Proof-theoretic and Higher-order Extensions of Logic Programming

Alberto Momigliano[1,2] and Mario Ornaghi[1]

[1] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
{momiglia,ornaghi}@dsi.unimi.it
[2] Laboratory for the Foundations of Computer Science, School of Informatics,
The University of Edinburgh, Scotland

Abstract We review the Italian contribution to proof-theoretic and higher-order extensions of logic programming; this originated from the realization that Horn clauses lacked standard abstraction mechanisms such as higher-order programming, scoping constructs and forms of information hiding. Those extensions were based on the Deduction and Computation paradigm as formulated in Miller et al's approach [51], which built logic programming around the notion of *focused uniform proofs* The Italian contribution has been both foundational and applicative, in terms of language extensions, implementation techniques and usage of the new features to capture various computation models. We argue that the emphasis has now moved to the theory and practice of logical frameworks, carrying with it a better understanding of the foundations of proof search.

1 Introduction and motivation

We start by trying to clarify the reasons behind our choice, discussion and classification of the literature stemming from the Italian contribution to proof-theoretic and higher-order extensions of logic programming (LP). These papers belong to the multitude of proposals of extensions of the foundations of logic programming, i.e. Horn clauses (HC). We can trace that both to the purported limited expressibility of HC — see the thorny issue of a logically motivated notion of negation — and to the lack of abstraction mechanisms that are present in modern programming languages to support the modular construction of software. Here we are referring to higher-order programming, modules, abstract datatypes, scoping constructs, state encapsulation and other forms of information hiding. One can argue that from the very beginning this has led to the introduction of "impure", i.e. extra-logical, features, such as cut, negation-as-failure or assert/retract. This outcome is not specific to LP and has been named *"recreating the Turing Machine" syndrome* [48]: starting from a computationally clean and semantically motivated language, one tends to add external mechanisms in order to make it suitable for programming-in-the-large. This inevitably tends to clutter the formal definition of the language (if any), making trusting the language itself and thus reasoning about it problematic.

Hence the opposing trend in the literature to go back to the original setting and base new constructs on more solid theoretical grounds, in our case, logic. A well-known (and somewhat worn out) example is again the logical foundations of negation. More

in general, it is by now usual to contrast the traditional model-theoretic approach (see Chapter [11] in this volume) to the proof-theoretic one, which "happens" to be at the core of most of the work about higher-order extensions of logic programming.

Arguably, many theoretical developments in logic have had an important impact in Computer Science. Concerning proof theory, we can isolate two different research directions, broadly corresponding to two different paradigms: Proofs as Programs and Deductions as Computations (DAC). In the Proof as Programs setting, proofs can be seen as programs (a.k.a. λ-terms), while computations correspond to (β)-reductions in a λ-calculus. The proof-theoretic basis is the normalisation or cut-elimination procedure. This approach fits with the foundations of functional programming, as well as of constructive program synthesis. In DAC, proofs themselves become the computations, while programs are specifications of non-logical symbols within the logic. Here, cut-elimination is the conditio sine qua non and proof-theory offers sophisticated restrictions to proof search in a cut-free system, while preserving completeness: a computation is modeled as a search for a proof, under suitable "uniformity" assumptions [51]. LP naturally falls in the DAC approach, which has been eloquently argued as one of its possible logical foundations elsewhere, e.g. [59].

In DAC, we distinguish between a non-logical signature, related to the problem domain, and the domain independent logical language. Each extension of the logical language has a corresponding extension of the proof system, bringing at the level of logic aspects that pertain to the computational level and allowing us to reason about them logically. A paradigmatic example of DAC is Miller et al's approach, which led to λProlog in the late 80's. The paper [47] clearly illustrates the basic ideas, starting from a precise notion of *uniform* proofs (to be defined shortly) and characterizing as "abstract logic programming systems" those where each goal has a uniform proof. The paper proves that (first-order) HC is an abstract LP system and then considers various extensions. In particular, it is shown how scoping and encapsulation can be modeled at the logical level, as well as how interesting higher-order programming techniques can be supported. Essentially, the idea behind abstract LP systems is that a sequent such as $\Sigma : \Gamma \longrightarrow G$ represents the state of an idealized LP interpreter with current program Γ, goal G and signature Σ. Both Γ and Σ may dynamically change during the computation. A goal-directed or uniform proof is then a cut-free proof in which every occurrence of a sequent whose right-hand side is non-atomic is the conclusion of a right-introduction rule. It uses a suitable backchaining rule to "invoke" the definitions of the non-logical symbols provided by Γ when an atomic goal A is reached. Examples of right (introduction) rules are $\forall_R$ and $\supset_R$, while BC is the backchain rule.

$$\frac{\Sigma, c : \Gamma \longrightarrow G(c)}{\Sigma : \Gamma \longrightarrow \forall x.\, G(x)}\,\forall_R \qquad \frac{\Sigma : \Gamma, D \longrightarrow G}{\Sigma : \Gamma \longrightarrow D \supset G}\,\supset_R \qquad \frac{\Sigma : \Gamma \longrightarrow G}{\Sigma : \Gamma \longrightarrow A}\,BC, G \supset A \in \langle\Gamma\rangle$$

The $\forall_R$ rule augments the signature by a new constant c of the type of x, while $\supset_R$ augments the program by the clause D. The backchaining rule *selects* a program clause $G \supset A$ in the *closure* $\langle\Gamma\rangle$ of Γ under the $\forall_L, \wedge_L$ rules and backchains on it (see Section 2.1 when this idea is realized via *focusing*). An abstract logic programming language is then a logical system for which uniform proofs are complete. To make our discussion more concrete we consider an example taken from [44], illustrating scoping and modularity.

Example 1. Consider the well known Prolog reverse program;

```
reverse(L,R) :- r(L,R,[]).
r([],Ys,Ys).
r([X|Xs],R,Ys) :- r(Xs,R,[X|Ys]).
```

`reverse/2` uses an auxiliary accumulator-based predicate `r/3` to implement the following simple algorithm: *start with the pair* ⟨`L`, `[]`⟩ *and iteratively push the elements of the first list into the second one*. This example shows two problems. Firstly, the definition of `r` ought to be used *locally*, inside the scope of the main predicate, but Prolog cannot (declaratively) hide it against undesired redefinitions. Secondly, the simple reverse algorithm needs only the variables `L` and `Ys` of `r(L,R,Ys)`: `R` merely captures the final result and passes it to the `reverse` predicate. Both problems can be solved by introducing suitable scoping mechanisms. The following shows how this can be accounted for using higher-order universal quantification and embedded implication to provide scope to the definition of the auxiliary predicate and to the individual variables used in it.

```
reverse(L,R) :-
 all rev\ (
         (rev([],R),
          all X,Xs,Ys\ (rev([X|Xs],Ys) :- rev(Xs,[X|Ys])))
              => rev(L,[])
        )
```

The notation follows [44], in particular `all r\ G(r)` is concrete syntax for $\forall x{:}\tau.\, G(x)$. ◇

Roughly, an interpreter based on uniform proof search will proceed as follows. To prove a goal, such as `reverse([1,2],V)`, it will replace `r` by a *new* binary predicate symbol, say `c`, and add to the current program the clauses:

```
c([],V), (all X,Xs,Ys\ c([X|Xs],Ys) :- c(Xs,[X|Ys])).
```

Then it will try to prove `c([1,2],[])` backchaining on the rightmost clause. The variable `V` will be instantiated to `[2,1]` with two further backchain steps, when the computation will eventually succeed with the goal `c([],[2,1])`.

Logically, the module corresponds to the following formula, where *ls* is the sort of lists, *i* the sort of integers, and *o*, as usual, is the type of propositions:

$$D_{rev} : \forall_{ls}\, l\, r.\, (\forall_{ls\to ls\to o} rev.\, rev([], r) \wedge \\ \forall_i\, x.\, \forall_{ls} x_s\, y_s.\, rev(x_s, [x|y_s]) \supset rev([x|x_s], y_s)) \supset rev(l, []) \supset (reverse(l, r))$$

In terms of logical rules, the behaviour of the interpreter corresponds to the gradual construction of the following proof tree, where we informally label the clauses on which

we backchain:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, c : \Gamma, D_{c1} : c([], V), D_{c2} : \forall\ x\ y_s\ x_s.\ (c(x_s, [x|y_s]) \supset c([x|x_s], y_s)) \longrightarrow c([], [2, 1])}{\Sigma, c : \Gamma, D_{c1} : c([], V), D_{c2} : \forall\ x\ y_s\ x_s.\ (c(x_s, [x|y_s]) \supset c([x|x_s], y_s)) \longrightarrow c([2], [1])}\ BC, D_{c2}}{\Sigma, c : \Gamma, D_{c1} : c([], V), D_{c2} : \forall\ x\ y_s\ x_s.\ (c(x_s, [x|y_s]) \supset c([x|x_s], y_s)) \longrightarrow c([1, 2], [])}\ BC, D_{c2}}{\Sigma, c : \Gamma \longrightarrow c([], V) \wedge \forall\ x\ y_s\ x_s.\ (c(x_s, [x|y_s]) \supset c([x|x_s], y_s)) \supset c([1, 2], [])}\ \supset_R, \wedge_L}{\longrightarrow \forall\ rev.\ (rev([], V) \wedge \forall x\ y_s\ x_s.\ (rev(x_s, [x|y_s]) \supset rev([x|x_s], y_s)) \supset rev([1, 2], []))}\ \forall_R}{\Sigma : \Gamma \longrightarrow reverse([1, 2], V)}\ BC, D_{rev}$$

We remark that the generation of new names required by the proof rule for $\forall$ protects the definition of `r`, since different uses will employ different names. Here, its definition is visible only to calls to `reverse` and will be discharged upon success. Furthermore, the possibility of using the definition of a predicate in the body of a clause and the explicit use of quantifier `all X,Xs,Ys` allows us to link the variable `R` in the definition of `reverse` precisely to the variable `R` of the predicate that will accumulate the final result, i.e., to `c([],R)`.

The previous example typifies our viewpoint: seeking extensions of LP in terms of languages endowed with a notion of uniform proofs, more precisely *focused* uniform proofs [2]. This shows a twofold *duality*:

- Between goals and clauses: a negative subformula of a goal is a program clause and a negative subformula of a program clause is a goal.
- Between goal-oriented proof search and clause selection (focusing), once backchaining is seen in a more general light.

This duality is more clearly seen in linear logic, where following Andreoli [2], each connective carries an unique intrinsic attribute called a *polarity* that determines its behaviour under search. Hence connectives can can be partitioned into *asynchronous* (those whose right rule is invertible) and *synchronous* (those whose left rule is invertible). This yields a highly normalized proof search mechanism, based on a systematic interleaving between asynchronous and synchronous reductions: one decomposes the asynchronous formulas until none remains, then picks a synchronous formula and decomposes it until new asynchronous subformulae arise, and so on. Proofs of this kind are called *focused proofs* and can be shown to be complete for entire classical linear logic. In the linear setting the polarity of a connective coincides with its being *positive/negative*: however Andreoli noted that an *arbitrary*, albeit fixed, assignment of polarity to atoms (a *bias*) will preserve completeness of focusing, with the understanding that a [negative] positive bias denote [a]syncronous behaviour. Notwithstanding its asymmetry, this observation applies to intuitionistic logic as well. In fact, it can be shown that, for the Horn fragment, a positive bias to atoms yields hyper-resolution (forward chaining), while a negative one SLD-resolution (backward chaining) [25]. More in general, uniform proofs can be seen as a special case of focusing, where atoms are given negative bias, which happens to be complete only when existentials and disjunctions are excluded from the syntax. These observation have been significantly generalized in [42].

However, there is another angle to "higher-order" extensions to which we have not done justice yet: work related to languages based on some form of λ-calculus. This is indeed the second way a language such as λProlog extends ordinary LP, an issue which was often argued for, when not distrusted since the early 80's [64]. The original rationale was adding some of the higher-order features of functional programming, namely handling functions (here predicates) as first-class citizens, without changing the computation paradigm. A classic example is the *mappred* predicate, corresponding to the `map` combinator in a language such as Standard ML:

Example 2.

```
mappred(P,[],[]).
mappred(P,[X|Xs],[Y|Ys]) :- P(X,Y), mappred(P,Xs,Ys).
```

A sample goal could be

```
P = (lambda x y\ reverse(x,y)), mappred(P,[[1,2],[3,4]],Ys).
```

with answer substitution `Ys = [[2,1],[4,3]]`. ◇

Although some nifty applications based on these features emerged early on, e.g. [32], predicate-as-values, we argue, never managed to attain the same prominence that it has in functional programming. Functional quantification instead has had a pivotal role in the theory and practice of *logical frameworks* [60], in so much as it supports higher-order abstract syntax (HOAS) [61]. This is a declarative treatment of the syntax of object logics, whose binding operators are all rendered via the λ-abstraction of the meta-logic, while bound variables of the object and meta-logic are identified. In this way seemingly banal but tediously complicated issues induced by α-equivalence and substitution principles are taken care once and for all by the meta-logic, making the specification and reasoning over object logic more concise and effective. This opened up an all new field, as we briefly touch upon in the Conclusions.

The rest of this overview is organized as follows: Section 2 succinctly presents the syntax and proof rules underlying the main LP language that we consider in separate subsections. In Section 3 we follow the same schema, highlighting the Italian contribution to the corresponding broad areas. Section 4 concludes by trying to evaluate the impact of these works on LP and computational logic more in general.

2 Calculi for intuitionistic and linear logic programming

Uniform proofs and abstract LP systems were presented in [51] as the basis for proof-theoretic extensions of LP. At about the same time, Girard's 1987 "Linear Logic" paper had a rippling effect in computer science and logic programming was quick to follow suit. In his 1990 thesis Andreoli established the foundation of focusing proofs in linear logic [2]. In 1991 the uniform proof approach was extended to linear logic programming by Miller & Hodas [41]. We start with the logic underlying λProlog.

$$\frac{}{\Sigma : \Gamma \longrightarrow \top}\ \vdash \top \qquad \frac{\Sigma : \Gamma \longrightarrow G_1 \qquad \Sigma : \Gamma \longrightarrow G_2}{\Sigma : \Gamma \longrightarrow G_1 \wedge G_2}\ \vdash \wedge$$

$$\frac{\Sigma : (\Gamma, D) \longrightarrow G}{\Sigma : \Gamma \longrightarrow D \supset G}\ \vdash \supset \qquad \frac{(\Sigma, c{:}A) : \Gamma \longrightarrow [c/x]G}{\Sigma : \Gamma \longrightarrow \forall x{:}\tau.\, G}\ \vdash \forall^c$$

$$\frac{\Sigma : \Gamma \xrightarrow{D} A}{\Sigma : \Gamma \longrightarrow A}\ \vdash \mathrm{fcs}, D \in \Gamma$$

. . . . . .

$$\frac{\Sigma : \Gamma \vdash A_r \doteq A : o}{\Sigma : \Gamma \xrightarrow{A_r} A}\ \mathrm{fcsAt} \qquad \frac{\Sigma : \Gamma \xrightarrow{D_i} A}{\Sigma : \Gamma \xrightarrow{D_1 \wedge D_2} A}\ \mathrm{fcs}\wedge_i$$

$$\frac{\Sigma : \Gamma \xrightarrow{[t/x]D} A}{\Sigma : \Gamma \xrightarrow{\forall x:\tau.\ D} A}\ \mathrm{fcs}\forall, \Sigma \vdash t : \tau \qquad \frac{\Sigma : \Gamma \longrightarrow G \qquad \Sigma : \Gamma \xrightarrow{D} A}{\Sigma : \Gamma \xrightarrow{G \supset D} A}\ \mathrm{fcs}\supset$$

Figure 1. Focused intuitionistic proofs for HOHF

2.1 λProlog

It is based on the so-called Higher-Order Hereditary Harrop Formulas, an intuitionistic fragment of Church higher-order logic. As we have mentioned in the Introduction, it enhances Prolog in two directions. The term language is extended to allow arbitrary λ-terms under full higher-order unification and the formula language is extended to allow usage of arbitrarily nested universal quantifiers and implications. It can be synthesized by the following grammar:

$$\begin{aligned}
\textit{Terms}\quad & t ::= c \mid x \mid \lambda x{:}\tau.\, t \mid t_1\ t_2 \\
\textit{Atoms}\quad & A ::= A_r \mid A_f \\
\textit{Clauses}\quad & D ::= A_r \mid G \supset D \mid D_1 \wedge D_2 \mid \forall x{:}\tau.\, D \\
\textit{Goals}\quad & G ::= A \mid \top \mid G_1 \wedge G_2 \mid D \supset G \mid \forall x{:}\tau.\, G \\
\textit{Signatures}\quad & \Sigma ::= \cdot \mid \Sigma, x{:}\tau \\
\textit{Programs}\quad & \Gamma ::= \cdot \mid \Gamma, D
\end{aligned}$$

We shall be fairly loose with typing issues, noting only that we use a ML-like prenex polymorphic system, so that for example universal quantification is given the type $\forall\alpha.\ (\alpha \supset o) \supset o$. To preserve the operational reading of logic programs as predicate definitions we require clause heads to be *rigid* atoms, denoted A_r, i.e. the head symbol is not a (free) variable.[1] Otherwise, we call the atom *flexible*, denoted A_f. Note that we could add existentials and disjunctions to the syntax of goals, but with no real expressive enhancement—see [56] for an investigation into *maximal* abstract logic programming languages.

[1] We gloss over other minor syntactic restrictions of occurrences of logical connectives in the scope of rigid atoms required to preserve goal-orientedness during proof search.

Some terminology: the above language is named HOHF; with H^fOHF we denote its restriction to quantification over variable and function symbols, that is o is only allowed as a range type. Examples of H^fOHF are Miller's L_λ [45] and LF [39]. FOHF is the further restriction to first-order quantification.

We now introduce a focused version of the uniform proofs system of [51] (Fig. 1); it defines the following judgements, where Γ contains the program and the possible dynamic assumptions; the judgment $\Sigma : \Gamma \vdash A_r \doteq A : o$, whose definition we omit and refer to the judgmental version in [22]), denotes higher-order unification.

$\Sigma : \Gamma \longrightarrow G$	Program Γ under signature Σ uniformly entails goal G.
$\Sigma : \Gamma \xrightarrow{D} A$	Focused clause D from Γ under signature Σ entails atom A.

We remark that the backchain rule *BC* of [44], considered in the introduction, can be derived by applying the focusing rules until the head of a clause is deemed to unify the atom on the right and then recursively applying the $\vdash$ rules.

2.2 Lolli

Based on the first-order language freely generated by multiplicative implication $\multimap$, additive unit, implication, conjunction and universal quantification, *Lolli*'s uniform proofs system [41] uses a single-conclusion sequent calculus (Fig. 2) that distinguishes two zones, Γ containing the (reusable) program together with the possible intuitionistic dynamic assumptions and Δ, containing the linear ones, seen as a multiset. Notice that while Lolli is first-order, its type-theoretic counterpart, the Linear Logical Framework LLF [23], has functional quantification; however, they have the same proof search aspects, safe from linear unification, as we detail in Section 3.2.

$\Sigma : \Gamma; \Delta \longrightarrow G$	Clauses $\Gamma; \Delta$ under signature Σ uniformly entails goal G.
$\Sigma : \Gamma; \Delta \xrightarrow{D} A$	Focused clause D from Γ *or* $\Delta \setminus D$ under signature Σ entails atom A.

We briefly examine the crucial rules, deviating from the literature by using the same notation for additive connectives as for their intuitionistic counterparts: the fcsAt rule encodes both initial rules of a linear calculus, by requiring the linear context to be empty: in fact, if the focus is on a linear A, then this must be the only assumption that can and must be consumed. If instead the focus is intuitionistic, there must be no leftover resources, lest the computation is failed. Note also the non-deterministic partitioning of the linear context in the focusing rule for $\multimap$, highlighted by the notation $\cup$ for multiset union, to be read backwards as resource splitting. From an additive viewpoint, rule $\vdash \top$ features an implicit weakening, while $\vdash \wedge$ an implicit contraction, both w.r.t. Δ.

We now give a first linear algorithm for reversing a list.

$$\frac{}{\Sigma : \Gamma; \Delta \longrightarrow \top} \vdash \top \qquad \frac{\Sigma : \Gamma; \Delta \longrightarrow G_1 \qquad \Sigma : \Gamma; \Delta \longrightarrow G_2}{\Sigma : \Gamma; \Delta \longrightarrow G_1 \wedge G_2} \vdash \wedge$$

$$\frac{\Sigma : (\Gamma, D); \Delta \longrightarrow G}{\Sigma : \Gamma; \Delta \longrightarrow D \supset G} \vdash \supset \qquad \frac{\Sigma : \Gamma; (\Delta \cup \{D\}) \longrightarrow G}{\Sigma : \Gamma; \Delta \longrightarrow D \multimap G} \vdash \multimap$$

$$\frac{\Sigma : \Gamma; \Delta \xrightarrow{D} A}{\Sigma : \Gamma; \Delta \longrightarrow A} \vdash \mathrm{fcs}_\Gamma, D \in \Gamma \qquad \frac{\Sigma : \Gamma; \Delta \xrightarrow{D} A}{\Gamma; (\Delta \cup \{D\}) \longrightarrow A} \vdash \mathrm{fcs}_\Delta$$

$$\frac{}{\Sigma : \Gamma; \cdot \xrightarrow{A} A} \mathrm{fcsAt} \qquad \frac{\Sigma : \Gamma; \Delta \xrightarrow{D_i} A}{\Sigma : \Gamma; \Delta \xrightarrow{D_1 \wedge D_2} A} \mathrm{fcs}\wedge_i$$

$$\frac{\Sigma : \Gamma; \cdot \longrightarrow G \qquad \Sigma : \Gamma; \Delta \xrightarrow{D} A}{\Sigma : \Gamma; \Delta \xrightarrow{G \supset D} A} \mathrm{fcs} \supset \qquad \frac{\Sigma : \Gamma; \Delta_1 \longrightarrow G \qquad \Sigma : \Gamma; \Delta_2 \xrightarrow{D} A}{\Sigma : \Gamma; (\Delta_1 \cup \Delta_2) \xrightarrow{G \multimap D} A} \mathrm{fcs} \multimap$$

Figure 2. Main rules of a focused calculus for Lolli

Example 3.

```
reverse(Xs, Ys) :- once(perm(Xs, Ys)).

perm([X|Xs], Ys) o- (elm(X) -o perm(Xs, Ys)).
perm([], Ys) o- perm(Ys).

perm([]).
perm([X|Xs]) o- elm(X) ^ perm(Xs).
```

The `perm/2` predicate simply loads (in reversed order) the elements of the input list in the linear context in the form of `elm`$(\cdot)$ assumptions; then calls `perm/1`, which consumes those assumptions. Because of the non-deterministic splitting induced by focusing on the second clause of `perm/1`, we generate, upon backtracking, all permutations of the given list. Hence the main `reverse` predicate selects the first solution with the meta-predicate `once/1`. ◇

2.3 LO

Linear Objects [4, 5] was the first proposal for a linear logic programming language. It extends Horn logic by generalizing clause heads to multisets of atoms connected by multiplicative disjunction ($\parr$), i.e. clauses have the form

$$G \multimap A_1 \parr \ldots, \parr A_n$$

The starting point was the family of concurrent LP languages (see Chapter [35] in this volume) as a way to provide a logical account of object-oriented computations: objects

are viewed as AND-concurrent, stream-communicating via shared variables (proof) processes, where the arguments in a goal are the slots and communication streams of an object. State transitions are realized with inference steps. For a canonical example, the goal

$$\texttt{point(InStrm,5,7,OutStrm)}$$

encodes a point with the given coordinates and communication streams `InStrm` and `OutStrm`, where a method (clause) such as

$$\texttt{point([proj-x|InStrm],X,Y,OutStrm) :- point(InStrm,X,0,OutStrm)}$$

specifies the transition resetting `Y` to `0` upon reception of the `proj-x` message.

In this sense, LO inherits an effect-free view of objects and does not exploit the linear logic context for state manipulation as in Lolli, since it lacks any form of scoping constructs. On the other hand, when seen as OR-concurrency $\parr$ directly supports a view of objects as multiset of independent units. The above object becomes

$$\texttt{point} \parr \texttt{in(InStrm)} \parr \texttt{x(5)} \parr \texttt{y(7)} \parr \texttt{out(OutStrm)},$$

where different atoms encode a point, its coordinates and communication mediums. In this way objects are amenable of inheritance, since a more specialized objects such as

$$\texttt{point} \parr \texttt{in(InStrm)} \parr \texttt{x(5)} \parr \texttt{y(7)} \parr \texttt{out(OutStrm)} \parr \texttt{colour(red)}$$

can call a method (a clause with multiple heads) such as

$$\texttt{point} \parr \texttt{in([proj} - \texttt{x|InStrm])} \parr \texttt{y(Y)} \multimap \texttt{point} \parr \texttt{in(InStrm)} \parr \texttt{y(0)}$$

by matching only a sub-multiset of the atoms encoding an object. Synchronizations of this kind can be managed using multiset rewriting techniques, but, as we will see in Section 3.2, such synchronization is expensive.[2]

LO's original proof theory [4] did not make focusing explicit, but the crucial rules can be reconstructed as in Figure 3, where we use as a one-sided multi-succedent calculus; since proof search has no dynamics, we can fix the program $\mathcal{P}$ and dispose of the signature.

$\longrightarrow \mathcal{G}$	Program $\mathcal{P}$ uniformly entails multiset of goals $\mathcal{G}$.
$\xrightarrow{D} \mathcal{A}$	Focused clause D from $\mathcal{P}$ entails multiset of atoms $\mathcal{A}$.

To better illustrate the operational semantics of LO, we revisit once more the simple reverse algorithm, where we manage to attain the same behavior of Example 1 even in the absence of scoping constructs: in fact, we exploit OR-concurrency to capture the final result and pass it to the main predicate.

[2] Historically, this is the first observation that the operational semantics of linear LP brings into intuitionistic proof search an additional "don't know" non-determinism.

$$\frac{\longrightarrow \{G_1, G_2\} \cup \mathcal{G}}{\longrightarrow \{G_1 \parr G_2\} \cup \mathcal{G}} \vdash \parr$$

$$\frac{\longrightarrow \{G\} \cup \mathcal{A}_1 \qquad \xrightarrow{A_1 \parr \ldots \parr A_n} \mathcal{A}_2}{\xrightarrow{G \multimap A_1 \parr \ldots \parr A_n} \mathcal{A}_1 \cup \mathcal{A}_2} \text{fcs} \multimap \qquad \frac{\longrightarrow G \qquad \xrightarrow{A_1 \parr \ldots \parr A_n} \mathcal{A}}{\xrightarrow{G \supset A_1 \parr \ldots \parr A_n} \mathcal{A}} \text{fcs} \supset$$

$$\frac{\xrightarrow{D_1} \mathcal{A}_1 \qquad \xrightarrow{D_2} \mathcal{A}_2}{\xrightarrow{D_1 \parr D_2} \mathcal{A}_1 \cup \mathcal{A}_2} \text{fcs} \parr$$

Figure 3. $\parr$-related rules in LO

Example 4.

$$\texttt{dr} \ : \ \texttt{reverse(Xs, Ys)} \ \texttt{:-} \ \texttt{rev(Xs, [])} \parr \texttt{result(Ys)}.$$

$$\texttt{dc} \ : \ \texttt{rev([X|Xs], Ys)} \multimap \texttt{rev(Xs, [X|Ys])}.$$
$$\texttt{dn} \ : \ \texttt{rev([], V)} \parr \texttt{result(V)}.$$

Intuitively, we backchain on the method dc until the input list is exhausted. Then we awaken the result(V) object by matching it with the dn method and return the instantiation for V. ◇

This corresponds to this proof-tree, where again we informally use a *BC* rule, decorated with the label of the clause on which we focus.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\texttt{L} \doteq [2,1]}{\xrightarrow{\texttt{rev([],L)}} \mathit{rev}([],[2,1])} \qquad \cfrac{\texttt{L} \doteq V}{\xrightarrow{\texttt{result(L)}} \mathit{result}(V)}}{\longrightarrow \mathit{rev}([],[2,1]),\ \mathit{result}(V)}\ BC, \texttt{dn}}{\longrightarrow \mathit{rev}([],[2,1]) \parr \mathit{result}(V)} \vdash \parr}{\longrightarrow \mathit{rev}([],[2,1]) \parr \mathit{result}(V)}\ BC, \texttt{dc}}{\longrightarrow \mathit{rev}([1,2],[]) \parr \mathit{result}(V)}\ BC, \texttt{dr}}{\longrightarrow \mathit{reverse}([1,2], V)}$$

Note that it is crucial that dc uses linear implication, allowing one to split resources as required.

We conclude this Section noting that LO's $\parr$ can also be seen as a form of *constructive disjunction* yielding indefinite answers; we will touch upon this links between linear and disjunctive logic programming in Section 3.3.

2.4 Forum

Forum [49] can be seen as the fusion of Lolli and LO and allows one to view entire linear logic as an abstract LP language. Indeed, simply adding multiplicative falsity $\bot$ to Lolli

yields a "goal-oriented" presentation of linear logic. Thus linear negation $B^{\perp}$ can be defined as expected ($B \multimap \perp$) and hence the other connectives by de Morgan dualities. In particular we can also view $B \parr C$ as $(B \multimap \perp) \multimap C$. Note that while these encodings do not interfere with the soundness and completeness of focused uniform proofs, they do not yield a predictable operational semantics such as the one a programmer would expect. In fact, focusing on $\perp$ is rather non-informative, leading a naive interpreter into a tight and endless loop. Thus, the view of Forum as a *specification* language [26] and efforts, some of which we mention in Section 3.4, to find a meaningful sub-language amenable to a programming language interpretation.

The relevant judgments comprise two-sided multi-succedent sequents where Γ, Φ have intuitionistic maintenance, and $\Delta, \mathcal{G}$ have a linear one.

$\Sigma : \Gamma; \Delta \longrightarrow \mathcal{G}; \Phi$	Clauses $\Gamma; \Delta$ under signature Σ uniformly entails multisets of goals $\mathcal{G}; \Phi$.
$\Sigma : \Gamma; \Delta \xrightarrow{D} \mathcal{A}; \Phi$	Focused clause D from Γ *or* $\Delta \setminus D$ under signature Σ entails multisets of atoms $\mathcal{A}$ and goals Φ.

We refer to [50] for the twenty proof rules.

3 The Italian contribution

The origin of the Italian interest in proof-theoretic extensions of LP can be traced back to Gabbay and Reily's N-Prolog [33,34], which featured embedded implication in goals, but no universal quantification: free variables can be shared in an implicational goal, creating certain difficulties especially when coupled with negation-as-failure. This language sparked a lot of interest, especially in Torino: A. Martelli, Giordano and others extensively researched applications w.r.t. modules and scoping constructs and extension to modal analysis, see e.g. [9]. We will not analyze this further as already well detailed in [17]. We will, however, briefly mention [37] that fixes some of the problems raised in [33]. The authors propose an operational semantics extending Stärk's ESLDNF, establishing a soundness and completeness for non-floundering queries is with respect to a completion theory interpreted in a three-valued modal logic.

3.1 λProlog

The second "wave" was initiated by Miller's sabbatical in Edinburgh, where he supervised Pareschi's thesis [58]; the latter exploited hypothetical reasoning and λ-terms to encode in a computational environment the features of certain linguistic theories, e.g. the rendering of *filler-gap dependencies*. Pareschi then hooked up with Andreoli to develop LO as we have mentioned in Section 2.3. Miller also supervised Arcelli's thesis [6] in Milano, where she related second-order λProlog to Reflective Prolog [27]. She and coauthors went on investigating applications of the language for example to program transformations [7]. Independently, Momigliano [52] extended Miller's [46], providing a way of encoding via the double negation translation of all classical logic into a focused uniform system. The language was FOHF, but the approach would apply to $\mathrm{H^fOHF}$ as well.

In [53] the issue of endowing a logical framework (namely H^{f}OHF) with a logically justified notion of negation is re-addressed, adapting the idea of *elimination* of negation [10] to the higher-order setting. This includes two separate phases. *Complementing terms*, i.e. in this case higher-order patterns: due the presence of *partially applied* λ-terms, intuitionistic λ-calculi are not closed under complementation, thus requiring one to develop a *strict*, i.e. relevant, λ-calculus, where we can directly express whether a function (here typically a higher-order logic variable) ought or not depend on its arguments. *Complementing clauses*, which can be seen as a negation normal form procedure which is consistent with intuitionistic provability. It entails finding a middle ground between the CWA usually associated with negation and the OWA typical of logical frameworks. This has come to be known as the *Regular World Assumption* that has shown to be a central notion in inductive meta-theorem proving [40, 63] in systems such as *Twelf* [62].

3.2 Lolli

A problem specific to proof search in linear logic is how to effectively split resources when dealing with multiplicative connectives, without trying exponentially many partitions of the linear context. Hodas and Miller developed a lazy splitting approach for the operational semantics of Lolli, called the input-output model of resource consumption [41]. This turns out to be just an instance of a more general resource management problem in linear logic programming (and, with a somewhat different emphasis, in linear theorem proving). As pointed out and addressed in [21], a properly understood operational semantics has to deal with two additional features. First, the $\top$ connective is allowed to consume any resource, a feature which is handy to wind up with success certain computations without burdening the user with tracking and consuming any remaining assumption. Secondly, additive conjunction requires *strict* resources, i.e. those which *can* be duplicated but *must* be used during the solution of a given goal. A final contribution of this paper is the *residuation* calculus, a form of resolution for sequent calculi that pushes all non-determinism out of focusing and into the introduction rules. This has also applications in proof-theoretic compilation [19].

The (linear) spine calculus [24] is an answer to a related issue: devising an efficient representation of the (linear) λ-calculus, tailored to make building blocks of LP such as unification efficient even in the higher-order case. In fact, and differently from the first-order case, even restricting to terms of atomic type, in a token such as

$$(\ldots(h\ M_1)\ldots M_n) \tag{1}$$

the head is deeply buried and hence not immediately accessible. This is further complicated in the linear case, where destructors can be arbitrarily interleaved. In the spine calculus every atomic term has the form $H \cdot S$, where H is the *root* and S the *spine*: a term such as (1) translates into $h \cdot (U_1; \ldots; U_n; \mathrm{NIL})$, where $';'$ associates to the *right*, U_i translates M_i and NIL represents the end of the spine.

The relevance of this contribution is twofold:

1. The restriction of this calculus to the intuitionistic case is the internal representation adopted in Twelf and it is also at the basis of the *Tejus* compiler for λProlog [57].

2. Exploiting the Curry-Howard correspondence, spines can be seen as a term assignment language for uniform provability, in particular for Lolli, LLF [23], and for any subsystem thereof, as we exemplify in Figure 4.

$$\frac{\Gamma, x : D \longrightarrow U : G}{\Gamma \longrightarrow (\lambda x{:}D.\ U) : D \supset G} \vdash \supset \qquad \frac{\Gamma \xrightarrow{h:D} S : A}{\Gamma \longrightarrow (h \cdot S) : A} \vdash \mathrm{fcs}, D \in \Gamma$$

$$\frac{}{\Gamma \xrightarrow{\mathrm{NIL}:A} A}\ \mathrm{fcsAt} \qquad \frac{\Gamma \longrightarrow U : G \qquad \Gamma \xrightarrow{S:D} A}{\Gamma \xrightarrow{(U;S):G\supset D} A}\ \mathrm{fcs} \supset$$

Figure 4. Proof terms for focused uniform proofs

We modify the main provability judgments to account for proof-terms, unifying Σ and Γ as usual in type theory:

$\Gamma \longrightarrow U : G$	U is a term (proof) of type (goal) G given assumptions Γ
$\Gamma \xrightarrow{D} S : A$	S is a spine (proof) consisting of heads of type (clause) D to terms S of type (goal) A given assumptions Γ.

Of course, once the spine representation was in place, there was still the need to provide an unification algorithm for this language. In [22] the authors fill this gap, providing a judgmental view of a linear pre-unification procedure in the style of Huet. Being a conservative extension of ordinary higher-order unification, it may not terminate and if it does, it returns a system of equations between flexible atoms, possibly yielding infinite numbers of incomparable unifiers. The paper shows also that it is not possible to simulate higher-order linear unification by generating standard higher-order solutions and promoting those which satisfy the linearity constraints. Even more noteworthy, an analogous notion to Miller's intuitionistic higher-order patterns [45], for which mgu's can be effectively found, does not seem to exist in the linear setting.

3.3 LO

Most of the research about linear logic programming as far as LO and Forum are concerned was spearheaded by Giorgio Levi and his school, in their research aiming to integrate (linear) logic programming with other paradigm such as concurrency and object-orientation, beginning with Guglielmi and Delzanno's thesis [28, 38]. The latter then moved to Genoa, where he collaborated with M. Martelli, Bozzano and others.

The relationship between linear and disjunctive LP mentioned in [4] is taken up in [12], where the authors show that LO can be seen as a sub-structural fragments of DLP, where contraction on the right is disallowed. More extensive connections between

a fragment of LO and DLP are further established using abstract interpretation methods [13]. A propositional bottom-up semantics for LO (and its extension with multiplicative unit LO_1) is proposed via a fixed point operator operating on (ideals of) multisets. Note that the semantics is effective for LO, but not for LO_1; the former, in fact, lacks the expressivity of *counting* resources, while in the latter it is possible to encode formalisms such as Petri nets with transfer arcs. Emphasis on the propositional side was also motivated by earlier work on partial evaluation of LO programs [3]. This yielded an approach to model-checking where verifying a safety problem encoded in temporal logic is akin to computing the fixed point of a linear logic program. This is further studied in [14], where bottom-up evaluation is extended to first order LO programs with universally quantified goals and possibly empty heads. See for more details the Chapter [29] in this volume.

We remark that bottom-up evaluation has now gained an important role in general sequent-based automated theorem proving [25, 42], as well as in the operational semantics of LolliMon [43], the first-order logic programming language underlying the Concurrent Logical Framework [66]. The latter integrates Lolli with a *monadic* modality encapsulating synchronous connectives.

3.4 Forum

Some early work exploited the connection between linear logic and multiset rewriting to encode aspects of planning and concurrency [15, 18]. More developed research was concerned with finding a logical counterpart of object-based languages such as the Object Calculus; [16] introduced $\mathbf{Ob}_{\multimap}$, an object language where methods are represented as logical formulae and whose operational semantics is realized via proof search. The language is then encoded in a *linear* extension of second-order N-Prolog, with a limited form of predicate quantification In [30] the authors present a restriction of Forum with the aim of integrating logic programming with the rewrite-based specification languages; intended applications are modelling of concurrent systems and metaprogramming. Clauses have the form $G_1 \supset \ldots G_n \supset (\mathfrak{B}\mathcal{A} \circ\!\!- G)$ and may again incorporate a form of predicate quantification, provided the underlying term language is basically first-order. State-based computations are specified similarly as in LO, i.e. storing resources on the right-hand side of the sequent and matching them with multi-headed clauses.

4 Conclusions

We have tried to show how the proof-theoretic approach to LP has led to a series of logically motivated logic programming languages of increasing power, supporting modern abstraction mechanisms via higher-order extensions and imperative features via resource-consciousness. The Italian contribution has been both foundational and applicative, in terms of language extensions, implementation techniques and usage of the new features to capture various computation models. We cannot leave out, however, that the original emphasis on endowing logic programming with some of the more successful features of functional programming has died down or, better, it has changed

emphasis. Indeed, the design of LolliMon is heavily influenced by Moggi's computational monads, which are omnipresent in functional languages such as Haskell. What has thrived, beyond a better understanding of the foundations of proof search that is showing promising fruits in general theorem proving, is the theory and practice of logical frameworks. We argue that this development from logical representation to meta-reasoning over the latter is a natural and welcomed one, which could not have happened without the proof-theoretical standpoint. We can isolate two trends in which Italian researchers have an active role:

1. The development of more expressive type-theoretic frameworks, from linear [23] to concurrent ones [65, 66].
2. The integration of HOAS and principle of (co)induction, both in standard systems [54] and in ones directly derived from logic programming such as the *Bedwyr* model-checker [8] and the *Abella* interactive theorem prover [36], see [55] for work on their logical foundations.

Acknowledgments This survey owes to many of Miller's papers, especially "An Overview of Linear Logic Programming" [50]. We thank Iliano Cervesato and Laura Giordano for bibliographic suggestions and the anonymous referees for many useful remarks.

References

1. María Alpuente and Maria I. Sessa, editors. *1995 Joint Conference on Declarative Programming, GULP-PRODE'95, Marina di Vietri, Italy*, 1995.
2. Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.
3. Jean-Marc Andreoli, Tiziana Castagnetti, and Remo Pareschi. Abstract interpretation of linear logic programming. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 295–314, Vancouver, Canada, October 1993. MIT Press.
4. Jean-Marc Andreoli and Remo Pareschi. LO and behold! Concurrent structured processes. In *Proceedings of OOPSLA'90*, pages 44–56, Ottawa, Canada, October 1990. Published as ACM SIGPLAN Notices, vol.25, no.10.
5. Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.
6. Francesca Arcelli. *Aspetti di ordine superiore e di metalivello della programmazione logica*. PhD thesis, DSI, Universitá di Milano, 1991.
7. Francesca Arcelli and Ferrante Formato. Implementing higher-order term-rewriting for program transformation in λProlog. In Alpuente and Sessa [1], pages 245–256.
8. David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 391–397. Springer, 2007.
9. Matteo Baldoni, Laura Giordano, and Alberto Martelli. A modal extension of logic programming: Modularity, beliefs and hypothetical reasoning. *J. Log. Comput.*, 8(5):597–635, 1998.
10. Roberto Barbuti, Paolo Mancarella, Dino Pedreschi, and Franco Turini. A transformational approach to negation in logic programming. *Journal of Logic Programming*, 8:201–228, 1990.

11. Annalisa Bossi and Maria Chiara Meo. *Theoretical Foundations and Semantics*, chapter 2. Volume 6000 of Dovier and Pontelli [31], 2010.
12. Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. On the relations between disjunctive and linear logic programming. *Electr. Notes Theor. Comput. Sci.*, 48, 2001.
13. Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. An effective fixpoint semantics for linear logic programs. *Theory Pract. Log. Program.*, 2(1):85–122, 2002.
14. Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. Model checking linear logic specifications. *TPLP*, 4(5-6):573–619, 2004.
15. Paola Bruscoli and Alessio Guglielmi. Expressiveness of the abstract logic programming language Forum in planning and concurrency. In María Alpuente, Roberto Barbuti, and Isidro Ramos, editors, *GULP-PRODE (2)*, pages 221–237, 1994.
16. Michele Bugliesi, Giorgio Delzanno, Luigi Liquori, and Maurizio Martelli. Object calculi in linear logic. *J. Log. Comput.*, 10(1):75–104, 2000.
17. Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *J. Log. Program.*, 19/20:443–502, 1994.
18. Iliano Cervesato. Petri nets and linear logic: a case study for logic programming. In Alpuente and Sessa [1], pages 313–320.
19. Iliano Cervesato. Proof-theoretic foundation of compilation in logic programming languages. In J. Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 115–129, Manchester, UK, June 1998. MIT Press.
20. Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81, Leipzig, Germany, March 1996. Springer-Verlag LNAI 1050.
21. Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Extended version of [20].
22. Iliano Cervesato and Frank Pfenning. Linear higher-order pre-unification. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Sumposium on Logic in Computer Science (LICS'97)*, pages 422–433, Warsaw, Poland, June 1997. IEEE Computer Society Press.
23. Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 1998. Special issue with invited papers from LICS'96, E. Clarke, editor.
24. Iliano Cervesato and Frank Pfenning. A linear spine calculus. *J. Log. Comput.*, 13(5):639–688, 2003.
25. Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *J. Autom. Reasoning*, 40(2-3):133–177, 2008.
26. Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, May 1995.
27. Stefania Costantini and Gaetano Aurelio Lanzarone. A metalogic programming language. In *ICLP*, pages 218–233, 1989.
28. Giorgio Delzanno. *Logic and Object-Oriented Programming in Linear Logic*. PhD thesis, Universitá di Pisa, February 1997.
29. Giorgio Delzanno, Roberto Giacobazzi, and Francesco Ranzato. *Analysis, Abstract Interpretation, and Verification in (Constraint Logic) Programming*, chapter 10. Volume 6000 of Dovier and Pontelli [31], 2010.
30. Giorgio Delzanno and Maurizio Martelli. Proofs as computations in linear logic. *Theoretical Computer Science*, 258(1-2):269–297, 2001.
31. Agostino Dovier and Enrico Pontelli, editors. *Twenty-five Years of Logic Programming in Italy*, volume 6000 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

32. Amy P. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *J. Autom. Reasoning*, 11(1):41–81, 1993.
33. Dov M. Gabbay. N-Prolog: An extension of Prolog with hypothetical implication II - logical foundations, and negation as failure. *J. Log. Program.*, 2(4):251–283, 1985.
34. Dov M. Gabbay and Uwe Reyle. N-Prolog: An extension of Prolog with hypothetical implications I. *J. Log. Program.*, 1(4):319–355, 1984.
35. Maurizio Gabbrielli, Catuscia Palamidessi, and Frank D. Valencia. *Concurrent and Reactive Constraint Programming*, chapter 12. Volume 6000 of Dovier and Pontelli [31], 2010.
36. Andrew Gacek. The Abella interactive theorem prover (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008.
37. Laura Giordano and Nicola Olivetti. Combining negation as failure and embedded implications in logic programs. *J. Log. Program.*, 36(2):91–147, 1998.
38. Alessio Guglielmi. *Abstract Logic Programming in Linear LogicIndependence and Causality in a First Order Calculus*. PhD thesis, Universitá di Pisa, April 1996.
39. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
40. Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.
41. Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.
42. Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46), 2009.
43. Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Pedro Barahona and Amy P. Felty, editors, *PPDP*, pages 35–46. ACM, 2005.
44. Dale Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.
45. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Proceedings of the International Workshop on Extensions of Logic Programming*, pages 253–281, Tübingen, Germany, 1989. Springer-Verlag LNAI 475.
46. Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, January 1989.
47. Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.
48. Dale Miller. A proposal for modules in λProlog. In R. Dyckhoff, editor, *Proceedings of the 4th International Workshop on Extensions to Logic Programming*. Springer-Verlag LNAI 798, 1993.
49. Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, September 1996.
50. Dale Miller. Overview of linear logic programming. In Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Phil Scott, editors, *Linear Logic in Computer Science*, volume 316 of *London Mathematical Society Lecture Note*, pages 119–150. Cambridge University Press, 2004.
51. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
52. Alberto Momigliano. Minimal negation and Hereditary Harrop Formulae. In *LFCS*, pages 326–335, 1992.

53. Alberto Momigliano. Elimination of negation in a logical framework. In Peter Clote and Helmut Schwichtenberg, editors, *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 411–426. Springer, 2000.
54. Alberto Momigliano and Simon Ambler. Multi-level meta-reasoning with higher-order abstract syntax. In *FoSSaCS*, pages 375–391, 2003.
55. Alberto Momigliano and Alwen Fernanto Tiu. Induction and co-induction in sequent calculus. In *TYPES*, pages 293–308, 2003.
56. Gopalan Nadathur. Correspondences between classical, intuitionistic and uniform provability. *Theoretical Computer Science*, 232:273–298, 2000.
57. Gopalan Nadathur. The metalanguage λProlog and its implementation. In Herbert Kuchen and Kazunori Ueda, editors, *FLOPS*, volume 2024 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2001.
58. Remo Pareschi. *Type-Driven Natural Language Analysis*. PhD thesis, University of Edinburgh, July 1989. University of Pennsylvania, Department of Computer and Information Science, Technical Report No. MS-CIS-89-45.
59. Frank Pfenning. Computation and deduction. Unpublished lecture notes, 217 pp. Revised March 2001, May 1992.
60. Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999.
61. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.
62. Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
63. Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie-Mellon University, 2000. CMU-CS-00-146.
64. O. H. D. Warren. Higher-order extensions to Prolog: Are they needed? In J. E. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence*, volume 10, pages 441–454. Halsted Press, 1982.
65. Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2003.
66. Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying properties of concurrent computations in CLF. *Electr. Notes Theor. Comput. Sci.*, 199:67–87, 2008.