1
An Overview of Linear Logic Programming

Dale Miller
INRIA/Futurs & Laboratoire d'Informatique (LIX)
École polytechnique, Rue de Saclay
91128 PALAISEAU Cedex FRANCE

Abstract

Logic programming can be given a foundation in sequent calculus by viewing computation as the process of building a cut-free sequent proof bottom-up. The first accounts of logic programming as *proof search* were given in classical and intuitionistic logic. Given that linear logic allows richer sequents and richer dynamics in the rewriting of sequents during proof search, it was inevitable that linear logic would be used to design new and more expressive logic programming languages. We overview how linear logic has been used to design such new languages and describe briefly some applications and implementation issues for them.

1.1 Introduction

It is now commonplace to recognize the important role of logic in the foundations of computer science. When a major new advance is made in our understanding of logic, we can thus expect to see that advance ripple into many areas of computer science. Such rippling has been observed during the years since the introduction of linear logic by Girard in 1987 [Gir87]. Since linear logic embraces computational themes directly in its design, it often allows direct and declarative approaches to computational and resource sensitive specifications. Linear logic also provides new insights into the many computational systems based on classical and intuitionistic logics since it refines and extends these logics.

There are two broad approaches by which logic, via the theory of proofs, is used to describe computation [Mil93]. One approach is the *proof reduction* paradigm, which can be seen as a foundation for *func-*

tional programming. Here, programs are viewed as natural deduction or sequent calculus proofs and computation is modeled using proof normalization. Sequents are used to type a functional program: a program fragment is associated with the *single-conclusion* sequent $\Delta \longrightarrow G$ if the code has the type declared in G when all its free variables have types declared for them in the set of type judgments Δ. Abramsky [Abr93] has extended this interpretation of computation to *multiple-conclusion* sequents of linear logic, $\Delta \longrightarrow \Gamma$, where Δ and Γ are both multisets of typing judgments. In that setting, cut-elimination can be seen as specifying concurrent computations. See also [BS94, Laf89, Laf90] for related uses of concurrency in proof normalization in linear logic. The more expressive types made possible by linear logic have also been used to provide static analysis of run-time garbage, aliases, reference counters, and single-threadedness [GH90, MOTW99, O'H91, Wad90].

Another approach to using proof theory to specify computation is the *proof search* paradigm, which can be seen as a foundation for *logic programming.* In this paper (which is an update to [Mil95]), we first provide an overview of the proof search paradigm and then outline the impact that linear logic has made to the design and expressivity of new logic programming languages.

1.2 Goal-directed proof search

When logic programming is considered abstractly, sequents directly encode the state of a computation and the changes that occur to sequents during bottom-up search for cut-free proofs encode the dynamics of computation. In particular, following the framework described in [MNPS91], a logic programming language consists of two kinds of formulas: *program clauses* describe the meaning of non-logical constants and *goals* are the possible consequences considered from collections of program clauses. A single-conclusion sequent $\Delta \longrightarrow G$ represents the state of an idealized logic programming interpreter in which the current logic program is Δ (a set or multiset of formulas) and the goal is G. These two classes of formulas are duals of each other in the sense that a negative subformula of a goal is a program clause and a negative subformula of a program clause is a goal formula.

1.2.1 Uniform proofs

The constants that appear in logical formulas are of two kinds: logical constants (connectives and quantifiers) and non-logical constants (predicates and function symbols). The "search semantics" of the former is fixed and independent of context: for example, the search for the proof of a disjunction or universal quantifier should be the same no matter what program is contained in the sequent for which a proof is required. On the other hand, the instructions for proving a formula with a non-logical constant head (that is, an atomic formula) are provided by the logic program in the sequent.

This separation of constants into logical and non-logical yields two different phases in proof search for a sequent. One phase is that of *goal reduction*, in which the search for a proof of a non-atomic formula uses the introduction rule for its top-level logical constant. The other phase is *backchaining*, in which the meaning of an atomic formula is extracted from the logic program part of the sequent.

The technical notion of *uniform proofs* is used to capture the notion of *goal-directed search*. When sequents are single-conclusion, a *uniform proof* is a cut-free proof in which every sequent with a non-atomic right-hand side is the conclusion of a right-introduction rule [MNPS91]. An interpreter attempting to find a uniform proof of a sequent would directly reflect the logical structure of the right-hand side (the goal) into the proof being constructed. As we shall see, left-introduction rules are used only when the goal formula is atomic and as part of the backchaining phase.

A specific notion of goal formula and program clause along with a proof system is called an *abstract logic programming language* if a sequent has a proof if and only if it has a uniform proof. As we shall illustrate below, first-order and higher-order variants of Horn clauses paired with classical provability [NM90] and hereditary Harrop formulas paired with intuitionistic provability [MNPS91] are two examples of abstract logic programming languages.

While backchaining is not part of the definition of uniform proofs, the structure of backchaining is consistent across several abstract logic programming languages. In particular, when proving an atomic goal, applications of left-introduction rules can be used in a coordinated decomposition of a program clause that yields not only a matching atomic formula occurrence to the atomic goal but also possibly new goals formulas for which additional proofs must be attempted [NM90, MNPS91, HM91].

1.2.2 Logic programming in classical and intuitionistic logics

In the beginning of the logic programming literature, there was one example of logic programming, namely, the first-order classical theory of Horn clauses, which was the logic basis of the popular programming language Prolog. However, no general framework existed for connecting logic and logic programming. The operational semantics of logic programs was presented as resolution [AvE82], an inference rule optimized for classical reasoning. Miller and Nadathur [MN86, Mil86, NM90] were probably the first to use the sequent calculus to examine design and correctness issues for logic programming languages. Moving to the sequent calculus made it nature to consider logic programming in settings other than just classical logic.

We first consider the design of logic programming languages within classical and intuitionistic logic, where the logical constants are taken to be *true*, $\wedge$, $\vee$, $\supset$, $\forall$, and $\exists$ (false and negation are not part of the first logic programs we consider).

Horn clauses can be defined simply as those formulas built from *true*, $\wedge$, $\supset$, and $\forall$ with the proviso that no implication or universal quantifier is to the left of an implication. A goal in this setting would then be any negative subformula of a Horn clause: more specifically, they would be either *true* or a conjunction of atomic formulas. It is shown in [NM90] that a proof system similar to the one in Figure 1.1 is complete for the classical logic theory of Horn clauses and their associated goal formulas. It then follows immediately that Horn clauses are an abstract logic programming language. (The syntactic variable A in Figure 1.1 denotes atomic formulas.) Notice that sequents in this and other proof systems contain a *signature* Σ as its first element: this signature contains type declarations for all the non-logical constants in the sequent. Notice also that there are two different kinds of sequent judgments: one with and one without a formula on top of the sequent arrow. The sequent $\Sigma : \Delta \xrightarrow{D} A$ denotes the sequent $\Sigma : \Delta, D \longrightarrow A$ but with the D formula being distinguished (that is, marked for backchaining).

Inference rules in Figure 1.1, and those that we shall show in subsequent proof systems, can be divided into four categories. The right-introduction rules (goal-reduction) are those using the unlabeled sequent arrow and in which the goal formula is non-atomic. The left-introduction rules (backchaining) are those with sequent arrows labeled with a formula and it is on that formula that introduction rules are applied. The initial rule forms the third category and is the only rule with a repeated

$$\frac{}{\Sigma : \Delta \longrightarrow true} \qquad \frac{\Sigma : \Delta \longrightarrow G_1 \qquad \Sigma : \Delta \longrightarrow G_2}{\Sigma : \Delta \longrightarrow G_1 \wedge G_2}$$

$$\frac{\Sigma : \Delta \xrightarrow{D} A}{\Sigma : \Delta \longrightarrow A}\ decide \qquad \frac{}{\Sigma : \Delta \xrightarrow{A} A}\ initial \qquad \frac{\Sigma : \Delta \xrightarrow{D_i} A}{\Sigma : \Delta \xrightarrow{D_1 \wedge D_2} A}$$

$$\frac{\Sigma : \Delta \longrightarrow G \qquad \Sigma : \Delta \xrightarrow{D} A}{\Sigma : \Delta \xrightarrow{G \supset D} A} \qquad \frac{\Sigma : \Delta \xrightarrow{D[t/x]} A}{\Sigma : \Delta \xrightarrow{\forall_\tau x.D} A}$$

Fig. 1.1. In the decide rule, $D \in \Delta$; in the left rule for $\wedge$, $i \in \{1, 2\}$; and in the left rule for $\forall$, t is a Σ-term of type τ.

$$\frac{\Sigma : \Delta, D \longrightarrow G}{\Sigma : \Delta \longrightarrow D \supset G} \qquad \frac{\Sigma, c{:}\tau : \Delta \longrightarrow G[c/x]}{\Sigma : \Delta \longrightarrow \forall_\tau x.G}$$

Fig. 1.2. The rule for universal quantification has the proviso that c is not declared in Σ.

occurrence of a schema variable in its conclusion. The decide rule forms the forth and final category: this rule is responsible for moving a formula from the logic program to above the sequent arrow.

In this proof system, left-introductions are now applied only on the formula annotating the sequent arrow. The usual notion of backchaining can be seen as an instance of a decide rule, which places a formula from the program (the left-hand context) on top of the sequent arrow, and then a sequence of left-introductions work on that distinguished formula. Backchaining ultimately performs a linking between a goal formula and a program clause via the repeated schema variable in the initial rule. In Figure 1.1, there is one decide rule and one initial rule: in a subsequent inference system, there are more of each category. Also, proofs in this system involving Horn clauses have a simple structure: all sequents in a given proof have identical left hand sides: signatures and programs are fixed and global during the search for a proof. If changes in sequents are meant to be used to encode dynamics of computation, then Horn clauses provide a weak start: the only dynamics are changes in goals which relegates such dynamics entirely to the non-logical domain of atomic formulas. As we illustrate with an example in Section 1.6, if one can use a logic programming language where sequents have more dynamics, then one can reason about some aspects of logic programs directly using logical tools.

Hereditary Harrop formulas can be presented simply as those formulas built from *true*, $\wedge$, $\supset$, and $\forall$ with no restrictions. Goal formulas, *i.e.*, negative subformulas of such formulas, would thus have the same structure. It is shown in [MNPS91] that a proof system similar to the one formed by adding to the inference rules in Figure 1.1 the rules in Figure 1.2 is complete for the intuitionistic logic theory of hereditary Harrop formulas and their associated goal formulas. It then follows immediately that hereditary Harrop formulas are an abstract logic programming language. The classical logic theory of hereditary Harrop formulas is not, however, an abstract logic programming language: Peirce's formula $((p \supset q) \supset p) \supset p$, for example, is classically provable but has no uniform proof. The original definition of hereditary Harrop formulas permitted disjunctions and existential quantifiers at the top-level of goal formulas. Such an extension makes little change to the logic's proof theory properties but does help to justify its name since all positive subformulas of program clauses are then Harrop formulas [Har60].

Notice that sequents in this new proof system have a slightly greater ability to change during proof search: in particular, both signatures and programs can increase as proof search moves upward. Thus, not all constants and program clauses need to be available at the beginning of a computation: instead they can be made available as search continues. For this reason, the hereditary Harrop formulas have been used to provide logic programming with approaches to modular programming [Mil89b] and abstract datatypes [Mil89a, NJK95].

1.2.3 Higher-order quantification and proof search

The impact of using higher-order quantification in proof search was systematically studied in the contexts of Horn clauses [MN86, Nad87, NM90] and hereditary Harrop formulas [MNPS91, NM98]. The higher-order setting for these studies was done using the subset of Church's Simple Theory of Types [Chu40] in which the "mathematical axioms" of extensionality, infinity, choice, etc, are not assumed.

Allowing quantification of variables of functional types only (that is, not at predicate type) is not a challenge for the high-level treatment of proof search. Such an extension to the first-order setting does make logic programming much more expressive and more challenging to implement. In particular, the presence of quantification at function types and of simply typed λ-terms [Chu40] endowed logic programming with the encoding technique called *higher-order abstract syntax* [MN87, HM88, PE88]. It was, in fact, the λProlog programming language [NM88] in which this style programming was first supported.

Allowing quantification at predicate type does provide some significant challenges to the proof theoretical analysis of proof search: we illustrate two such issues here.

One issue with predicate quantification is that during proof search, the careful restriction to having program clauses on the left of the sequent arrow and goal formulas on the right might be broken via higher-order instantiation with terms containing logical connectives. For example, consider a logic program containing the following two clauses:

$$\forall P[P\ a \supset q\ b] \quad \text{and} \quad \forall x[q\ x \supset r].$$

Here, the first clause is a higher-order Horn clause following the definition in [NM90]. If we take an instance of this logic program in which P in the first clause is instantiated by $\lambda w.\neg q\ w$, we have clauses logically equivalent to

$$[q\ a \vee q\ b] \quad \text{and} \quad \forall x[q\ x \supset r].$$

Notice that with respect to this second logic program the atomic goal r has a classical logic proof but does not have a uniform proof. Thus, the instance of a higher-order Horn clause does not necessarily result in another higher-order Horn clause. Fortunately, for both the theory of higher-order Horn clauses and higher-order hereditary Harrop formulas, it is possible to prove that the only higher-order instances that are required during proof search are those that preserve the invariance of the initial syntactic restriction to Horn clauses or hereditary Harrop formulas [MNPS91, NM90].

A second issue is more related the operational reading of clauses: program clauses are generally seen as contributing meaning to specific predicates, such as those that, for example, define the relations of concatenation or sorting of lists. These predicate constants have occurrences at strictly positive positions within program clauses: such a positive occurrence is called a *head* of a clause. If one allows predicate variables instead of constants in such head positions, then in a sense, such program clauses would be contributing meaning to any predicate. For this reason, head symbols are generally restricted to be constants. If all head symbols in a logic program are constant, it is also easy to show that that logic program is consistent (that is, some formulas are not deducible from it).

If certain mild restrictions are placed on the occurrences of logical connectives within the scope of non-logical constants (that is, within atoms), then higher-order variants of Horn clauses and hereditary Harrop formulas are known to be abstract logic programming languages [NM90, MNPS91]. Higher-order quantification of predicates can provide logic programming specifications with direct and natural ways to do higher-order programming, as is popular in functional programming languages, as well as providing a means of lexically scoping and hiding predicates [Mil89a].

Allowing higher-order head positions does have, at least, a theoretical interest. Full first-order intuitionistic logic is not an abstract logic programming language since both $\vee$ and $\exists$ can cause incompleteness of uniform proofs. For example, both

$$p \vee q \longrightarrow q \vee p \quad \text{and} \quad \exists x.B \longrightarrow \exists x.B$$

have intuitionistic proofs but neither sequent has a uniform proof. As we have seen above, eliminating disjunction and existential quantification yields immediately abstract logic programming languages (at least within intuitionistic logic). As is well known, higher-order quantification allows one to define the intuitionistic disjunction $B \vee C$ as $\forall p((B \supset p) \supset (C \supset p) \supset p)$ and the existential quantifier $\exists x.Bx$ as $\forall p((\forall x.Bx \supset p) \supset p)$. Both of these formulas have the predicate variable p in a head position. Notice that if the two sequents displayed above are rewritten using these two definitions, the resulting sequents would have uniform proofs. Felty has shown that higher-order intuitionistic logic based on $true$, $\wedge$, $\supset$, and $\forall$ for all higher-order types (with no restriction of predicate variable occurrences) is an abstract logic programming language [Fel93].

1.2.4 Uniform proofs with multiple conclusion sequents

In the multiple-conclusion setting, goal-reduction should continue to be independent not only from the logic program but also from other goals, *i.e.*, multiple goals should be reducible simultaneously. Although the sequent calculus does not directly allow for simultaneous rule application, it can be simulated easily by referring to permutations of inference rules [Kle52]. In particular, we can require that if two or more right-introduction rules can be used to derive a given sequent, then all possible orders of applying those right-introduction rules can be obtained from any other order simply by permuting right-introduction inferences. It is easy to see that the following definition of uniform proofs for multiple-conclusion sequents generalizes that for single-conclusion sequents: a cut-free, sequent proof Ξ is *uniform* if for every subproof Ψ of Ξ and for every non-atomic formula occurrence B in the right-hand side of the end-sequent of Ψ, there is a proof Ψ' that is equal to Ψ up to permutation of inference rules and is such that the last inference rule in Ψ' introduces the top-level logical connective occurring in B [Mil93, Mil96]. The notion of abstract logic programming language can be extended to the case where this extended notion of uniform proof is complete. As evidence of the usefulness of this definition, Miller in [Mil93] used it to specify a π-calculus-like process calculus in linear logic and showed that it was an abstract logic programming language in this new sense.

Given this definition of uniform proofs for multiple conclusion sequent calculus, an interesting next step would be to turn to linear logic and start to identify subsets for which goal directed search is complete and to identify backchaining rules. Fortunately and surprisingly, the work of Andreoli in his PhD thesis [And90] on focused proof search for linear logic provides a complete analysis along these lines for all of linear logic.

1.3 Linear logic and focused proofs

As we have seen, the goal-directed proof search analysis of logic programming in classical and intuitionistic logic revealed three general observations: (1) Two sets of formulas can be identified for use as goals and as program clauses. (2) These two classes are duals of each other, at least in the sense that a negative subformula of a formula in one class is a formula in the other class. (3) Goal formulas are processed immediately by a sequence of invertible right-rules and program clauses are used via a focused application of left-rules know as backchaining.

Andreoli analyzed the structure of proof search in linear logic using the notion of *focused proof* [And90, And92]. His analysis made it possible to extend the above three observations to all of linear logic and to provide a deep and elegant explanation for why they hold. Andreoli classified the logical connectives into two sets of connectives. *Asynchronous* connectives are those whose right-introduction rule is invertible and *synchronous* connectives are those whose right-introduction is not invertible; that is, the success of applying a right-introduction rule for a synchronous connective required information from the context. (We say that a formula is asynchronous or synchronous depending on the top-level connective of the formula.) He also observed that these two classes of connectives are de Morgan duals of each other.

Given these distinctions between formulas, Andreoli showed that a complete bottom-up proof search procedure for cut-free proofs in linear logic (using one-sided sequents) can be described roughly as follows: first decompose all asynchronous formulas and when none remain, pick some synchronous formula, introduce its top-level connective and then continue decomposing all synchronous subformulas that might arise. Thus interleaving between asynchronous reductions and synchronous reductions yields a highly normalized proof search mechanism. Proofs built in this fashion are called *focused proofs.*

A consequence of this completeness of focused proofs is that all of linear logic can be seen as logic programming, at least once we choose the proper presentation of linear logic. In such a presentation, focused proofs capture the notion of uniform proofs and backchaining at the same time. Since all of linear logic can be seen as logic programming, we delay presenting more details about focused proofs until the next section where we present several linear logic programming languages.

1.4 Linear logic programming languages

We now present the designs of some linear logic programming languages. Our first language, Forum, provides a basis for considering all of linear logic as logic programming. We shall also look at certain subsets of Forum since they will allow us to focus on particular structural features of proof search and particular application areas.

1.4.1 The Forum presentation of linear logic

The logic programming languages based on classical and intuitionistic logics considered earlier used the connectives *true*, $\wedge$, $\supset$, and $\forall$. We shall now consider a presentation of linear logic using the corresponding connectives, namely, $\top$, $\&$, $\Rightarrow$, and $\forall$, along with the distinctly linear connectives $\multimap$, $\bot$, $⅋$, and $?$. Together, this collection of connectives yields a presentation of all of linear logic since the missing connectives are directly definable using the following logical equivalences.

$$B^\perp \equiv B \multimap \bot \qquad 0 \equiv \top \multimap \bot \qquad 1 \equiv \bot \multimap \bot \qquad \exists x.B \equiv (\forall x.B^\perp)^\perp$$
$$!\,B \equiv (B \Rightarrow \bot) \multimap \bot \qquad B \oplus C \equiv (B^\perp \& C^\perp)^\perp \qquad B \otimes C \equiv (B^\perp ⅋ C^\perp)^\perp$$

This collection of connectives is not minimal: for example, ? and $⅋$, can be defined in terms of the remaining connectives

$$?\,B \equiv (B \multimap \bot) \Rightarrow \bot \quad \text{and} \quad B ⅋ C \equiv (B \multimap \bot) \multimap C.$$

Unlike many treatments of linear logic, we shall treat $B \Rightarrow C$ as a logical connective (which corresponds to $!\,B \multimap C$). From the proof search point-of-view, the four intuitionistic connectives *true*, $\wedge$, $\supset$, and $\forall$ correspond naturally with the four linear logic connectives $\top$, $\&$, $\Rightarrow$, and $\forall$ (in fact, the correspondence is so strong for the quantifiers that we write them the same in both settings). We shall call this particular presentation of linear logic the *Forum* presentation of linear logic or simply *Forum*.

Notice that all the logical connectives used in Forum are asynchronous: that is, their right-introduction rules are invertible. Since we are using two sided sequents, asynchronous formulas have a synchronous behavior when they are introduced on the left of the sequent arrow. Thus, goal reduction correspondences to the reduction of asynchronous connectives and backchaining correspondences to the focused decomposition of synchronous connectives (via left-introduction rules).

The proof systems in Figures 1.1 and 1.2 that describe logic programming in classical and intuitionistic logic used two styles of sequents: $\Sigma : \Delta \ \longrightarrow G$ and $\Sigma : \Delta \xrightarrow{D} A$, where Δ is a set of formulas. These sequent judgments are generalized here to $\Sigma : \Psi; \Delta \longrightarrow \Gamma; \Upsilon$ (for goal-reduction) and $\Sigma : \Psi; \Delta \xrightarrow{D} \mathcal{A}; \Upsilon$ (for backchaining), where Ψ and Υ are sets of formulas (classical maintenance), Δ and Γ are multisets of formulas (linear maintenance), $\mathcal{A}$ is a multiset of atomic formulas, and D is a formula. Notice that placement of the linear context next to the sequent arrow and classical context away from the arrow is standard notation in the literature of linear logic programming, but is the opposite convention used by Girard in his LU proof system [Gir93].

The focusing result of Andreoli [And92] can be formulated [Mil96] as the completeness of the proof system for linear logic using the proof system in Figure 1.3. This proof system appears rather complicated at first glance, so it is worth noting the following organization of these inference rules: there are 8 right-introduction rules, 7 left-introduction rules, 2 initial rules, and 3 decide rules. Notice that 2 of the decide rules place a formula on the sequent arrow while the third copies of formula from the classically maintained right context to the linear maintained right context. This third decide rule is a combination of contraction and

$$\frac{}{\Sigma:\Psi;\Delta\longrightarrow\top,\Gamma;\Upsilon}\qquad\frac{\Sigma:\Psi;\Delta\longrightarrow B,\Gamma;\Upsilon\quad\Sigma:\Psi;\Delta\longrightarrow C,\Gamma;\Upsilon}{\Sigma:\Psi;\Delta\longrightarrow B\mathbin{\&}C,\Gamma;\Upsilon}$$

$$\frac{\Sigma:\Psi;\Delta\longrightarrow\Gamma;\Upsilon}{\Sigma:\Psi;\Delta\longrightarrow\bot,\Gamma;\Upsilon}\qquad\frac{\Sigma:\Psi;\Delta\longrightarrow B,C,\Gamma;\Upsilon}{\Sigma:\Psi;\Delta\longrightarrow B\parr C,\Gamma;\Upsilon}$$

$$\frac{\Sigma:\Psi;B,\Delta\longrightarrow C,\Gamma;\Upsilon}{\Sigma:\Psi;\Delta\longrightarrow B\multimap C,\Gamma;\Upsilon}\qquad\frac{\Sigma:B,\Psi;\Delta\longrightarrow C,\Gamma;\Upsilon}{\Sigma:\Psi;\Delta\longrightarrow B\Rightarrow C,\Gamma;\Upsilon}$$

$$\frac{y:\tau,\Sigma:\Psi;\Delta\longrightarrow B[y/x],\Gamma;\Upsilon}{\Sigma:\Psi;\Delta\longrightarrow\forall_\tau x.B,\Gamma;\Upsilon}\qquad\frac{\Sigma:\Psi;\Delta\longrightarrow\Gamma;B,\Upsilon}{\Sigma:\Psi;\Delta\longrightarrow\,?B,\Gamma;\Upsilon}$$

$$\frac{\Sigma:B,\Psi;\Delta\xrightarrow{B}\mathcal{A};\Upsilon}{\Sigma:B,\Psi;\Delta\longrightarrow\mathcal{A};\Upsilon}\qquad\frac{\Sigma:\Psi;\Delta\xrightarrow{B}\mathcal{A};\Upsilon}{\Sigma:\Psi;B,\Delta\longrightarrow\mathcal{A};\Upsilon}\qquad\frac{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},B;B,\Upsilon}{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A};B,\Upsilon}$$

$$\frac{}{\Sigma:\Psi;\cdot\xrightarrow{A}A;\Upsilon}\qquad\frac{}{\Sigma:\Psi;\cdot\xrightarrow{A}\cdot;A,\Upsilon}$$

$$\frac{}{\Sigma:\Psi;\cdot\xrightarrow{\bot}\cdot;\Upsilon}\qquad\frac{\Sigma:\Psi;\Delta\xrightarrow{G_i}\mathcal{A};\Upsilon}{\Sigma:\Psi;\Delta\xrightarrow{G_1\&G_2}\mathcal{A};\Upsilon}\qquad\frac{\Sigma:\Psi;B\longrightarrow\cdot;\Upsilon}{\Sigma:\Psi;\cdot\xrightarrow{?B}\cdot;\Upsilon}$$

$$\frac{\Sigma:\Psi;\Delta_1\xrightarrow{B}\mathcal{A}_1;\Upsilon\quad\Sigma:\Psi;\Delta_2\xrightarrow{C}\mathcal{A}_2;\Upsilon}{\Sigma:\Psi;\Delta_1,\Delta_2\xrightarrow{B\parr C}\mathcal{A}_1,\mathcal{A}_2;\Upsilon}\qquad\frac{\Sigma:\Psi;\Delta\xrightarrow{B[t/x]}\mathcal{A};\Upsilon}{\Sigma:\Psi;\Delta\xrightarrow{\forall_\tau x.B}\mathcal{A};\Upsilon}$$

$$\frac{\Sigma:\Psi;\Delta_1\longrightarrow\mathcal{A}_1,B;\Upsilon\quad\Sigma:\Psi;\Delta_2\xrightarrow{C}\mathcal{A}_2;\Upsilon}{\Sigma:\Psi;\Delta_1,\Delta_2\xrightarrow{B\multimap C}\mathcal{A}_1,\mathcal{A}_2;\Upsilon}$$

$$\frac{\Sigma:\Psi;\cdot\longrightarrow B;\Upsilon\quad\Sigma:\Psi;\Delta\xrightarrow{C}\mathcal{A};\Upsilon}{\Sigma:\Psi;\Delta\xrightarrow{B\Rightarrow C}\mathcal{A};\Upsilon}$$

Fig. 1.3. A proof system for the Forum presentation of linear logic. The right-introduction rule for $\forall$ has the proviso that y is not declared in the signature Σ, and the left-introduction rule for $\forall$ has the proviso that t is a Σ-term of type τ. In left-introduction rule for &, $i\in\{1,2\}$.

dereliction rule for ? and is used to "decide" on a new goal formula on which to do reductions.

Because linear logic can be seen as the logic behind classical and intuitionistic logic, it is possible to see both Horn clauses and hereditary Harrop formulas as subsets of Forum. It is a simple matter to see that the proof systems in Figures 1.1 and 1.2 result from restricting the proof system for Forum in Figure 1.3 to Horn clauses and to hereditary Har-

$$\frac{}{\Sigma : \Psi; \Delta \longrightarrow \top} \qquad \frac{\Sigma : \Psi; \Delta \longrightarrow G_1 \qquad \Sigma : \Psi; \Delta \longrightarrow G_2}{\Sigma : \Psi; \Delta \longrightarrow G_1 \,\&\, G_2}$$

$$\frac{\Sigma : \Psi, G_1; \Delta \longrightarrow G_2}{\Sigma : \Psi; \Delta \longrightarrow G_1 \Rightarrow G_2} \quad \frac{\Sigma : \Psi; \Delta, G_1 \longrightarrow G_2}{\Sigma : \Psi; \Delta \longrightarrow G_1 \multimap G_2} \quad \frac{y : \tau, \Sigma : \Psi; \Delta \longrightarrow B[y/x]}{\Sigma : \Psi; \Delta \longrightarrow \forall_\tau x.B}$$

$$\frac{\Sigma : \Psi, D; \Delta \xrightarrow{D} A}{\Sigma : \Psi, D; \Delta \longrightarrow A} \qquad \frac{\Sigma : \Psi; \Delta \xrightarrow{D} A}{\Sigma : \Psi; \Delta, D \longrightarrow A} \qquad \frac{}{\Sigma : \Psi; \cdot \xrightarrow{A} A}$$

$$\frac{\Sigma : \Psi; \Delta \xrightarrow{D_i} A}{\Sigma : \Psi; \Delta \xrightarrow{D_1 \wedge D_2} A} \qquad \frac{\Sigma : \Psi; \cdot \longrightarrow G \qquad \Sigma : \Psi; \Delta \xrightarrow{D} A}{\Sigma : \Psi; \Delta \xrightarrow{G \Rightarrow D} A}$$

$$\frac{\Sigma : \Psi; \Delta_1 \longrightarrow G \qquad \Sigma : \Psi; \Delta_2 \xrightarrow{D} A}{\Sigma : \Psi; \Delta_1, \Delta_2 \xrightarrow{G \multimap D} A} \qquad \frac{\Sigma : \Psi; \Delta \xrightarrow{D[t/x]} A}{\Sigma : \Psi; \Delta \xrightarrow{\forall_\tau x.D} A}$$

Fig. 1.4. The proof system for Lolli. The rule for universal quantification has the proviso that y is not in Σ. In the $\forall$-left rule, t is a Σ-term of type τ.

rop formulas. Below we overview various other subsets of linear logic that have been proposed as specification languages and as abstract logic programming languages.

1.4.2 Lolli

The connectives $\perp$, $⅋$, and ? force the genuinely classical feel of linear logic. (In fact, using the two linear logic equivalences $?B \equiv (B \multimap \perp) \Rightarrow \perp$ and $B ⅋ C \equiv (B \multimap \perp) \multimap C$ we see that we only need to add $\perp$ to a system with the two implication $\multimap$ and $\Rightarrow$ to get full classical linear logic.) Without these three connectives, the multiple-conclusion sequent calculus given for Forum in Figure 1.3 can be replaced by one with only single-conclusion sequents.

The collection of connectives one gets from dropping these three connectives from Forum, namely $\top$, $\&$, $\Rightarrow$, $\multimap$, and $\forall$, form the *Lolli* logic programming language. Presenting a sequent calculus for Lolli is a simple matter. First, remove any inference rule in Figure 1.3 involving $\perp$, $⅋$, and ?. Second, abbreviate the sequents $\Sigma : \Psi; \Delta \longrightarrow G; \cdot$ and $\Sigma : \Psi; \Delta \xrightarrow{D} A; \cdot$ as $\Sigma : \Psi; \Delta \longrightarrow G$ and $\Sigma : \Psi; \Delta \xrightarrow{D} A$. The resulting proof system for Lolli is given in Figure 1.4. The completeness of this proof system for Lolli was given directly by Hodas and Miller in [HM94], although it follows directly from the completeness of focused proofs [And92], at least once focused proofs are written as the Forum proof system.

1.4.3 Uncurrying program clauses

Frequently it is convenient to view a program clause, such as

$$\forall\bar{x}[G_1 \Rightarrow G_2 \multimap A],$$

which contains two goals, as a program clause containing one goal: the formula

$$\forall\bar{x}[(!\,G_1 \otimes G_2) \multimap A].$$

is logically equivalent to the formula above and brings the two goals into the one expression $!\,G_1 \otimes G_2$. Such a rewriting of a formula to a logically equivalent formula is essentially the *uncurrying* of the formula, where uncurrying is the rewriting of formulas using the following equivalences in the forward direction.

$$\begin{array}{rcl} H & \equiv & \mathbf{1} \multimap H \\ B \multimap C \multimap H & \equiv & (B \otimes C) \multimap H \\ B \Rightarrow H & \equiv & !\,B \multimap H \\ (B \multimap H) \,\&\, (C \multimap H) & \equiv & (B \oplus C) \multimap H \\ \forall x.(B(x) \multimap H) & \equiv & (\exists x.B(x)) \multimap H \end{array}$$

(The last equivalence assumes that x is not free in H.) Allowing occurrences of $\mathbf{1}$, $\otimes$, $!$, $\oplus$, and $\exists$ into goals does not cause any problems with the completeness of uniform provability and some presentations of linear logic programming language [HM94, Mil96, PH94] allow for such occurrences.

1.4.4 Other subsets of Forum

Although all of linear logic can be seen as abstract logic programming, it is still of interest to examine subsets of linear logic for use as specification languages or as programming languages. These subsets are often motivated by picking a small subset of linear logic that is expressive enough to specify problems of a certain application domain. Below we list some subsets of linear logic that have been identified in the literature.

If one maps *true* to $\top$, $\wedge$ to $\&$, and $\supset$ to $\Rightarrow$, then both Horn clauses and hereditary Harrop formulas can be identified with linear logic formulas. Proofs given for these two sets of formulas in Figures 1.1 and 1.2

are essentially the same as those for the corresponding proofs in Figure 1.4. Thus, viewing these two classes of formulas as being based on linear instead of intuitionistic logic does not change their expressiveness. In this sense, Lolli can be identified as being hereditary Harrop formulas extended with linear implication. When one is only interested in cut-free proofs, a second translation of Horn clauses and hereditary Harrop formulas into linear logic is possible. In particular, if negative occurrences of *true*, $\wedge$, and $\supset$ are translated to $\mathbf{1}$, $\otimes$, and $\multimap$, respectively, while positive occurrences of *true*, $\wedge$, and $\supset$ are translated to $\top$, $\&$, and $\Rightarrow$, respectively, then the resulting proofs in Figure 1.4 of the linear logic formulas yield proofs similar to those in Figures 1.1 and 1.2 [HM94]. (The notion here of positive and negative occurrences are with respect to occurrences within a cut-free proof: for example, a positive occurrence in a formula on the left of a sequent arrow is judged to be a negative occurrence for this translation.) Thus, if the formula

$$\forall\bar{x}[(A_1 \wedge (A_2 \supset A_3) \wedge A_4) \supset A_0]$$

appears on the left of the sequent arrow, it is translated as

$$\forall\bar{x}[(A_1 \otimes (A_2 \Rightarrow A_3) \otimes A_4) \multimap A_0]$$

and if it appears on the right of the sequent arrow, it is translated as

$$\forall\bar{x}[(A_1 \mathbin{\&} (A_2 \multimap A_3) \mathbin{\&} A_4) \Rightarrow A_0].$$

Historically speaking, the first proposal for a linear logic programming language was LO (Linear Objects) by Andreoli and Pareschi [AP91a, AP91b]. LO is an extension to the Horn clause paradigm in which atomic formulas are generalized to multisets of atomic formulas connected by $⅋$. In LO, backchaining is again multiset rewriting, which was used to specify object-oriented programming and the coordination of processes. LO is a subset of the LinLog [And90, And92], where formulas are of the form

$$\forall\bar{y}(G_1 \hookrightarrow \cdots \hookrightarrow G_m \hookrightarrow (A_1 ⅋ \cdots ⅋ A_p)).$$

Here $p > 0$ and $m \geq 0$; occurrences of $\hookrightarrow$ are either occurrences of $\multimap$ or $\Rightarrow$; $G_1, \ldots G_m$ are built from $\perp$, $⅋$, $?$, $\top$, $\&$, and $\forall$; and $A_1, \ldots A_m$ are atomic formulas. In other words, these are formula in Forum where the "head" of the formula is not empty (*i.e.*, $p > 0$) and where the goals $G_1, \ldots G_m$ do not contain implications. Andreoli argues that arbitrary linear logic formulas can be "skolemize" (by introducing new non-logical constants) to yield only LinLog formulas, such that proof search involving the original and the skolemize formulas are isomorphic. By applying uncurrying, the displayed formula above can be written in the form

$$\forall \bar{y}(G \multimap (A_1 \parr \cdots \parr A_p))$$

where G is composed of the top-level synchronous connectives and of the subformulas $G_1, \ldots, G_m$, which are all composed of asynchronous connectives. In LinLog, goal formulas have no synchronous connective in the scope an asynchronous connective.

1.4.5 Other language designs

Another linear logic programming language that has been proposed is the Lygon system of Harland and Pym [HPW96]. They based their design on notions of goal-directed proof and multiple conclusion uniform proofs [PH94] that unfortunately differ from those presented here. The operational semantics for proof search that they developed is different and more complex than the alternation of asynchronous and synchronous search that is used for, say, Forum.

Let G and H be formulas composed of $\perp$, $\parr$, and $\forall$. Closed formulas of the form $\forall \bar{x}[G \multimap H]$ where H is not (logically equivalent to) $\perp$ have been called *process clauses* in [Mil93] and are used there to encode a calculus similar to the π-calculus: the universal quantifier in goals are used to encode name restriction. These clauses written in the contrapositive (replacing, for example, $\parr$ with $\otimes$) have been called *linear Horn clauses* by Kanovich and have been used to model computation via multiset rewriting [Kan94].

Various other specification logics have also been developed, often designed directly to deal with particular application areas. In particular, the language ACL by Kobayashi and Yonezawa [KY93, KY94] captures simple notions of asynchronous communication by identifying the send and read primitives with two complementary linear logic connectives. Lincoln and Saraswat have developed a linear logic version of concurrent constraint programming [LS93, Sar93] and Fages, Ruet, and Soliman have analyzed similar extensions to the concurrent constraint paradigm [FRS98, RF97].

Some aspects of dependent typed λ-calculi overlap with notions of abstract logic programming languages. Within the setting of intuitionistic, single-side sequents, uniform proofs are similar to $\beta\eta$-long normal forms in natural deduction and typed λ-calculus. The LF logical framework

[HHP93] can be mapped naturally [Fel91] into a higher-order extension of hereditary Harrop formulas [MNPS91]. Inspired such a connection and by the design of Lolli, Cervesato and Pfenning developed a linear extension to LF called Linear LF [CP96, CP02].

1.5 Applications of linear logic programming

One theme that occurs often in applications of linear logic programming is that of *multiset rewriting*: a simple paradigm that has wide applications in computational specifications. To see how such rewriting can be captured in proof search, consider the rewriting rule

$$a, a, b \Rightarrow c, d, e,$$

which specifies that a multiset can be rewritten by first removing two occurrences of a and one occurrence of b and then have one occurrence each of c, d, and e added. Since the left-hand of sequents in Figure 1.4 and the left- and right-hand sides of sequents in Figure 1.3 have multisets of formulas, it is an easy matter to write clauses in linear logic which can rewrite multisets when they are used in backchaining.

To rewrite the right-hand multiset of a sequent using the rule above, simply backchain over the clause $c \parr d \parr e \multimap a \parr a \parr b$. To illustrate such rewriting directly via Forum, consider the sequent $\Sigma : \Psi; \Delta \longrightarrow a, a, b, \Gamma; \Upsilon$ where the above clause is a member of Ψ. A proof for this sequent can then look like the following (where the signature Σ is not displayed).

$$\dfrac{\dfrac{\dfrac{\Psi;\Delta \longrightarrow c,d,e,\Gamma;\Upsilon}{\Psi;\Delta \longrightarrow c \parr d \parr e,\Gamma;\Upsilon} \qquad \dfrac{\dfrac{}{\Psi;\cdot \xrightarrow{a} a;\Upsilon} \quad \dfrac{}{\Psi;\cdot \xrightarrow{a} a;\Upsilon} \quad \dfrac{}{\Psi;\cdot \xrightarrow{b} b;\Upsilon}}{\Psi;\cdot \xrightarrow{a \parr a \parr b} a,a,b;\Upsilon}}{\Psi;\Delta \xrightarrow{c \parr d \parr e \multimap a \parr a \parr b} a,a,b,\Gamma;\Upsilon}}{\Psi;\Delta \longrightarrow a,a,b,\Gamma;\Upsilon}$$

We can interpret this fragment of a proof as a rewriting of the multiset a, a, b, Γ to the multiset c, d, e, Γ using the rule displayed above.

To rewrite the left-hand context instead, a clause such as

$$a \multimap a \multimap b \multimap (c \multimap d \multimap e \multimap A_1) \multimap A_0$$

or (using the uncurried form)

$$(a \otimes a \otimes b) \otimes ((c \otimes d \otimes e) \multimap A_1) \multimap A_0$$

can be used in backchaining. Operationally this clause means that to prove the atomic goal A_0, first remove two occurrence of a and one of b from the left-hand multiset, then add one occurrence each of c, d, and e, and then proceed to attempt a proof of A_1.

Of course, there are additional features of linear logic than can be used to enhance this primitive notion of multiset rewriting. For examples, the ? modal on the right and the ! modal on the left can be used to place items in multisets than cannot be deleted and the additive conjunction & can be used to copy multisets.

Listed below are some application areas where proof search and linear logic have been used. A few representative references for each area are listed.

Object-oriented programming Capturing inheritance was a goal of the early LO system [AP91b] and modeling state encapsulation was a motivation [HM90] for the design of Lolli. State encapsulation was also addressed using Forum in [DM95, Mil94].

Concurrency Linear logic is often been considered a promising declarative foundation for concurrency primitives in specification languages and programming languages. Via reductions to multiset rewriting, several people have found encodings of Petri nets into linear logic [GG90, AFG90, BG90, EW90]. The specification logic ACL of Kobayashi and Yonezawa is an asynchronous calculus in which the send and read primitives are identified with two complementary linear logic connectives [KY93, KY94]. Miller [Mil93] described how features of the π-calculus [MPW92] can be modeled in linear logic and Bruscoli and Guglielmi [BG96] showed how specifications in the Gamma language [BLM96] can be related to linear logic.

Operational semantics Forum has been used to specify the operational semantics of the imperative features in Algol [Mil94] and ML [Chi95] and the concurrency features of Concurrent ML [Mil96]. Forum was used by Chirimar to specify the operational semantics of a pipe-lined, RISC processor [Chi95] and by Chakravarty to specify the operational semantics of a parallel programming language that combines functional and logic programming paradigms [Cha97]. Linear logic has also been used to express and to reason about the operational semantics of security protocols [CDL$^+$99, Mil03]. A similar approach to using linear logic was also applied to specifying real-time finite-state systems [KOS98].

Object-logic proof systems Intuitionistic based systems, such as the LF dependent type system and hereditary Harrop formulas, are popular choices for the specification of natural deduction proof systems [HHP93, FM88]. The linear logic aspects of both Lolli and Linear LF have been used to specify natural deduction systems for a wider collection of object-logics than are possible with these non-linear logic frameworks [HM94, CP96]. By admitting full linear logic and multiple conclusion sequents, Forum provides a natural setting for the specification of object-level sequent calculus proof systems. In classical linear logic, the duality between left and rule introduction rules and between the cut and initial rules is easily explained using the meta-level linear negation. Some examples of specifying object-level sequent proof systems in Forum are given in [Mil94, MP02, Ric98].

Natural language parsing Lambek's precursor to linear logic [Lam58] was motivated in part to deal with natural language syntax. An early use of Lolli was to provide a simple and declarative approach to gap threading and island constraints within English relative clauses [HM94, Hod92] that built on an approach first proposed by Pareschi using intuitionistic logic [Par89, PM90]. Researchers in natural language syntax are generally quick to look closely at most advances in proof theory, and linear logic has not been an exception: for a few additional references, see [DLPS95, Mor95, Moo96].

1.6 Examples of reasoning about a linear logic program

One of the reasons to use logic as the source code for a programming languages is that the actual artifact that is the program should be amenable to direct manipulation and analysis in ways that might be hard or impossible in more conventional programming languages. One method for reasoning directly on logic programming involves the cut rule (via modus ponens) and cut-elimination. We consider here two examples of how the meta-theory of linear logic can be used to prove properties of logic programs.

While much of the motivation for designing logic programming languages based on linear logic has been to add expressiveness to such languages, linear logic can also help shed some light on conventional programs. In this section we consider the linear logic specification for the reverse of lists and formally show it is symmetric.

Let the constants *nil* and $(\cdot :: \cdot)$ denote the two constructors for lists. Consider specifying the binary relation *reverse* that relates two lists if one is the reverse of the other. To specify the computation of reversing a list, consider making two piles on a table. Initialize one pile to the list you wish to reverse and initialize the other pile to be empty. Next, repeatedly move the top element from the first pile to the top of the second pile. When the first pile is empty, the second pile is the reverse of the original list. For example, the following is a trace of such a computation.

$(a :: b :: c :: nil)$	nil
$(b :: c :: nil)$	$(a :: nil)$
$(c :: nil)$	$(b :: a :: nil)$
nil	$(c :: b :: a :: nil)$

In more general terms: first pick a binary relation *rv* to denote the pairing of lists above (this predicate will be an auxiliary predicate to reverse). If we wish to reverse the list L to get K, then start with the atomic formula $(rv\ L\ nil)$ and do a series of backchaining over the clause

$$\forall X \forall P \forall Q (rv\ P\ (X :: Q) \multimap rv\ (X :: P)\ Q)$$

to get to the formula $(rv\ nil\ K)$. Once this is done, K is the result of reversing L. The entire specification of reverse can be written as the following single formula.

$$\begin{array}{l}\forall L \forall K [\ \forall rv\ ((\forall X \forall P \forall Q (rv\ P\ (X :: Q) \multimap rv\ (X :: P)\ Q)) \Rightarrow \\ \qquad\qquad rv\ nil\ K \multimap rv\ L\ nil) \multimap reverse\ L\ K\ ]\end{array}$$

Notice that the clause used for repeatedly moving the top elements of lists is to the left of an intuitionistic implication (so it can be used any number of times) while the formula representing the base case of the recursion, namely $(rv\ nil\ K)$, is to the left of a linear implication (thus it must be used exactly once).

Consider proving that reverse is symmetric: that is, if $(reverse\ L\ K)$ is proved from the above clause, then so is $(reverse\ K\ L)$. The informal proof of this is simple: in the trace table above, flip the rows and the columns. What is left is a correct computation of reversing again, but the start and final lists have exchanged roles. This informal proof is easily made formal by exploiting the meta-theory of linear logic. A more formal proof proceeds as follows. Assume that $(reverse\ L\ K)$ can be proved. There is only one way to prove this (backchaining on the above clause for *reverse*). Thus the formula

$$\forall rv((\forall X \forall P \forall Q(rv\ P\ (X :: Q) \multimap rv\ (X :: P)\ Q)) \Rightarrow rv\ nil\ K \multimap rv\ L\ nil)$$

is provable. Since we are using logic, we can instantiate this quantifier with any binary predicate expression and the result is still provable. So we choose to instantiate it with the λ-expression $\lambda x \lambda y (rv\ y\ x)^\perp$. The resulting formula

$$\begin{array}{r}(\forall X \forall P \forall Q(rv\ (X :: Q)\ P)^\perp \multimap (rv\ Q\ (X :: P)^\perp)) \Rightarrow \\ (rv\ K\ nil)^\perp \multimap (rv\ nil\ L)^\perp\end{array}$$

can be simplified by using the contrapositive rule for negation and linear implication, and hence yields

$$(\forall X \forall P \forall Q(rv\ Q\ (X :: P) \multimap rv\ (X :: Q)\ P) \Rightarrow rv\ nil\ L \multimap rv\ K\ nil)$$

If we now universally generalize on rv we again have proved the body of the reverse clause, but this time with L and K switched. Notice that we have succeeded in proving this fact about reverse without explicit reference to induction.

For another example of using linear logic's meta-theory to reason directly on specifications, consider the problem of adding a global counter to a functional programming language that already has primitives for, say conditionals, application, abstraction, etc [Mil96]. Now add *get* and *inc* expressions: evaluation of *get* causes the counter's value to be returned while evaluation of *inc* causes the counter's value to be incremented. Figure 1.5 contains three specifications, E_1, E_2, and E_3, of such a counter: all three specifications store the counter's value in an atomic formula as the argument of the predicate r. In these three specifications, the predicate r is existentially quantified over the specification in which it is used so that the atomic formula that stores the counter's value is itself local to the counter's specification (such existential quantification of predicates is a familiar technique for implementing abstract datatypes in logic programming [Mil89a]). The first two specifications store the counter's value on the right of the sequent arrow, and reading and incrementing the counter occurs via a synchronization between the *eval*-atom and the r-atom. In the third specification, the counter is stored as a linear assumption on the left of the sequent arrow, and synchronization is not used: instead, the linear assumption is "destructively" read and then rewritten in order to specify *get* and *inc* (counters such as these

$$
\begin{array}{rl}
E_1 = \exists r[& (r\ 0)^\perp \otimes \\
& !\forall K \forall V(\mathit{eval}\ \mathit{get}\ V\ K \parr r\ V \circ\!- \mathit{eval}\ K \parr r\ V)) \otimes \\
& !\forall K \forall V(\mathit{eval}\ \mathit{inc}\ V\ K \parr r\ V \circ\!- K \parr r\ (V+1))]
\end{array}
$$

$$
\begin{array}{rl}
E_2 = \exists r[& (r\ 0)^\perp \otimes \\
& !\forall K \forall V(\mathit{eval}\ \mathit{get}\ (-V)\ K \parr r\ V \circ\!- K \parr r\ V) \otimes \\
& !\forall K \forall V(\mathit{eval}\ \mathit{inc}\ (-V)\ K \parr r\ V \circ\!- K \parr r\ (V-1))]
\end{array}
$$

$$
\begin{array}{rl}
E_3 = \exists r[& (r\ 0) \otimes \\
& !\forall K \forall V(\mathit{eval}\ \mathit{get}\ V\ K \circ\!- r\ V \otimes (r\ V \multimap K)) \otimes \\
& !\forall K \forall V(\mathit{eval}\ \mathit{inc}\ V\ K \circ\!- r\ V \otimes (r\ (V+1) \multimap K))]
\end{array}
$$

Fig. 1.5. Three specifications of a global counter.

are described in [HM94]). Finally, in the first and third specifications, evaluating the *inc* symbol causes 1 to be added to the counter's value. In the second specification, evaluating the *inc* symbol causes 1 to be subtracted from the counter's value: to compensate for this unusual implementation of *inc*, reading a counter in the second specification returns the negative of the counter's value.

The use of $\otimes$, $!$, $\exists$, and negation in Figure 1.5 is for convenience in displaying these abstract datatypes. The curry/uncurry equivalence

$$\exists r(R_1^\perp \otimes !\, R_2 \otimes !\, R_3) \multimap G \equiv \forall r(R_2 \Rightarrow R_3 \Rightarrow G \parr R_1)$$

directly converts a use of such a specification into a formula of Forum (given α-conversion, we may assume that r is not free in G).

Although these three specifications of a global counter are different, they should be equivalent in the sense that evaluation cannot tell them apart. Although there are several ways that the equivalence of such counters can be proved (for example, operational equivalence), the specifications of these counters are, in fact, *logically* equivalent. In particular, the three entailments $E_1 \vdash E_2$, $E_2 \vdash E_3$, and $E_3 \vdash E_1$ are provable in linear logic. The proof of each of these entailments proceeds (in a bottom-up fashion) by choosing an eigenvariable to instantiate the existential quantifier on the left-hand specification and then instantiating the right-hand existential quantifier with some term involving that eigenvariable. Assume that in all three cases, the eigenvariable selected is the predicate symbol s. Then the first entailment is proved by instantiating the right-hand existential with $\lambda x.s\ (-x)$; the second entailment is proved using the substitution $\lambda x.(s\ (-x))^\perp$; and the third entailment is proved using the substitution $\lambda x.(s\ x)^\perp$. The proof of the first two entailments must also use the equations

$$\{-0 = 0, -(x+1) = -x-1, -(x-1) = -x+1\}.$$

The proof of the third entailment requires no such equations.

Clearly, logical equivalence is a strong equivalence: it immediately implies that evaluation cannot tell the difference between any of these different specifications of a counter. For example, assume $E_1 \vdash \mathit{eval}\ M\ V\ \top$. Then by cut and the above entailments, we have $E_2 \vdash \mathit{eval}\ M\ V\ \top$.

1.7 Effective implementations of proof search

There are several challenges facing the implementers of linear logic programming languages. One problem is how to split multiset contexts when proving a tensor or backchaining over linear implications. If the multiset contexts of a sequent have $n \geq 0$ formulas in them, then can be as many as 2^n ways that a context can be partitioned into two multisets. Often, however, very few of these splits will lead to a successful proof. An obvious approach to address the problem of splitting context would be to do the split lazily. One approach to such lazy splitting was presented in [HM94] where proof search was seen to be a kind of input/output process. When proving one part of a tensor, all formulas are given to that attempt. If the proof process is successful, any formulas remaining would then be output from that attempt and handed to the remaining part of the tensor. A rather simple interpreter for such a model of resource consumption and its Prolog implementation is given in [HM94]. Experience with this interpreter showed that the presence of the additive connectives – $\top$ and & – caused significant problems with efficient interpretation. Several researchers have developed significant variations to the model of lazy splitting. See for example, [CHP96, Hod94, LP97]. Similar implementation issues concerning the Lygon logic programming language are described in [WH95]. More recent approaches to accounting for resource consumption in linear logic programming uses constraint solving to treat the different aspects of resources sharing and consumption in different parts of the search for a proof [And01, HP03].

Based on such approaches to lazy splitting, various interpreters of linear logic programming languages have been implemented. To date, however, only one compiling effort has been made. Tamura and Kaneda [TK96] have developed an extension to the Warren abstract machine (a commonly used machine model for logic programming) and a compiler for a subset of Lolli. This compiler was shown in [HT01] to perform surprisingly well for a certain theorem proving application where linear logic provided a particularly elegant specification.

1.8 Research in sequent calculus proof search

Since the majority of linear logic programming is described using sequent calculus proof systems, a great deal of the work in understanding and implementing these languages has focused on properties of the sequent calculus. Besides the work mentioned already concerning refinements to proof search, there is the related work of Galmiche, Boudinet, and Perrier [GB94, GP94], Tammet [Tam94], and Guglielmi [Gug96], and Gabbay and Olivetti [GO00]. And, of course, there is the recent work of Girard on *Locus solum* [Gir01].

Below is briefly described three areas certainly deserving additional consideration and which should significantly expand our understanding and application of proof search and logic programming.

1.8.1 Polarity and proof search.

Andreoli observed the critical role of polarity in proof search: the notion of asynchronous behavior (goal-reduction) and synchronous behavior (backchaining) are de Morgan duals of each other. There have been other uses of polarity in proof systems and proof search. In [Gir93], Girard introduced the LU system in which classical, intuitionistic, and linear logics share a common proof system. Central to their abilities to live together is a notion of polarity: positive, negative, and neutral. As we have shown in this paper, linear logic enhances the expressiveness of logic programming languages presented in classical and intuitionistic logic, but this comparison is made after they have been *translated* into linear logic. It would be interesting to see if there is one logic programming language that contains, for example, a classical, intuitionistic, and linear implication.

1.8.2 Non-commutativity.

Having a non-commutative conjunction or disjunction within a logic programming language should significantly enhance the expressiveness of the language. Lambek's early calculus [Lam58] was non-commutative but it was also weak in that it did not have modals and additive connectives. In recent years, a number of different proposals for non-commutative versions of linear logic have been considered. Abrusci [Abr91] and later Ruet and Abruzzi [AR99, Rue00] have developed one such approach. Remi Baudot [Bau00] and Andreoli and Maieli [AM99] developed focusing strategies for this logic and have designed abstract logic programming languages based on the proposal of Abrusci and Ruet. Alessio Guglielmi has proposed a new approach to representing proofs via the *calculus of structures* and presents a non-commutative connective which is self-dual [GS01]. Paola Bruscoli has shown how that non-commutative connective can be used to code sequencing in the CCS specification language [Bru02]. Christian Retoré has also proposed a non-commutative, self dual connective within the context of proof nets [Ret97, Ret99]. Finally, Pfenning and Polakow have developed a non-commutative version of intuitionistic linear logic with a sequential operator and have demonstrated its uses in several applications [Pol01, PY01, PP99a, PP99b]. Currently, non-commutativity has the appearance of being rather complicated and no single proposal seems to be canonical at this point.

1.8.3 Reasoning about specifications.

One of the reasons for using logic to make specifications in the first place must surely be that the meta-theory of logic should help in establishing properties of logic programs: cut and cut-elimination will have a central role here. While this was illustrated in Section 1.6, very little of this kind of reasoning has been done for logic programs written in logics programming languages more expressive than Horn clauses. The examples in Section 1.6 are also not typical: most reasoning about logic specifications will certainly involve induction. Also, many properties of computational specifications involve being able to reason about all paths that a computation may take: simulation and bisimulation are examples of such properties [Mil89c]. The proof theoretical notion of *fixpoint* [Gir92] and of *definition* [HSH91, MMP03, SH93] has been used to help capture such notions. See, for example, the work on integrating inductions and definitions into intuitionistic logic [MM97, MM00, MM02]. Extending such work to incorporate co-induction and to embrace logics other than intuitionistic logic should certainly be considered.

Of course, there are many other avenues that work in proof search and logic programming design can take. For example, one can investigate rather different logics, for example, the logic of bunched implications [OP99, Pym99], for their suitability as logic programming languages. Model theoretic semantics suitable for reasoning about linear logic specification would certainly be desirable, especially if they can provide simple, natural, and compositional notions of meaning. Also, several application areas of linear logic programming seems convincing enough that work on improving the effectiveness of interpreters and compilers certainly seems appropriate.

Acknowledgments Parts of this overview where written while the author was at Penn State University and was supported in part by NSF grants CCR-9912387, CCR-9803971, INT-9815645, and INT-9815731. Alwen Tiu provided useful comments on a draft of this paper.

Bibliography

[Abr91] V. Michele Abrusci. Phase semantics and sequent calculus for pure non-commutative classical linear propositional logic. *Journal of Symbolic Logic*, 56(4):1403–1451, December 1991.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[AFG90] A. Asperti, G.-L. Ferrari, and R. Gorrieri. Implicative formulae in the 'proof as computations' analogy. In *Principles of Programming Languages (POPL'90)*, pages 59–71. ACM, January 1990.

[AM99] J.-M. Andreoli and R. Maieli. Focusing and proof nets in linear and noncommutative logic. In *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 1581 of *LNAI*. Springer, 1999.

[And90] Jean-Marc Andreoli. *Proposal for a Synthesis of Logic and Object-Oriented Programming Paradigms.* PhD thesis, University of Paris VI, 1990.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[And01] Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1):131–163, 2001.

[AP91a] J.-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proceedings of OOPSLA 91*, pages 212–229, 1991.

[AP91b] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[AR99] V. Michele Abrusci and Paul Ruet. Non-commutative logic I: The multiplicative fragment. *Annals of Pure and Applied Logic*, 101(1):29–64, 1999.

[AvE82] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[Bau00] Rémi Baudot. *Programmation Logique: Non commutativité et Polarisation.* PhD thesis, Université Paris 13, Laboratoire d'informatique de Paris Nord (L.I.P.N.), December 2000.

[BG90] C. Brown and D. Gurr. A categorical linear framework for petri nets. In *Logic in Comptuer Science (LICS'90)*, pages 208–219, Philadelphia, PA, June 1990. IEEE Computer Society Press.

[BG96] Paola Bruscoli and Alessio Guglielmi. A linear logic view of Gamma style computations as proof searches. In Jean-Marc Andreoli, Chris Hankin, and Daniel Le Métayer, editors, *Coordination Programming: Mechanisms, Models and Semantics.* Imperial College Press, 1996.

[BLM96] Jean-Pierre Banâtre and Daniel Le Métayer. Gamma and the chemical reaction model: ten years after. In *Coordination programming: mechanisms, models and semantics*, pages 3–41. World Scientific Publishing, IC Press, 1996.

[Bru02] Paola Bruscoli. A purely logical account of sequentiality in proof search. In Peter J. Stuckey, editor, *Logic Programming, 18th International Conference*, volume 2401 of *LNAI*, pages 302–316. Springer-Verlag, 2002.

[BS94] Gianluigi Bellin and Philip J. Scott. On the pi-calculus and linear logic. *Theoretical Computer Science*, 135:11–65, 1994.

[CDL+99] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW'99*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[Cha97] Manuel M. T. Chakravarty. *On the Massively Parallel Execution of Declarative Programs.* PhD thesis, Technische Universität Berlin, Fachbereich Informatik, February 1997.

[Chi95] Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, University of Pennsylvania, February 1995.

[CHP96] Iliano Cervesato, Joshua Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors, *Proceedings of the 1996 Workshop on Extensions to Logic Programming*, pages 28–30, Leipzig, Germany, March 1996. Springer-Verlag LNAI.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logic framework. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[CP02] Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. *Information & Computation*, 179(1):19–75, November 2002.

[DLPS95] M. Dalrymple, J. Lamping, F. Pereira, and V. Saraswat. Linear logic for meaning assembly. In *Proceedings of the Workshop on Computational Logic for Natural Language Processing*, 1995.

[DM95] Giorgio Delzanno and Maurizio Martelli. Objects in Forum. In *Proceedings of the International Logic Programming Symposium*, 1995.

[EW90] U. Engberg and G. Winskel. Petri nets and models of linear logic. In A. Arnold, editor, *CAAP'90*, LNCS 431, pages 147–161. Springer Verlag, 1990.

[Fel91] Amy Felty. Transforming specifications in a dependent-type lambda calculus to specifications in an intuitionistic logic. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks.* Cambridge University

Press, 1991.
[Fel93] Amy Felty. Encoding the calculus of constructions in a higher-order logic. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 233–244. IEEE, June 1993.
[FM88] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61–80, Argonne, IL, May 1988. Springer-Verlag.
[FRS98] François Fages, Paul Ruet, and Sylvain Soliman. Phase semantics and verification of concurrent constraint programs. In Vaughan Pratt, editor, *Symposium on Logic in Computer Science*. IEEE, July 1998.
[GB94] Didier Galmiche and E. Boudinet. Proof search for programming in intuitionistic linear logic. In D. Galmiche and L. Wallen, editors, *CADE-12 Workshop on Proof Search in Type-Theoretic Languages*, pages 24–30, Nancy, France, June 1994.
[GG90] Vijay Gehlot and Carl Gunter. Normal process representatives. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 200–207, Philadelphia, Pennsylvania, June 1990. IEEE Computer Society Press.
[GH90] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 333–343, Philadelphia, Pennsylvania, June 1990. IEEE Computer Society Press.
[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
[Gir92] Jean-Yves Girard. A fixpoint theorem in linear logic. Email to the linear@cs.stanford.edu mailing list, February 1992.
[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.
[Gir01] Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11(3):301–506, June 2001.
[GO00] Dov M. Gabbay and Nicola Olivetti. *Goal-Directed Proof Theory*, volume 21 of *Applied Logic Series*. Kluwer Academic Publishers, August 2000.
[GP94] Didier Galmiche and Guy Perrier. Foundations of proof search strategies design in linear logic. In *Symposium on Logical Foundations of Computer Science*, pages 101–113, St. Petersburg, Russia, 1994. Springer-Verlag LNCS 813.
[GS01] Alessio Guglielmi and Lutz Straßburger. Non-commutativity and MELL in the calculus of structures. In L. Fribourg, editor, *CSL 2001*, volume 2142 of *LNCS*, pages 54–68, 2001.
[Gug96] Alessio Guglielmi. *Abstract Logic Programming in Linear Logic—Independence and Causality in a First Order Calculus*. PhD thesis, Università di Pisa, 1996.
[Har60] R. Harrop. Concerning formulas of the types $A \to B \vee C$, $A \to (Ex)B(x)$ in intuitionistic formal systems. *Journal of Symbolic Logic*, pages 27–32, 1960.
[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
[HM88] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In *Fifth International Logic*

Programming Conference, pages 942–959, Seattle, Washington, August 1988. MIT Press.

[HM90] Joshua Hodas and Dale Miller. Representing objects in a logic programming language with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 511–526. MIT Press, June 1990.

[HM91] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 32–42, Amsterdam, July 1991.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[Hod92] Joshua Hodas. Specifying filler-gap dependency parsers in a linear-logic programming language. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 622–636, 1992.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994.

[HP03] James Harland and David Pym. Resource-distribution via boolean constraints. *ACM Transactional on Computational Logic*, 4(1):56–90, 2003.

[HPW96] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology*, pages 391 – 405, July 1996.

[HSH91] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. II. Programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, October 1991.

[HT01] Joshua S. Hodas and Naoyuki Tamura. lolliCop — A linear logic implementation of a lean connection-method theorem prover for first-order classical logic. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of IJCAR: International Joint Conference on Automated Reasoning*, number 2083 in LNCS, pages 670–684, 2001.

[Kan94] Max Kanovich. The complexity of Horn fragments of linear logic. *Annals of Pure and Applied Logic*, 69:195–241, 1994.

[Kle52] Stephen Cole Kleene. Permutabilities of inferences in Gentzen's calculi LK and LJ. *Memoirs of the American Mathematical Society*, 10:1–26, 1952.

[KOS98] M.I. Kanovich, M. Okada, and A. Scedrov. Specifying real-time finite-state systems in linear logic. In *Second International Workshop on Constraint Programming for Time-Critical Applications and Multi-Agent Systems (COTIC)*, Nice, France, September 1998. Also appears in the *Electronic Notes in Theoretical Computer Science*, Volume 16 Issue 1 (1998) 15 pp.

[KY93] Naoki Kobayashi and Akinori Yonezawa. ACL - a concurrent linear logic programming paradigm. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 279–294. MIT Press, October 1993.

[KY94] Naoki Kobayashi and Akinori Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 3:279–294, 1994.

[Laf89] Yves Lafont. Functional programming and linear logic. Lecture notes for the Summer School on Functional Programming and Constructive Logic, Glasgow, United Kingdom, 1989.

[Laf90] Yves Lafont. Interaction nets. In *Seventeenth Annual Symposium on Principles of Programming Languages*, pages 95–108, San Francisco, California, 1990. ACM Press.

[Lam58] J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–169, 1958.

[LP97] Pablo López and Ernesto Pimentel. A lazy splitting system for Forum. In *AGP'97: Joint Conference on Declarative Programming*, 1997.

[LS93] P. Lincoln and V. Saraswat. Higher-order, linear, concurrent constraint programming. Available as ftp://parcftp.xerox.com/pub/ccp/lcc/hlcc.dvi., January 1993.

[Mil86] Dale Miller. A theory of modules for logic programming. In Robert M. Keller, editor, *Third Annual IEEE Symposium on Logic Programming*, pages 106–114, Salt Lake City, Utah, September 1986.

[Mil89a] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[Mil89b] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, January 1989.

[Mil89c] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[Mil93] Dale 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 Extensions to Logic Programming*, number 660 in LNCS, pages 242–265. Springer-Verlag, 1993.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, July 1994. IEEE Computer Society Press.

[Mil95] Dale Miller. A survey of linear logic programming. *Computational Logic: The Newsletter of the European Network in Computational Logic*, 2(2):63 – 67, December 1995.

[Mil96] Dale Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1):201–232, September 1996.

[Mil03] Dale Miller. Encryption as an abstract data-type: An extended abstract. In Iliano Cervesato, editor, *Proceedings of FCS'03: Foundations of Computer Security*, pages 3–14, 2003.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 434–445, Warsaw, Poland, July 1997. IEEE Computer Society Press.

[MM00] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[MM02] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.

[MMP03] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. *TCS*, 294(3):411–437, 2003.

[MN86] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[MNPS91] 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.

[Moo96] Michael Moortgat. Categorial type logics. In Johan van Benthem and Alice ter Meulen, editors, *Handbook of Logic and Language*, pages 93–177. Elsevier, Amsterdam, 1996.

[Mor95] Glyn Morrill. Higher-order linear logic programming of categorial deduction. In *7th Conference of the Association for Computational Linguistics*, pages 133–140, Dublin, Ireland, 1995.

[MOTW99] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science*, 228(1–2):175–210, 1999.

[MP02] Dale Miller and Elaine Pimentel. Using linear logic to reason about sequent systems. In Uwe Egly and Christian G. Fermüller, editors, *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2002.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992.

[Nad87] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming*. PhD thesis, University of Pennsylvania, May 1987.

[NJK95] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, November 1995.

[NM88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.

[NM90] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 499 – 590. Clarendon Press, Oxford, 1998.

[O'H91] P. W. O'Hearn. Linear logic and interference control: Preliminary report. In S. Abramsky, P.-L. Curien, A. M. Pitts, D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Proceedings of the Conference on Category Theory and Computer Science*, pages 74–93, Paris, France, 1991. Springer-Verlag LNCS 530.

[OP99] P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[Par89] Remo Pareschi. *Type-driven Natural Language Analysis.* PhD thesis, University of Edinburgh, 1989.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[PH94] David J. Pym and James A. Harland. The uniform proof-theoretic foundation of linear logic programming. *Journal of Logic and Computation*, 4(2):175 – 207, April 1994.

[PM90] Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990.

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications.* PhD thesis, Department of Computer Science, Caregnie Mellon, August 2001.

[PP99a] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 295–309, L'Aquila, Italy, 1999. Springer-Verlag LNCS 1581.

[PP99b] Jeff Polakow and Frank Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In Andre Scedrov and Achim Jung, editors, *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, 1999.

[PY01] Jeff Polakow and Kwangkeun Yi. Proving syntactic properties of exceptions in an ordered logical framework. In *Fifth International Symposium on Functional and Logic Programming (FLOPS 2001)*, Tokyo, Japan, March 2001.

[Pym99] David J. Pym. On bunched predicate logic. In G. Longo, editor, *Proceedings of LICS99: 14th Annual Symposium on Logic in Computer Science*, pages 183–192, Trento, Italy, 1999. IEEE Computer Society Press.

[Ret97] Christian Retoré. Pomset logic: a non-commutative extension of classical linear logic. In *Proceedings of TLCA*, volume 1210, pages 300–318, 1997.

[Ret99] Christian Retoré. Pomset logic as a calculus of directed cographs. In V. M. Abrusci and C. Casadio, editors, *Dynamic Perspectives in Logic and Linguistics: Proof Theoretical Dimensions of Communication Processes*, pages 221–247, 1999.

[RF97] P. Ruet and F. Fages. Concurrent constraint programming and non-commutative logic. In *Proceedings of the 11^th Conference on Computer Science Logic*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[Ric98] Giorgia Ricci. *On the expressive powers of a Logic Programming presentation of Linear Logic (FORUM).* PhD thesis, Department of Mathematics, Siena University, December 1998.

[Rue00] Paul Ruet. Non-commutative logic II: sequent calculus and phase semantics. *Mathematical Structures in Computer Science*, 10(2):277–312, 2000.

[Sar93] V. Saraswat. A brief introduction to linear concurrent constraint

programming. Available as
ftp://parcftp.xerox.com/pub/ccp/lcc/lcc-intro.dvi., 1993.

[SH93] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, June 1993.

[Tam94] T. Tammet. Proof strategies in linear logic. *Journal of Automated Reasoning*, 12:273–304, 1994.

[TK96] Naoyuki Tamura and Yukio Kaneda. Extension of wam for a linear logic programming language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Second Fuji International Workshop on Functional and Logic Programming*, pages 33–50. World Scientific, November 1996.

[Wad90] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, pages 561–581. North Holland, 1990.

[WH95] M. Winikoff and J. Harland. Implementing the Linear Logic Programming Language Lygon. In *Proceedings of the International Logic Programming Symposium*, pages 66–80, December 1995.