve into the list l the values of the attributes specified in *attrlist*. We will assume an auxiliary operation $get(object.attr, value)$ for retrieving the value of an attribute. The condition *cond* can be expressed in the following way:

$$\lambda\text{OBJ}.\lambda\overline{\text{ATTRS}}.\text{SEL}\ \mathbf{and}\ \lambda\overline{\text{ATTRS}}.\text{COND},$$

where SEL is an **and** conjunction of *get* predicates which retrieve the needed values of the attributes, while COND is a condition over such values; $\overline{\text{ATTRS}}$ represents a tuple of variables. The use of λabstractions lets the conditions be evaluated on different objects yielding to different bindings for the abstracted variables.

Example 5.4 The first query of the example 4.3 can be written like this:

$$\mathbf{select}([name]\ \mathbf{intolist}\ l\ \mathbf{from}\ employee\ \mathbf{where}\ \lambda\text{OBJ}.\lambda\text{S}.get(\text{OBJ}.salary, \text{S})\ \mathbf{and}\ \lambda\text{S}.(\text{S} < 5000)).$$

□

$\forall$ **trans**(**select**(AL **intolist** L **from** CLNAME **where** COND); R) $\circ\!\!-$
(DEL $\Rightarrow$ **select**(AL **intolist** L **from** CLNAME **where** COND)) &
trans(R).

$\forall$ (SEL **object**(CLNAME, ID, STATE) $\overline{\text{VARS}}$) &
(COND $\overline{\text{VARS}}$) &
getmult(**object**(CLNAME, ID, STATE).AL, VL) $\Rightarrow$
select(AL **intolist** [VL|L] **from** CLNAME **where** (SEL **and** COND)) $\parr$
object(CLNAME, ID, STATE) $\circ\!\!-$
select(AL **intolist** L **from** CLNAME **where** (SEL **and** COND)).

$\forall$ (SEL **object**(CLNAME, ID, STATE) $\overline{\text{VARS}}$) &
not(COND $\overline{\text{VARS}}$) $\Rightarrow$
select(AL **intolist** L **from** CLNAME **where** (SEL **and** COND)) $\parr$
object(CLNAME, ID, STATE) $\circ\!\!-$
select(AL **intolist** L **from** CLNAME **where** (SEL **and** COND)).

$\forall$ **select**(AL **intolist** [] **from** CLNAME **where** CND).

Fig. 3. Clauses for the primitive **select**

Query semantics is defined by the clauses reported in figure 3, where DEL represents the linear clause $\forall$ **trans**(R) $\circ\!\!-$ $\bot$. The first clause lets the query be evaluated on a separate branch of the proof tree (the clause DEL allows the deletion of the transactions which are not involved in the query), while the second and third clauses effectively evaluate the query on one object at a time (every object is considered and deleted from the state after consideration). The second clause considers the case of an object which satisfies the query, while the second one the case of an object which does not. The last clause terminates the recursion. The current state of the computation is not affected by the evaluation of the query, as the evaluation takes place in a separate branch, as said before. There are many subtle points to consider, hence the reader is referred to [7] for a complete and detailed discussion; in that work, the reader can find as well extensions that take into account class inheritance and queries on multiple classes.

OTHER EXTENSIONS. In this paragraph we shall briefly sketch the implementation of other features of Chimera that, for reason of space, we cannot discuss in details. *Class attributes* and *methods* can be managed by means of object terms (in the state part of a sequent) like **c_object**($clname, state, meths$); method representation and calling can be simplified in this case. *Views* can be coded by means of clauses in the unbounded context of a sequent (they can be treated as predefined queries). Operations over lists of elements (which are very common in databases, and can be used to process query outputs) can be implemented in a straightforward manner. The notions of *persistent* and *temporary object* can be managed adding a new parameter $lifetime$ to the terms representing objects, and modifying the relevant rules. *Multiple inheritance* can be supported with little effort. User transactions can be enriched with some notion of *concurrency control*, which can be implemented using some form of *locking protocol*, and with *commit-rollback* primitives, which can be implemented, as in real databases, with some kind of *log file*. Actually, the meta-level behavior of an interpreter for $\mathcal{E}_{hhf}$(through backtracking) already provides us with some notion of 'rollback', to some extent.

6 Comparisons and Future Work

Based on the proof theoretical foundations given by Forum, in our work we have fixed an interpretation of linear sequents and proofs and a sublanguage, $\mathcal{E}_{hhf}$, in which it is possible to encode interesting examples of extensions of logic programming. Based on the resulting Linear Logic framework, in [12] we have shown

many different examples of high-level specifications of a variety of programming aspects, e.g., imperative assignments, processes simulation, object-oriented features.

Similarly to other approaches like LO [3], ACL [29], and Lygon [24] we have chosen a multi-conclusion setting in order to capture concurrent aspects in the logical derivations. However we have also tried to find a significant class of formulas suitable to merge the higher-order features of hereditary Harrop formulas [39] with the features of LL (thus looking for foundations of higher-order LL programming), whereas the cited approaches were focused on the LL aspects. These features also distinguish $\mathcal{E}_{hhf}$ from Lolli [27] focused on the problem of updates in a LL single-conclusion setting. As an application of this ideas, in this paper we have presented an encoding of Chimera in order to deal with various aspects of database management systems: state updates, queries, object-oriented features, deductive rules, triggers, user transactions, and so on. Viewing a computation as a search for uniform proofs provides us with a clear *operational* semantics, and, what's more, the logical nature of the adopted framework give us a well-founded *declarative* semantics.

An important point in the logic formalism of semantics aspects that we have outlined, lies in the choice of a general purpose logic as LL which, for the aspects it considers, can be considered as one of the most interesting non-standard logics applied to computer science. Furthermore, in our approach we are able to capture many aspects modeled by other ad hoc logical frameworks introduced in database-theory like Transaction Logic $\mathcal{TL}$ [6]. In this setting a general characterization of updates is given by using *oracles* which allow one to be parametric over the type of updates and a sort of trace semantics to model sequential and concurrent transactions. In our proof theoretical approach we have shown that sequential and parallel executions can be merged by using combinations of continuations passing and logical connectives (i.e., $\parr$). The operational semantics is clearly stated by the logical rules of LL. Furthermore, updates can be naturally modeled by using clauses of the form

$$old\_state \parr update \circ\!\!- new\_state$$

and thus with an implicit atomicity in its execution. In $\mathcal{TL}$ updates are written using the sequential operator in order to ensure the same behavior (since $\mathcal{TR}$ clauses are like Prolog-ones, i.e., with a single head).

Reasoning over updates and methods invocations is ensured here by a natural backtracking process that can be considered during the proof search process. The same idea can be used to rollback to a safe state.

Another important difference derives from the different nature of the considered formalisms. In fact, based on our previous work like [13] we are developing specifications for many different paradigms and in particular for object-oriented ones. Integrating this aspect into a database setting allows to fully characterize deductive object-oriented databases. An aspect not yet provided by $\mathcal{TL}$.

As future development of this work, we plan to study from a more general point of view the relations between Linear Logic and the other approaches (e.g., [6, 41]) used to formalize the semantics of database systems, in order to capture other important notions like *atomicity* of a sequence of operations.

Furthermore, we will implement a complete encoding of Chimera in the $\mathcal{E}_{hhf}$ system, under development using λProlog, to experiment the connections between databases languages and higher-order specification languages.

Notes

You can find more information about our work at the http address *www.disi.unige.it/person/DelzannoG/*.

References

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

2. J.M. Andreoli. Logic Programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
3. J.M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. In D.H. Warren and P.Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 495–510. The MIT Press, Cambridge, MA, 1990.
4. G. Bellin and Philip J. Scott. On the π-calculus and linear logic. Manuscript, 1992.
5. E. Bertino, G. Guerrini, and D. Montesi. Deductive Object Databases. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 213–235, 1994.
6. A. J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In *Proceedings of the JICSLP 96*, pages 142–156. The MIT Press, 1996.
7. M. Bozzano. A Linear Logic Specification of ChimeraM (in Italian) Master Thesis-DISI University of Genova, 1997.
8. S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules: the IDEA Methodology*. Addison Wesley, 1997.
9. S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active rule management in Chimera. In J. Widom and S. Ceri, editors, *Active Database Systems*. Morgan Kaufmann, 1994.
10. S. Ceri and R. Manthey. Consolidated specification of Chimera. Technical report, ESPRIT Rep.IDEA.DE.2P.006.01, November 1993.
11. J. Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.
12. G. Delzanno. *Logic & Object-Oriented Programming in Linear Logic*. PhD thesis, Università of Pisa, Dipartimento di Informatica, March 1997.
13. G. Delzanno and M. Martelli. Objects in Forum. In *Proceedings of the International Logic Programming Symposium*, pages 115–129. The MIT Press, 1995.
14. G. Delzanno and M. Martelli. Proofs as computations. In *Proceedings of the GULP 96*, pages 115–129, 1995.
15. U. Engberg and G. Winskel. Petri nets as models of linear logic. In A. Arnold, editor, *Proceedings of Colloquium on Trees in Algebra and Programming*, pages 147–161, Copenhagen, Denmark, 1990. Springer-Verlag LNCS 389.
16. L.J. Gallagher. Object SQL: Language Extensions for Object Data Management. In *Proceedings of the 1st International Conference on Information and Knowledge Management (CIKM)*, Baltimore, Maryland, November 1992.
17. J.H. Gallier. *Logic for Computer Science*. Harper and Row, 1986.
18. D. Galmiche and G. Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, 1994.
19. N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 327–336, 1991.
20. N. Gehani, H. Jagadish, and O. Shmueli. Event Specification in Active Object-Oriented Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.
21. V. Gehlot and C.A. Gunter. Nets as tensor theories. In G. De Michelis, editor, *Proceedings of the tenth International Conference on Application and Theory of Petri Nets*, pages 174–191, Bonn, Germany, 1989. Extended and revised version available as Technical Report MS-CIS-89-68, University of Pennsylvania.
22. J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1:1–102, 1987.
23. G. Guerrini, E. Bertino, and R. Bal. A formal definition of the Chimera object-oriented data model. *Journal of Intelligent Information Systems*, 1997.
24. J.A. Harland and D. J. Pym. The Uniform Proof-Theoretical Foundation of Linear Logic Programming (Extended Abstract). In *Proceedings of the International Logic Programming Symposium*, pages 304–318. The MIT Press, 1991.
25. J.A. Harland and D. J. Pym. On resolution in Fragments of Classical Linear Logic. In *LPAR '92, St. Petersburg, Lectures Notes in Artificial Intelligence*, pages 30–41, 1992.
26. J. Hodas and D. Miller. Representing Objects in a Logic Programming Language with Scoping Constructs. In D. H. Warren and P. Szeredi, editors, *Proceedins of 7th International Conference on Logic Programming*, pages 511–526. The MIT Press, Cambridge, MA, 1990.
27. J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

28. M. Kifer, G.Lausen, and J.Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, pages 741–843, 1995.
29. N. Kobayashi and A. Yonezawa. Asynchronous Communication Model based on Linear Logic. *Formal Aspects of Computing*, 3:279–294, 1994. Short version appeared in Joint International Conference and Symposium on Logic Programming, Washington, DC, November 1992, Workshop on Linear Logic and Logic Programming.
30. N. Kobayashi and A. Yonezawa. Logical, Testing, and Observation Equivalence for Processes in a Linear Logic Programming. Technical report, Department of Information Science the University of Tokyo, 1995.
31. P. Lincoln. *Computational Aspects of Linear Logic*. PhD thesis, Stanford Univeristy, 1992.
32. D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6:79–108, 1989.
33. D. Miller. Lexical Scoping as Universal Quantification. In *Proceedins of 6th International Conference on Logic Programming*, pages 268–283. The MIT Press, Cambridge, MA, 1989.
34. D. Miller. Unification Under a Mixed Prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.
35. D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P.Mello, editors, *Proceedings of the 1992 Workshop on Extension to Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 242–265. Springer-Verlag, Berlin, 1993.
36. D. Miller. A Multiple-Conclusion Meta-Logic. In *Proceedins of the 1994 Symposium on Logics in Computer Science, Paris*, pages 272–281, 1994.
37. D. Miller. Survey of Linear Logic Programming. *Computational Logic: The Newsletter of the European Network of Excellence in Computational Logic*, 2(2):63–67, 1995.
38. D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1):201–232, 1996.
39. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
40. G. Nadathur and D. Miller. Higher-Order Horn Clauses. *Journal of the ACM*, 37(4):777–814, 1990.
41. D. Peleg. Concurrent Dynamic-Logic. *Journal of ACM*, 34(1):450–479, 1987.
42. Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. *ENTCS*, 3, 1996.
43. H. Schellinx. Some Syntactical Observations on Linear Logic. *Journal of Logic and Computation*, 1(4):537–559, 1991.
44. M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23:733–742, 1976.

A A Proof System for Full Linear Logic

The considered language consists of
a) Simply typed λ-terms, i.e., constants, variables, abstractions ($\lambda\, x.t$) and applications ($t\ s$) (always considered in normal form); terms with type o correspond to formulas.
c) a set of *logical operators* $Op = \{0, 1, \bot, \top, ()^\bot, !, ?, \otimes, \parr, \&, \oplus, \forall, \exists\}$. It allows to construct the the LL formulas, following the grammar

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

where A is an atomic formula i.e. $h\ t_1\ \ldots\ t_n$ having type o. Quantifications are encoded by λ-terms as $\forall\lambda xF$, i.e., constants like $\forall$ having type $(\tau \to o) \to o$ are introduced in the language. Before presenting the inference rules of the linear sequent calculus (without quantification), we consider the linear negation that is essential for the symmetrical character of LL.
The negation of formula is defined by the following equalities:
$F^{\bot\bot} = F$, $\mathbf{1}^\bot = \bot$, $\bot^\bot = \mathbf{1}$, $\top^\bot = \mathbf{0}$, $\mathbf{0}^\bot = \top$, $(F \otimes G)^\bot = F^\bot \parr G^\bot$ and $(F \parr G)^\bot = F^\bot \otimes G^\bot$, $(F\&G)^\bot = F^\bot \oplus G^\bot$ and $(F \oplus G)^\bot = F^\bot\&G^\bot$, $(\forall xF)^\bot = \exists xF^\bot$ and $(\exists xF)^\bot = \forall xF^\bot$, $(!F)^\bot = ?F^\bot$ and $(?F)^\bot = !F^\bot$.

$$\frac{}{A, \Gamma \longrightarrow A, \Delta}\ identity \qquad \frac{\Gamma \longrightarrow A, \Delta \quad A, \Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ cut \qquad \frac{B, A, \Gamma \longrightarrow \Delta}{A, B, \Gamma \longrightarrow \Delta}\ ex_l \qquad \frac{\Gamma \longrightarrow B, A, \Delta}{\Gamma \longrightarrow A, B, \Delta}\ ex_r$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow ?A, \Delta}\ w? \qquad \frac{\Gamma \longrightarrow ?A, ?A, \Delta}{\Gamma \longrightarrow ?A, \Delta}\ c? \qquad \frac{A, !\Gamma \longrightarrow ?\Delta}{?A, !\Gamma \longrightarrow ?\Delta}\ ?_l \qquad \frac{\Gamma \longrightarrow A, \Delta}{\Gamma \longrightarrow ?A, \Delta}\ ?_r$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, !A \longrightarrow \Delta}\ w! \qquad \frac{\Gamma, !A, !A \longrightarrow \Delta}{\Gamma, !A \longrightarrow \Delta}\ c! \qquad \frac{A, \Gamma \longrightarrow \Delta}{!A, \Gamma \longrightarrow \Delta}\ !_l \qquad \frac{!\Gamma \longrightarrow A, ?\Delta}{!\Gamma \longrightarrow !A, ?\Delta}\ !_r$$

$$\frac{A, \Gamma \longrightarrow \Delta \quad B, \Gamma \longrightarrow \Delta}{A \oplus B, \Gamma \longrightarrow \Delta}\ \oplus_l \qquad \frac{\Gamma \longrightarrow A_i, \Delta}{\Gamma \longrightarrow A_1 \oplus A_2, \Delta}\ \oplus_r :\ i \in \{1, 2\}$$

$$\frac{A_i, \Gamma \longrightarrow \Delta}{A_1\&A_2, \Gamma \longrightarrow \Delta}\ \&_l :\ i \in \{1, 2\} \qquad \frac{\Gamma \longrightarrow A, \Delta \quad \Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A\&B, \Delta}\ \&_r$$

$$\frac{\Gamma \longrightarrow A, \Delta \quad \Gamma, B \longrightarrow \Delta}{A \rightsquigarrow B, \Gamma \longrightarrow \Delta}\ \rightsquigarrow_l \qquad \frac{A, \Gamma \longrightarrow \Delta}{\Gamma \longrightarrow A \rightsquigarrow B, \Delta}\ \rightsquigarrow'_r \qquad \frac{\Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A \rightsquigarrow B, \Delta}\ \rightsquigarrow''_r$$

$$\frac{A, \Gamma_1 \longrightarrow \Delta_1 \quad B, \Gamma_2 \longrightarrow \Delta_2}{A \parr B, \Gamma_1, \Gamma_2 \longrightarrow \Delta_1, \Delta_2}\ \parr_l \qquad \frac{\Gamma \longrightarrow A, B, \Delta}{\Gamma \longrightarrow A \parr B, \Delta}\ \parr_r$$

$$\frac{A, B, \Gamma \longrightarrow \Delta}{A \otimes B, \Gamma \longrightarrow \Delta}\ \otimes_l \qquad \frac{\Gamma_1 \longrightarrow A, \Delta_1 \quad \Gamma_2 \longrightarrow B, \Delta_2}{\Gamma_1, \Gamma_2 \longrightarrow A \otimes B, \Delta_1, \Delta_2}\ \otimes_r$$

$$\frac{\Gamma_1 \longrightarrow A, \Delta_1 \quad \Gamma_2, B \longrightarrow \Delta_2}{A \multimap B, \Gamma_1, \Gamma_2 \longrightarrow \Delta_1, \Delta_2}\ \multimap_l \qquad \frac{A, \Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A \multimap B}\ \multimap_r$$

$$\frac{\Gamma \longrightarrow \bot, \Delta}{\Gamma \longrightarrow \Delta}\ \bot \qquad \frac{}{\longrightarrow \mathbf{1}}\ \mathbf{1} \qquad \frac{}{\Gamma \longrightarrow \top, \Delta}\ \top \qquad \frac{}{\mathbf{0}, \Gamma \longrightarrow \Delta}\ \mathbf{0}$$

Fig. 4. A Sequent Calculus for (Propositional) LL.

B A Proof System for $\mathcal{E}_{hhf}$

The subset of LL formulas used in $\mathcal{E}_{hhf}$ is given by the following grammar:

$$\begin{array}{lcl}
\mathcal{D} & ::= & \mathcal{D} \,\&\, \mathcal{D} \mid \forall x.\mathcal{D} \mid \mathcal{H} \circ\!\!- \mathcal{G} \mid \mathcal{G} \Rightarrow \mathcal{D} \mid \mathcal{H}. \\
\mathcal{G} & ::= & \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid \forall x.\mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D} \Rightarrow \mathcal{G} \mid A \mid \bot \mid \top. \\
\mathcal{H} & ::= & \mathcal{H} \parr \mathcal{H} \mid A_r. \\
\mathcal{R} & ::= & \mathcal{R} \,\&\, \mathcal{R} \mid \mathcal{R} \parr \mathcal{R} \mid \forall x.\mathcal{R} \mid \bot \mid A_r \in \Pi_{\Sigma_{\mathcal{R}}}.
\end{array}$$

The proof system given in Figure 5 derives from Forum one by a specialization of its rules to $\mathcal{E}_{hhf}$-formulas. In the following figure $\langle \Delta \rangle$ is given by $\bigcup_{D \in \Delta} \langle D \rangle$, and $\langle D \rangle$ is given by: $\langle D_1 \& D_2 \rangle = \langle D_1 \rangle \cup \langle D_2 \rangle$, $\langle \forall_\tau x.D \rangle = \bigcup_{t \in \mathcal{T}_\tau} \langle D[t/x] \rangle$, where $\mathcal{T}_\tau$ is the set of Σ-terms of type τ, otherwise $\langle D \rangle = \{D\}$.

Search Rules

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

$$\frac{}{\Gamma; \Delta \longrightarrow_\Sigma \top, \Omega \parallel \Theta} \;\top_r \qquad \frac{\Gamma; \Delta \longrightarrow_\Sigma \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_\Sigma \bot, \Omega \parallel \Theta} \;\bot_r$$

$$\frac{\Gamma; \Delta \longrightarrow_{y:\tau,\Sigma} A[y/x], \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_\Sigma \forall_\tau x.A, \Omega \parallel \Theta} \;\forall_r \; (y \notin \Sigma) \qquad \frac{\Gamma; \Delta \longrightarrow_\Sigma A_1, A_2, \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_\Sigma A_1 \parr A_2, \Omega \parallel \Theta} \;\parr_r$$

$$\frac{B, \Gamma; \Delta \longrightarrow_\Sigma A, \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_\Sigma B \Rightarrow A, \Omega \parallel \Theta} \;\Rightarrow_r \qquad \frac{\Gamma; B, \Delta \longrightarrow_\Sigma A, \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_\Sigma B \multimap A, \Omega \parallel \Theta} \;\multimap_r$$

Backchaining (bc)

$$\frac{\Gamma; \Delta \longrightarrow_\Sigma \Omega \parallel \Xi}{\Gamma; \Delta, D \longrightarrow_\Sigma \Upsilon \parallel \Theta}$$

$H \circ\!\!- B \in \langle D \rangle$,
$H = A_1 \parr \ldots \parr A_n$,
$\{A_1, \ldots, A_n\} \subseteq \Upsilon \uplus \Theta$
$\Omega \uplus \Xi \equiv ((\Upsilon \uplus \Theta) \setminus \{A_1, \ldots, A_n\}) \uplus \{B\}$.

Guard Rule (gr)

$$\frac{\Gamma; \emptyset \longrightarrow_\Sigma G \parallel \emptyset \qquad \Gamma; \Delta, E \longrightarrow_\Sigma \Upsilon \parallel \Theta}{\Gamma; \Delta, D \longrightarrow_\Sigma \Upsilon \parallel \Theta} \qquad G \Rightarrow E \in \langle D \rangle.$$

Absorb

$$\frac{\Gamma, D; \Delta, D \longrightarrow_\Sigma \Upsilon \parallel \Theta}{\Gamma, D; \Delta \longrightarrow_\Sigma \Upsilon \parallel \Theta}$$

Initial

$$\frac{}{\Gamma; D \longrightarrow_\Sigma \Upsilon \parallel \Theta}$$

$(A_1 \parr \ldots \parr A_n) \in \langle D \rangle$,
$\{A_1, \ldots, A_n\} \equiv \Upsilon \uplus \Theta$.

Verify

$$\frac{}{\Gamma; \emptyset \longrightarrow_\Sigma \emptyset \parallel \Theta}$$

If there exists $\Phi \in \Gamma$ s.t. Φ is an $\mathcal{R}$-formula and Φ matches Θ .

Fig. 5. Rules for $\mathcal{E}_{hhf}$

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