AN OVERVIEW OF LINEAR LOGIC PROGRAMMING

DALE MILLER

Abstract. Logic programming can be given a foundation in sequent calculus, viewing computation as the process of building a cut-free sequent proof from the bottom-up. Earliest accounts of logic programming were based in classical logic and then later in intuitionistic logic. The use of linear logic to design new logic programming languages was inevitable given that it allows for more dynamics in the way sequents change during the search for a proof and since it can account for logic programming in these other logics. We overview how linear logic has been used to design new logic programming languages and describe some applications and implementation issues for such languages.

§1. Introduction. It is now common place to recognize the important role of logic in the foundations of computer science in general and programming languages more specifically. For this reason, when a major new advance is made in our understanding of logic, we can expect to see that advance ripple into other areas of computer science. Such rippling has been observed during the years since the first introduction of linear logic [30]. Linear logic not only embraces computational themes directly in its design and foundation but also extends and enriches classical and intuitionistic logic, thus providing possibly new insights into the many computational systems built on those two logics.

There are two broad ways in which proof theory and computation can be related [64]. One relies on *proof reduction* and can be seen as a foundation for *functional 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 [1] has extended this interpretation of computation to *multiple-conclusion* sequents, $\Delta \longrightarrow \Gamma$, where Δ and Γ are both multi-sets of propositions. In that setting, cut-elimination can be seen as a general form of computation and the programs specified are concurrent in nature. In particular, Abramsky presents a method for "realizing" the computational content of multiple-conclusion proofs in linear logic that yields programs in various concurrency formalisms. See also [14, 53, 54] for related uses of concurrency in proof normalization in linear logic. The more expressive types made possible by linear logic have been used to help provide static analysis of such things as run-time garbage, aliases, reference counters, and single-threadedness [36, 57, 75, 93].

The other way that proof theory provides a foundation to computation relies on *proof search*: this view of computation can be seen as a foundation for *logic programming*. This topic is the focus of this paper.

§2. Computation as 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 [68], sequents can be seen as containing two kinds of formulas: *program clauses* describing the meaning of non-logical constants and *goals* describing the desired conclusion of the program for which a search is being made. A sequent $\Delta \longrightarrow \Gamma$ represents the state of an idealized logic programming interpreter in which the current logic program is Δ and the goal is Γ, both of which might be sets or multisets of formulas. These two classes are duals of each other in the sense that a negative subformula of a goal is a program clause and the negative subformula of a program clause is a goal formula.

2.1. Goal-directed search and uniform proofs. Since proof search can contain a great deal of non-deterministic than does not seem computationally important, a normal form for proof search is generally imposed. One approach to presenting a normal form is based on a notion of *goal-directed search* using the technical notion of *uniform proofs*. In the single-conclusion setting, where Γ contains one formula, uniform proofs have a particularly simple definition [68]: a *uniform proof* is a cut-free proof where every sequent with a non-atomic right-hand side is the conclusion of a right-introduction rule. 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. Left-introduction rules can only be used when the goal formula is atomic and, in this way, goal-reduction is done without any reference to the logic program. 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 [50]. 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 [64, 66].

A given logic and proof system is called an *abstract logic programming language* if a sequent has a proof if and only if it has a uniform proof. First-order and

higher-order variants of Horn clauses paired with classical logic [72] and hereditary Harrop formulas paired with intuitionistic logic [68] are two examples of abstract logic programming languages. The cut rule and cut-elimination can play various meta-theoretic roles, such as guarantor of completeness and of canonical models [45, 63] and as a tool for reasoning about encoded computations.

The notion of uniform proofs, however, does not fully illuminate what goes on in proof search in abstract logic programming languages. While the definition of uniform proofs capture the fact that goals can be reduced without referring to context, that definition say nothing about proving atomic goal formulas. In that case, the context (the logic program) must play a role. In particular, in the various examples of abstract logic programming languages that have been studied (*e.g.*, [72, 68, 44]) atomic goals were all proved using a suitable generalization of *backchaining*. Backchaining turned out to be a phase in proof construction that used left-introduction rules in a focused decomposition of a program clause that yielded not only a matching atomic formula occurrence to one in the goal but also possibly new goals formulas for which additional proofs are needed.

2.2. Focusing proof search. In studying normal forms of proof search for linear logic, Andreoli was able to complete and generalize the picture above in two directions. First, he referred to goal-decomposition as described above as *asynchronous* proof construction and backchaining as *synchronous* proof construction and observed that the logical connectives responsible for these two phases of search were duals of each other. For example, & (additive conjunction) on the right is asynchronous while & on the left is synchronous. If a single-sided sequent is used (no formulas on the left of the sequent arrow), then this is equivalent to saying that & is asynchronous and $\oplus$ is synchronous. Secondly, Andreoli showed that a suitable interleaving of asynchronous and synchronous, similar to that for goal-decomposition and backchaining, was complete for all of linear logic [6, 7]. In other words, all of linear logic can be viewed as an abstract logic programming language.

Because linear logic can be seen as the logic behind classical and intuitionistic logic, it is possible to translate Horn clauses and hereditary Harrop formulas into linear logic and to apply Andreoli's focused proofs to their translations: indeed, the first-order logic results of those systems can be seen as consequences of Andreoli's completeness theorem for linear logic.

§3. Logic programming in classical and intuitionistic logics. 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).

In the beginning of the logic programming literature, there was one example of logic programming, namely, first-order Horn clauses, which was the logic basis of the popular programming language Prolog. No general framework for the connection between logic and logic programming was available. The operational semantics of logic programs was presented as resolution [10], an inference rule optimized for classical reasoning: variations of resolution to other logical settings were complex and generally artificial. Miller and Nadathur [67, 72, 73] were probably the first to use the sequent calculus to explain design and correctness

issues for a logic programming language (in particular, a higher-order version of Horn clauses). The sequent calculus seemed to have more explanatory powers and allowed the separation of language design and implementation details that were not allowed with resolution, where inference rules and unification were merged.

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 [72] that a proof system similar to the one in Figure 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. 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.

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

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

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

FIGURE 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.

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

FIGURE 2. The rule for universal quantification has the proviso that c is not declared in Σ.

Inference rules in Figure 1, and those that we shall show in subsequent proof systems, can be divided into four categories. The right-introduction rules (goal-reduction) and left-introduction rules (backchaining) form two classes. The remaining two classes do not contain instances of logical connectives. The third class of rules is that of the "initial" rules: these rules have an empty premise and their conclusion has a repeated occurrence of a schema variable. The final class is that of the "decide" rules: in these rules, formulas are moved from one part of a context to another part. In Figure 1, there is one such decide rule in which a formula from the left-hand side of the sequent arrow is moved to on top

of the sequent arrow. The cut rule, which does not interest us in this discussion, would comprise a fifth class of inference rule.

In this proof system, left-introductions are now focused 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 synchronization between a goal formula and a program clause via the repeated schema variable in the initial rule. In Figure 1, there is one decide rule and one initial rule: in a subsequent inference system, there are more of each class. 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 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 [68] that a proof system similar to the one formed by adding to the inference rules in Figure 1 the rules in Figure 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.

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 [62] and abstract datatypes [61].

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 $p \vee q \longrightarrow q \vee p$ and $\exists x.B \longrightarrow \exists x.B$ have intuitionistic proofs but neither sequent has a uniform proof. Positive occurrences of $\vee$ and $\exists$ in goals formulas (and in negative occurrences of program clauses) can be allowed and without lossing the completeness of uniform proofs. (When such occurrences of $\vee$ and $\exists$ are allowed, these formulas can be seen as a restriction on Harrop formulas [41].) As is well known, higher-order quantification allows one to pick a different set of primitive logical connectives for intuitionistic logic. For example,

the intuitionistic disjunction $B \vee C$ can be defined as

$$\forall p((B \supset p) \supset (C \supset p) \supset p).$$

In that case, the sequent corresponding to $p \vee q \longrightarrow q \vee p$ would, in fact, have a uniform proof. Felty has shown that higher-order intuitionistic logic based on *true*, $\wedge$, $\supset$, and $\forall$ for all higher-order types is an abstract logic programming language [26].

§4. Logic programming in linear logic. While higher-order quantification does have important roles to play in logic programming, it does not seem to offer a direct way to get at the computational foundation of proof search: following the work of Andreoli on *focused proofs*, linear logic certainly has great promise for expanding such a foundation.

4.1. The Forum presentation of linear logic. In the classical and intuitionistic logics considered in the previous section, logic programming languages were based on the connectives *true*, $\wedge$, $\supset$, and $\forall$, and the proof systems used two sequent, $\Sigma : \Delta \longrightarrow G$ and $\Sigma : \Delta \xrightarrow{D} A$, where Δ is a set of formulas. We shall now consider a presentation of linear logic using the connectives $\top$, $\&$, $\bot$, $\parr$, $\Rightarrow$, $\multimap$, ?, and $\forall$. This collection of connectives yields a presentation of all of linear logic since it contains a complete set of connectives. The missing connectives are directly definable using the following logical equivalences.

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

This collection of connectives is not minimal. For example, ? and $\parr$ can be defined in terms of the remaining connectives.

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

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 or simply *Forum*.

The two sequent judgments, for goal-reduction (right-introductions) and backchaining (left-introductions), used in this presentation of linear logic are written as $\Sigma : \Psi; \Delta \longrightarrow \Gamma; \Upsilon$ and $\Sigma : \Psi; \Delta \xrightarrow{D} \mathcal{A}; \Upsilon$, 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. This is, unfortunately, the opposite convention used by Girard's LU proof system [32] where sequents have similarly divided contexts on the left and right of the sequent arrow.

The focusing result of Andreoli [7] can be formulated [66] as the completeness of the proof system for linear logic using the proof system in Figure 3. This proof system appears rather complicated at first glance, so it is worth noting that all

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

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

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

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

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

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

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

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

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

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

FIGURE 3. A proof system for 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 &, $i \in \{1, 2\}$.

its inference rules fit into the four classes mentioned before: there are 8 right rules, 7 left 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 bounded right context to the bounded right context. This third decide rule is a combination of contraction and dereliction rule for ? and is used to "decide" on a new goal formula on which to do reductions.

Forum is a recently proposed linear logic programming languages. Below we overview various other subsets of linear logic that have been proposed as specification languages and as abstract logic programming languages.

4.2. Lolli. The connectives $\bot$, $\parr$, and ? force the genuinely classical feel of linear logic. Without these three connectives, the multiple-conclusion sequent calculus given for Forum in Figure 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

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

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

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

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

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

FIGURE 4. The proof system Lolli. The rule for universal quantification has the proviso that c is not declared in Σ. In the $\forall$-left rule, t is a Σ-term of type τ.

language. Presenting a sequent calculus for Lolli is a simple matter. First, remove any inference rule in Figure 3 involving $\bot$, $\parr$, 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 4. The completeness of this proof system for Lolli was given by Hodas and Miller in [45]. Given the completeness of the Forum proof system, the correctness of the Lolli proof system is a simple consequence.

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 [45, 66, 84] allow for such occurrences.

4.4. Other subsets of linear logic. 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. 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 and 2 are essentially the same as those for the corresponding proofs in Figure 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 4 of the linear logic formulas yield proofs identical to those in Figures 1 and 2. (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, the program clause

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

can be translated as either

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

using the first of these translations or as

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

using the second (assuming it is to appear on the left of the sequent arrow). This latter formula is, of course, the uncurried form of the formula

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

Historically speaking, the first proposal for a linear logic programming language was LO (Linear Objects) by Andreoli and Pareschi [5, 9]. LO is an extension to the Horn clause paradigm in which atomic formulas are generalized to be multisets of atomic formulas connected by $⅋$ (the multiplicative disjunctions). In LO, backchaining becomes multiset rewriting, which was used by the authors to specify object-oriented programming and the coordination of processes. LO is a subset of the LinLog [6, 7], 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 $\bot$, $⅋$, ?, $\top$, &, 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.

4.5. Other uses of linear logic in proof search. Harland and Pym have analyzed proof search in linear logic [84] and proposed a subset of linear logic they call Lygon [39] as a logic programming language. Since they chose a different definition for uniform proofs and goal-directed search than what is presented here, Lygon is not an abstract logic programming language in the sense given here. Since its motivations lie within proof search and since it is a subset of linear logic and, hence, of Forum, examples and applications of Lygon can generally be translated into either Lolli or Forum rather directly.

Let G and H be formulas composed of $\bot$, $\parr$, and $\forall$. Closed formulas of the form $\forall \bar{x}[G \multimap H]$ where H is not $\bot$ have been called *process clauses* in [64] and were used there to encode a calculus similar to the π-calculus: the universal quantifier in goals were 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 has used them to model computation via multiset rewriting [48].

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 [51, 52] 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 and used linear logic connectives to extend previous languages in this paradigm [56, 89].

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 [40] can be mapped naturally [25] into a higher-order extension of hereditary Harrop formulas [68]. Inspired by a subset of linear logic similar to Lolli, Cervesato and Pfenning developed an extension to LF called Linear LF [19].

§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

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

that specifies that a multiset should 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 4 and the left- and

right-hand sides of sequents in Figure 3 have multisets of formulas, it is an easy matter to write clauses in linear logic which can rewrite these multisets when they are used in backchaining.

To rewrite the right-hand multiset following 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 suppressed in sequents).

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

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 by backchaining on the clause 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 would mean something like: 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 an early goal of the LO system [9] and capturing state encapsulation was a motivation [43] for the design of Lolli. State encapsulation was also addressed using Forum in [23, 65].

Concurrency: Linear logic has often been seen as providing a possible declarative foundation for concurrent specification and programming languages. Via reductions to multiset rewriting, several people have found encodings of Petri nets into linear logic [29, 11, 15, 24]. The specification logic ACL of Kobayashi and Yonezawa is an asynchronous calculus in which the send and read primitives were essentially identified to two complementary linear logic connectives [51, 52]. Miller [64] describe how features of the π-calculus [69] can be modeled in linear logic and Bruscoli and Guglielmi [16] showed

how specifications in the Gamma language [12] can be related to linear logic.

Operational semantics: Forum has been successfully used to specify the operational semantics of imperative features such as those in Algol [65] and ML [21] and the concurrency features of Concurrent ML [66]. Forum was also used by Chirimar to specify the operational semantics of a pipelined, RISC processor [21] and by Manuel Chakravarty [20] to specify the operational semantics of a parallel programming language that combines functional and logic programming paradigms. Cervesato *et.al.* [17] use linear logic to express the operational semantics of security protocols and used that framework to reason about those protocols. A similar approach to using linear logic was also applied to specifying real-time finite-state systems [49].

Object-logic proof systems: Both Lolli and Linear LF have been used to refine the usual, intuitionistic specifications of object-level natural deduction systems and to allow for the specification of natural deduction systems for a wider collection of object-logics than were possible with either hereditary Harrop formulas or LF [45, 19]. Forum shows promise for providing a framework for the specification of a wide range of sequent calculus proof systems much as single-conclusion systems have been used for the specification of natural deduction systems. Some examples of specifying sequent proof systems in Forum are given in [65, 87].

Natural language parsing: Lambek's precursor to linear logic [55] was motivated in part to deal with natural language syntax. An early motivation for Lolli [45, 42] came from the fact that it improved on the declarative approach to gap threading within English relative clauses first proposed by Pareschi [77, 78]. 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 just a few references, see [22, 71, 70].

§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 ammenable to direct manipulation and analysis in ways that might be hard or impossible in more conventional programming languages. 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 each other. First, consider how to compute the reverse of a list. Make a place for 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: if we wish to reverse the list L to get K, first pick a binary relation rv to denote the pairing of lists above (this predicate will not denote the reverse); then start with the atom $(rv\ L\ nil)$ and do a series of backchaining over the clause

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

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.

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

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 $(rv\ nil\ K)$, the base case of the recursion, is to the left of a linear implication (must be used once).

Now 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

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

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 (taken from [66]) of using linear logic's meta-theory consider the three specification of how evaluation of a counter might be added to the specification of evaluation for a functional or imperative language [66]. The counter is global and the formula $(r\ n)$ declares that the counter contains value n. Evaluation of *get* causes the counter's value to be returned while evaluation

$$
\begin{aligned}
E_1 = \exists r[\ \ & (r\ 0)^{\perp} \otimes \\
& !\forall K \forall V (\mathit{eval}\ \mathit{get}\ V\ K\ \parr r\ V \multimap \mathit{eval}\ K\ \parr r\ V)) \otimes \\
& !\forall K \forall V (\mathit{eval}\ \mathit{inc}\ V\ K\ \parr r\ V \multimap K\ \parr r\ (V+1))] \\
E_2 = \exists r[\ \ & (r\ 0)^{\perp} \otimes \\
& !\forall K \forall V (\mathit{eval}\ \mathit{get}\ (-V)\ K\ \parr r\ V \multimap K\ \parr r\ V) \otimes \\
& !\forall K \forall V (\mathit{eval}\ \mathit{inc}\ (-V)\ K\ \parr r\ V \multimap K\ \parr r\ (V-1))] \\
E_3 = \exists r[\ \ & (r\ 0) \otimes \\
& !\forall K \forall V (\mathit{eval}\ \mathit{get}\ V\ K \multimap r\ V \otimes (r\ V \multimap K)) \otimes \\
& !\forall K \forall V (\mathit{eval}\ \mathit{inc}\ V\ K \multimap r\ V \otimes (r\ (V+1) \multimap K))]
\end{aligned}
$$

FIGURE 5. Three specifications of a global counter.

of *inc* causes the counter's value to be incremented. Figure 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 data types in logic programming [61]). 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 an *eval*-atom and an 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 the *get* and *inc* functions (counters such as these are described in [45]). 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 5 is for convenience in displaying these abstract data types. 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 eigen-variable to instantiate the existential quantifier on the left-hand specification and then instantiating the right-hand existential quantifier with some term involving that

eigen-variable. Assume that in all three cases, the eigen-variable 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 proposition, we have $E_2 \vdash \mathit{eval}\ M\ V\ \top$.

§7. Effective implementations of proof search. There are many several challenges facing the implementers of linear logic programming languages. One easily observed problem is that of splitting multiset contexts when proving a tensor or backchaining over linear implications. If the multiset contexts of a sequent have $n \geq 0$ formula in them, then there are 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 [45] where proof search was seen to be a kind of input/output process. When proving one part of a tensor, all formulas were 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 were given in [45]. 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, [18, 47, 76]. Similar implementation issues concerning the Lygon logic programming language are described in [94]. 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 [8, 38].

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 [92] 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 [46] to perform surprisingly well for a certain theorem proving application where linear logic provided a particularly elegant specification.

§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 work in understanding and implementing these languages has focused on

properties of proofs, rather than on model theoretic considerations. Besides the work mentioned already concerning refinements to proof search, there is the related work Galmiche, Boudinet, and Perrier [27, 28], Tammet [91], and Guglielmi [34]. And, of course, there is also the recent monograph of Girard's on *Locus solum* [33].

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.

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 [32], 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.

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 [55] 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. Abruzzi [2] and later Ruet and Abruzzi [3, 88] have developed one such approach. Remi Baudot [13] and Andreoli and Maieli [4] developed focusing strategies for this logic and hence design abstract logic programming languages based on the proposal of Abruzzi 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 [35]. Christian Retoré has also proposed a non-commutative, self dual connective within the context of proof nets [85, 86]. 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 [79, 82, 80, 81]. Currently, non-commutativity has the appearance of being rather complicated and no single proposal seems to be canonical at this point.

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 6, very little of this kind of reasoning has been done for logic programs written in logics stronger than Horn clauses. The examples in Section 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. The proof theoretical notion of *fixpoint* [31] and of *definition*

[37, 90] has been used to help capture such notions. See, for example, the work on integrating inductions and definitions into intuitionistic logic [58, 60, 59]. 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 [74, 83], for their suitability as logic programming languages. 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 Miller has received funding from NSF grants CCR-9912387, CCR-9803971, INT-9815645, and INT-9815731. Alwen Tiu provided useful comments on a draft of this paper.

REFERENCES

[1] Samson Abramsky, *Computational interpretations of linear logic*, ***Theoretical Computer Science***, vol. 111 (1993), pp. 3–57.

[2] V. Michele Abrusci, *Phase semantics and sequent calculus for pure non-commutative classical linear propositional logic*, ***The Journal of Symbolic Logic***, vol. 56 (1991), no. 4, pp. 1403–1451.

[3] V. Michele Abrusci and Paul Ruet, *Non-commutative logic i: The multiplicative fragment*, ***Annals of Pure and Applied Logic***, vol. 101 (1999), no. 1, pp. 29–64.

[4] J.-M. Andreoli and R. Maieli, *Focusing and proof nets in linear and noncommutative logic*, ***International conference on logic for programming and automated reasoning (lpar)***, LNAI, vol. 1581, Springer, 1999.

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

[6] Jean-Marc Andreoli, *Proposal for a synthesis of logic and object-oriented programming paradigms*, ***Ph.D. thesis***, University of Paris VI, 1990.

[7] ———, *Logic programming with focusing proofs in linear logic*, ***Journal of Logic and Computation***, vol. 2 (1992), no. 3, pp. 297–347.

[8] J.M. Andreoli, *Focussing and proof construction*, To appear, *Annals of Pure and Applied Logic*, 2001.

[9] J.M. Andreoli and R. Pareschi, *Linear objects: Logical processes with built-in inheritance*, ***New Generation Computing***, vol. 9 (1991), no. 3-4, pp. 445–473.

[10] K. R. Apt and M. H. van Emden, *Contributions to the theory of logic programming*, ***Journal of the ACM***, vol. 29 (1982), no. 3, pp. 841–862.

[11] A. Asperti, G.-L. Ferrari, and R. Gorrieri, *Implicative formulae in the 'proof as computations' analogy*, ***Principles of programming languages (POPL'90)***, ACM, January 1990, pp. 59–71.

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

[13] Rémi Baudot, *Programmation logique: Non commutativité et polarisation*, ***Ph.D. thesis***, Université Paris 13, Laboratoire d'informatique de Paris Nord (L.I.P.N.), December 2000.

[14] Gianluigi Bellin and Philip J. Scott, *On the pi-calculus and linear logic*, ***Theoretical Computer Science***, vol. 135 (1994), pp. 11–65.

[15] C. Brown and D. Gurr, *A categorical linear framework for petri nets*, ***Logic in comptuer science (lics'90)*** (Philadelphia, PA), IEEE Computer Society Press, June 1990, pp. 208–219.

[16] Paola Bruscoli and Alessio Guglielmi, *A linear logic view of Gamma style computations as proof searches*, ***Coordination programming: Mechanisms, models and semantics*** (Jean-Marc Andreoli, Chris Hankin, and Daniel Le Métayer, editors), Imperial College Press, 1996.

[17] Cervesato, Durgin, Lincoln, Mitchell, and Scedrov, *A meta-notation for protocol analysis*, ***PCSFW: Proceedings of the 12th computer security foundations workshop***, IEEE Computer Society Press, 1999.

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

[19] Iliano Cervesato and Frank Pfenning, *A linear logic framework*, ***Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*** (New Brunswick, New Jersey), IEEE Computer Society Press, July 1996, An extended version of this paper will appear in *Information and Computation.*, pp. 264–275.

[20] Manuel M. T. Chakravarty, *On the massively parallel execution of declarative programs*, ***Ph.D. thesis***, Technische Universität Berlin, Fachbereich Informatik, February 1997.

[21] Jawahar Chirimar, *Proof theoretic approach to specification languages*, ***Ph.D. thesis***, University of Pennsylvania, February 1995.

[22] M. Dalrymple, J. Lamping, F. Pereira, and V. Saraswat, *Linear logic for meaning assembly*, ***Proceedings of the workshop on computational logic for natural language processing***, 1995.

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

[24] U. Engberg and G. Winskel, *Petri nets and models of linear logic*, ***Caap'90*** (A. Arnold, editor), LNCS 431, Springer Verlag, 1990, pp. 147–161.

[25] Amy Felty, *Transforming specifications in a dependent-type lambda calculus to specifications in an intuitionistic logic*, ***Logical frameworks*** (Gérard Huet and Gordon D. Plotkin, editors), Cambridge University Press, 1991.

[26] ———, *Encoding the calculus of constructions in a higher-order logic*, ***Eighth annual symposium on logic in computer science*** (M. Vardi, editor), IEEE, June 1993, pp. 233–244.

[27] Didier Galmiche and E. Boudinet, *Proof search for programming in intuitionistic linear logic*, ***Cade-12 workshop on proof search in type-theoretic languages*** (Nancy, France) (D. Galmiche and L. Wallen, editors), June 1994, pp. 24–30.

[28] Didier Galmiche and Guy Perrier, *Foundations of proof search strategies design in linear logic*, ***Symposium on logical foundations of computer science*** (St. Petersburg, Russia), Springer-Verlag LNCS 813, 1994, Also available as Technical Report CRIN 94-R-112 from the Centre di Recherche en Informatique de Nancy, pp. 101–113.

[29] Vijay Gehlot and Carl Gunter, *Normal process representatives*, ***Proceedings of the Fifth Annual Symposium on Logic in Computer Science*** (Philadelphia, Pennsylvania), IEEE Computer Society Press, June 1990, pp. 200–207.

[30] Jean-Yves Girard, *Linear logic*, ***Theoretical Computer Science***, vol. 50 (1987), pp. 1–102.

[31] ———, *A fixpoint theorem in linear logic*, Email to the linear@cs.stanford.edu mailing list, February 1992.

[32] ———, *On the unity of logic*, ***Annals of Pure and Applied Logic***, vol. 59 (1993), pp. 201–217.

[33] ———, *Locus solum*, ***Mathematical Structures in Computer Science***, vol. 11 (2000), no. 3.

[34] Alessio Guglielmi, *Abstract logic programming in linear logic—independence and causality in a first order calculus*, ***Ph.D. thesis***, Università di Pisa, 1996.

[35] Alessio Guglielmi and Lutz Strassburger, *Non-commutativity and MELL in the calculus of structures*, ***Csl 2001*** (L. Fribourg, editor), LNCS, vol. 2142, Springer-Verlag, 2001, pp. 54–68.

[36] Juan C. Guzmán and Paul Hudak, *Single-threaded polymorphic lambda calculus*, ***Proceedings of the Fifth Annual Symposium on Logic in Computer Science*** (Philadelphia,

Pennsylvania), IEEE Computer Society Press, June 1990, pp. 333–343.

[37] Lars Hallnäs and Peter Schroeder-Heister, *A proof-theoretic approach to logic programming. ii. Programs as definitions*, ***Journal of Logic and Computation***, vol. 1 (1991), no. 5, pp. 635–660.

[38] James Harland and David Pym, *Resource-distribution via boolean constraints*, To appear, *Transactional on Computational Logic*, 2002.

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

[40] Robert Harper, Furio Honsell, and Gordon Plotkin, *A framework for defining logics*, ***Journal of the ACM***, vol. 40 (1993), no. 1, pp. 143–184.

[41] R. Harrop, *Concerning formulas of the types* $A \to B \vee C$, $A \to (Ex)B(x)$ *in intuitionistic formal systems*, ***The Journal of Symbolic Logic***, (1960), pp. 27–32.

[42] Joshua Hodas, *Specifying filler-gap dependency parsers in a linear-logic programming language*, ***Proceedings of the joint international conference and symposium on logic programming*** (K. Apt, editor), 1992, pp. 622–636.

[43] Joshua Hodas and Dale Miller, *Representing objects in a logic programming language with scoping constructs*, ***1990 international conference in logic programming*** (David H. D. Warren and Peter Szeredi, editors), MIT Press, June 1990, pp. 511–526.

[44] ———, *Logic programming in a fragment of intuitionistic linear logic: Extended abstract*, ***Sixth Annual Symposium on Logic in Computer Science*** (Amsterdam) (G. Kahn, editor), July 1991, pp. 32–42.

[45] ———, *Logic programming in a fragment of intuitionistic linear logic*, ***Information and Computation***, vol. 110 (1994), no. 2, pp. 327–365.

[46] Joshua Hodas, Kevin Watkins, Naoyuki Tamura, and Kyoung-Sun Kang, *Efficient implementation of a linear logic programming language*, ***Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*** (Joxan Jaffar, editor), 1998, pp. 145 – 159.

[47] Joshua S. Hodas, *Logic programming in intuitionistic linear logic: Theory, design, and implementation*, ***Ph.D. thesis***, University of Pennsylvania, Department of Computer and Information Science, May 1994, Available as University of Pennsylvania Technical Reports MS-CIS-92-28 or LINC LAB 269.

[48] Max Kanovich, *The complexity of Horn fragments of linear logic*, ***Annals of Pure and Applied Logic***, vol. 69 (1994), pp. 195–241.

[49] M.I. Kanovich, M. Okada, and A. Scedrov, *Specifying real-time finite-state systems in linear logic*, ***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.

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

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

[52] Naoki Kobayashi and Akinori Yonezawa, *Asynchronous communication model based on linear logic*, ***Formal Aspects of Computing***, vol. 3 (1994), pp. 279–294, Short version appeared in Joint International Conference and Symposium on Logic Programming, Washington, DC, November 1992, Workshop on Linear Logic and Logic Programming.

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

[54] ———, *Interaction nets*, ***Seventeenth Annual Symposium on Principles of Programming Languages*** (San Francisco, California), ACM Press, 1990, pp. 95–108.

[55] J. Lambek, *The mathematics of sentence structure*, ***American Mathematical Monthly***, vol. 65 (1958), pp. 154–169.

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

[57] 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***, vol. 228 (1999), no. 1–2, pp. 175–210, Special issue on papers presented at MFPS'95.

[58] RAYMOND MCDOWELL and DALE MILLER, *A logic for reasoning with higher-order abstract syntax*, ***Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*** (Warsaw, Poland) (Glynn Winskel, editor), IEEE Computer Society Press, July 1997, pp. 434–445.

[59] ———, *Cut-elimination for a logic with definitions and induction*, ***Theoretical Computer Science***, vol. 232 (2000), pp. 91–119.

[60] RAYMOND MCDOWELL, DALE MILLER, and CATUSCIA PALAMIDESSI, *Encoding transition systems in sequent calculus*, ***Theoretical Computer Science***, vol. 197 (2001), no. 1-2, To appear.

[61] DALE MILLER, *Lexical scoping as universal quantification*, ***Sixth International Logic Programming Conference*** (Lisbon, Portugal), MIT Press, June 1989, pp. 268–283.

[62] ———, *A logical analysis of modules in logic programming*, ***Journal of Logic Programming***, vol. 6 (1989), no. 1-2, pp. 79–108.

[63] ———, *Abstract syntax and logic programming*, ***Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming***, LNAI, no. 592, Springer-Verlag, 1992, pp. 322–337.

[64] ———, *The π-calculus as a theory in linear logic: Preliminary results*, ***Proceedings of the 1992 Workshop on Extensions to Logic Programming*** (E. Lamma and P. Mello, editors), LNCS, no. 660, Springer-Verlag, 1993, pp. 242–265.

[65] ———, *A multiple-conclusion meta-logic*, ***Ninth Annual Symposium on Logic in Computer Science*** (Paris) (S. Abramsky, editor), IEEE Computer Society Press, July 1994, pp. 272–281.

[66] ———, *Forum: A multiple-conclusion specification language*, ***Theoretical Computer Science***, vol. 165 (1996), no. 1, pp. 201–232.

[67] DALE MILLER and GOPALAN NADATHUR, *Higher-order logic programming*, ***Proceedings of the Third International Logic Programming Conference*** (London) (Ehud Shapiro, editor), June 1986, pp. 448–462.

[68] DALE MILLER, GOPALAN NADATHUR, FRANK PFENNING, and ANDRE SCEDROV, *Uniform proofs as a foundation for logic programming*, ***Annals of Pure and Applied Logic***, vol. 51 (1991), pp. 125–157.

[69] ROBIN MILNER, JOACHIM PARROW, and DAVID WALKER, *A calculus of mobile processes, Part I*, ***Information and Computation***, (1992), pp. 1–40.

[70] MICHAEL MOORTGAT, *Categorial type logics*, ***Handbook of Logic and Language*** (Johan van Benthem and Alice ter Meulen, editors), Elsevier, Amsterdam, 1996, pp. 93–177.

[71] GLYN MORRILL, *Higher-order linear logic programming of categorial deduction*, ***7th Conference of the Association for Computational Linguistics*** (Dublin, Ireland), 1995, pp. 133–140.

[72] GOPALAN NADATHUR and DALE MILLER, *Higher-order Horn clauses*, ***Journal of the ACM***, vol. 37 (1990), no. 4, pp. 777–814.

[73] ———, *Higher-order logic programming*, ***Handbook of Logic in Artificial Intelligence and Logic Programming*** (Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors), vol. 5, Clarendon Press, Oxford, 1998, pp. 499 – 590.

[74] P. O'HEARN and D. PYM, *The logic of bunched implications*, ***The Bulletin of Symbolic Logic***, vol. 5 (1999), no. 2, pp. 215–243.

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

[76] E. PIMENTEL P. LÓPEZ, *A lazy splitting system for Forum*, ***AGP'97: Joint Conference on Declarative Programming***, 1997.

[77] REMO PARESCHI, *Type-driven natural language analysis*, ***Ph.D. thesis***, University of Edinburgh, 1989.

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

[79] Jeff Polakow, *Ordered linear logic and applications*, ***Ph.D. thesis***, Department of Computer Science, Caregnie Mellon, August 2001, Available as Technical Report CMU-CS-01-152.

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

[81] ———, *Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic*, ***Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*** (New Orleans, Louisiana) (Andre Scedrov and Achim Jung, editors), 1999.

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

[83] David J. Pym, *On bunched predicate logic*, ***Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*** (Trento, Italy) (G. Longo, editor), IEEE Computer Society Press, 1999, pp. 183–192.

[84] David J. Pym and James A. Harland, *The uniform proof-theoretic foundation of linear logic programming*, ***Journal of Logic and Computation***, vol. 4 (1994), no. 2, pp. 175 – 207.

[85] Christian Retoré, *Pomset logic: a non-commutative extension of classical linear logic*, ***Proceedings of tlca***, vol. 1210, 1997, pp. 300–318.

[86] ———, *Pomset logic as a calculus of directed cographs*, ***Dynamic perspectives in logic and linguistics: Proof theoretical dimensions of communication processes*** (V. M. Abrusci and C. Casadio, editors), 1999, pp. 221–247.

[87] Giorgia Ricci, *On the expressive powers of a logic programming presentation of linear logic (FORUM)*, ***Ph.D. thesis***, Department of Mathematics, Siena University, December 1998.

[88] Paul Ruet, *Non-commutative logic II: sequent calculus and phase semantics*, ***Mathematical Structures in Computer Science***, vol. 10 (2000), no. 2, pp. 277–312.

[89] V. Saraswat, *A brief introduction to linear concurrent constraint programming*, Available as file://parcftp.xerox.com/pub/ccp/lcc/lcc-intro.dvi.Z., 1993.

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

[91] T. Tammet, *Proof strategies in linear logic*, ***Journal of Automated Reasoning***, vol. 12 (1994), pp. 273–304, Also available as Programming Methodology Group Report 70, Chalmers University, 1993.

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

[93] Philip Wadler, *Linear types can change the world!*, ***Programming concepts and methods***, North Holland, 1990, pp. 561–581.

[94] M. Winikoff and J. Harland, *Implementing the Linear Logic Programming Language Lygon*, ***Proceedings of the international logic programming symposium***, December 1995, pp. 66–80.

E-mail: dale@cse.psu.edu

COMPUTER SCIENCE AND ENGINEERING, 220 POND LAB,
PENNSYLVANIA STATE UNIVERSITY, UNIVERSITY PARK, PA 16802-6106 USA