A Multiple-Conclusion Meta-Logic

Dale Miller
Computer Science Department, University of Pennsylvania
Philadelphia, PA 19104-6389 USA
dale@saul.cis.upenn.edu

Abstract

The theory of cut-free sequent proofs has been used to motivate and justify the design of a number of logic programming languages. Two such languages, λProlog and its linear logic refinement, Lolli [12], provide for various forms of abstraction (modules, abstract data types, higher-order programming) but lack primitives for concurrency. The logic programming language, LO (Linear Objects) [2] provides for concurrency but lacks abstraction mechanisms. In this paper we present Forum, a logic programming presentation of all of linear logic that modularly extends the languages λProlog, Lolli, and LO. Forum, therefore, allows specifications to incorporate both abstractions and concurrency. As a meta-language, Forum greatly extends the expressiveness of these other logic programming languages. To illustrate its expressive strength, we specify in Forum a sequent calculus proof system and the operational semantics of a functional programming language that incorporates such non-functional features as counters and references.

1 Introduction

In [17] a proof theoretic foundation for logic programming was proposed in which logic programs are collections of formulas used to specify the meaning of non-logical constants and computation is identified with *goal-directed* search for proofs. Using the sequent calculus, this can be formalized by having the sequent $\Sigma; \Delta \longrightarrow G$ denote the state of an idealized logic programming interpreter, where the current set of non-logical constants (the signature) is Σ, the current logic program is the set of formulas Δ, and the formula to be established, called the query or goal, is G. All the non-logical constants in G and the formulas in Δ are contained in Σ. A *goal-directed* or *uniform* proof is then a cut-free proof in which every occurrence of a sequent whose right-hand side is non-atomic is the conclusion of a right-introduction rule. The bottom-up search for uniform proofs is goal-directed to the extent that if the goal has a logical connective as its head, that occurrence of that connective must be introduced: the left-hand side of a sequent is only considered when the goal is atomic. A logic programming language is then a logical system for which uniform proofs are complete. The logics underlying λProlog and Lolli [12] satisfy such a completeness result.

When extending this notion of goal-directed search to multiple-conclusion sequents, the following problem is encountered: if the right-hand side of a sequent contains two or more non-atomic formulas, how should the logical connectives at the head of those formulas be introduced? There seems to be two choices. One choice simply requires that one of the possible introductions be done [10]. This has the disadvantage that there might be an interdependency between right-introduction rules in that one may need to appear lower in a proof than another, in which case, logical connectives in the goal would not be reflected directly and simply into the structure of the proof. A second choice requires that all right-hand rules should be introduced simultaneously. Although the sequent calculus cannot deal directly with simultaneous rule application, reference to *permutabilities* of inference rules [13] can indirectly address simultaneity. That is, 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, in fact, be done and the resulting proofs are all equal modulo permutations of right-introduction rules.

Using this second approach, we generalize the previous definition of uniform proof as follows: a cut-free sequent proof Ξ is *uniform* if for every subproof Ξ' of Ξ and for every non-atomic formula occurrence B in the right-hand side of the end-sequent of Ξ', there is a proof Ξ'' that is equal to Ξ' up to a permutation of inference rules and is such that the last inference rule in Ξ'' introduces the top-level logical connective of B. It is shown in [16] that the π-calculus [18] can be seen as a particular logic program in this sense.

In this paper, we employ the logical connectives of Girard [8] (typeset as in that paper) and the quantification and term structures of Church's Simple Theory of Types [5]. A *signature* is a finite set of pairs, written $c\colon\tau$, where c is a token and τ is a simple type (over some fixed set of base types). A closed, simply typed λ-term t is a Σ-*term* if all the non-logical constants in t are declared types in Σ. The base type o is used to denote formulas, and the various logical constants are given types over o. For example, the binary logical connectives have the type $o \to o \to o$ and the quantifiers $\forall_\tau$ have the type $(\tau \to o) \to o$. A Σ-term B of type o is also called a Σ-*formula*. The infix symbol $\Rightarrow$ denotes intuitionistic implication; that is, $B \Rightarrow C$ is equivalent to $!\,B \multimap C$, and the infix symbol $\circ\!-$ (which associates to the left) denotes the converse of $\multimap$. The expression $B \equiv C$ abbreviates the formula $(B \multimap C)\ \&\ (C \multimap B)$: if this formulas is provable in linear logic, we say that B and C are *logically equivalent.*

All of linear logic can be seen as a logic programming language since there is a presentation of linear logic for which uniform proofs are complete. To motivate the design of this presentation, which we call *Forum*, we first describe the four logic programming languages that it extends. *Horn clauses*, the logical foundation of Prolog, are formulas of the form $\forall\bar{x}(G \Rightarrow A)$ where G may contain occurrences of $\&$ and $\top$. (We shall use $\bar{x}$ as a syntactic variable ranging over a list of variables and A as a syntactic variables ranging over atomic formulas.) In such clauses, occurrences of $\Rightarrow$ and $\forall$ are restricted so that they do not occur to the left of an implication. As a result of this restriction, uniform proofs involving Horn clauses do not contain right-introduction rules for $\Rightarrow$ and $\forall$. *Hereditary Harrop formulas* [17], the logical foundation of λProlog, result from removing the restriction on $\Rightarrow$ and $\forall$ in Horn clauses: that is, such formulas can be built freely from $\top$, $\&$, $\Rightarrow$, and $\forall$. The logic at the foundation of Lolli is the result of adding $\multimap$ to the connectives present in hereditary Harrop formulas: that is, Lolli programs are freely built from $\top$, $\&$, $\multimap$, $\Rightarrow$, and $\forall$. (Some presentations of hereditary Harrop formulas and Lolli allow certain occurrences of disjunctions ($\oplus$) and existential quantifiers: since such occurrences can be defined within the logic programming setting (as we shall see), they are not considered directly here.) The formulas used in LO are of the form $\forall\bar{x}(G \multimap A_1 \parr \cdots \parr A_n)$ where $n \geq 1$ and G may contain occurrences of $\&$, $\top$, $\parr$, $\bot$. Similar to the Horn clause case, occurrences of $\multimap$ and $\forall$ are restricted so that they do not occur to the left of an implication.

The reason that Lolli does not include LO is the presence of $\parr$ and $\bot$ in the latter. This suggests the following definition for Forum: it is the linear logic theory of the formulas freely generated from $\top$, $\&$, $\parr$, $\bot$, $\multimap$, $\Rightarrow$, and $\forall$. It is this definition that we study in the rest of this paper.

Since the logics underlying Prolog, λProlog, Lolli, LO, and Forum differ in what logical connectives are allowed at what polarity, richer languages modularly contain weaker languages. This is a direct result of the cut-elimination theorem for linear logic. Thus a Forum program that does not happen to use $\bot$, $\parr$, and $\multimap$ will, in fact, have the same uniform proofs as are described for λProlog. Similarly, a program containing just a few occurrences of these connectives can be understood as a λProlog program that takes a few exceptional steps, but otherwise behaves as a λProlog program.

Forum is a presentation of all of linear logic since it contains a complete set of connectives. The connectives missing from Forum are directly definable using the following logical equivalences.

$$\begin{array}{ccc} B^\perp \equiv B \multimap \bot & 0 \equiv \top \multimap \bot & 1 \equiv \bot \multimap \bot \\ \multicolumn{3}{c}{!\,B \equiv (B \Rightarrow \bot) \multimap \bot \qquad ?\,B \equiv (B \multimap \bot) \Rightarrow \bot} \\ \multicolumn{3}{c}{B \oplus C \equiv (B^\perp \ \&\ C^\perp)^\perp \qquad B \otimes C \equiv (B^\perp \parr C^\perp)^\perp} \\ \multicolumn{3}{c}{\exists x.B \equiv (\forall x.B^\perp)^\perp} \end{array}$$

The other logic programming languages we have mentioned can, of course, capture the expressiveness of full logic by introducing non-logical constants and programs to describe their meaning. Felty in [6] uses a meta-logical presentation to specify full logic at the object-level. Andreoli [1] provides a "compilation-like" translation of linear logic into LinLog (of which LO is a subset). Forum has a more immediate relationship to all of linear logic since no non-logical symbols need to be used to provide complete coverage of linear logic.

As a presentation of linear logic, Forum may appear rather strange since it uses neither the cut rule (uniform proofs are cut-free) nor the dualities that follow from uses of negation (since negation is not a primitive). The execution of a Forum program (in the logic programming sense of the search for a proof) makes no use of cut or of the basic dualities. These aspects of linear logic, however, are important in meta-level arguments about specifications written in Forum. For example, a specification of a sequent calculus proof system for intuitionistic logic can be transformed into a natural deduction proof system by a use of linear logic's negation (see Section 3). The choice of primitives for this presentation makes it easy to keep close to the usual computational significance of backchaining, and the presence of the two implications, $\multimap$ and $\Rightarrow$, makes the specification of object-level inference rules natural.

2 Proof Search

Inference rules in cut-free proofs over formulas containing only the logical constants $\top$, $\&$, $\invamp$, $\bot$, $\multimap$, $\Rightarrow$, and $\forall$ have numerous opportunities to be permuted over each other. In particular, any two occurrences of right-rules permute over each other, any two occurrences of left-rules permute over each other, and any left rule occurring immediately below a right-rule can be permuted up. These observations about permutabilities can be integrated into a special proof system, given in Figure 1. Here, two styles of sequents are considered. These sequents are written as $\Sigma : \Psi; \Delta \longrightarrow \Gamma$ and $\Sigma : \Psi; \Delta \xrightarrow{B} \Gamma$, where Σ is a signature, Ψ is a *set* of Σ-formulas, Δ is a *multiset* of Σ-formulas, Γ is a *list* of Σ-formulas, and B is a Σ-formula. The intended meanings of these two sequents in linear logic are $!\,\Psi, \Delta \longrightarrow \Gamma$ and $!\,\Psi, \Delta, B \longrightarrow \Gamma$, respectively. (Here, $!\,\Psi$ denotes the multiset that results from placing $!$ on each of the formulas in the set Ψ.) In the proof system of Figure 1, the only right rules are those for sequents of the form $\Sigma : \Psi; \Delta \longrightarrow \Gamma$. In fact, the only formula in Γ that can be introduced is the left-most, non-atomic formula in Γ. This style of selection is specified by using the syntactic variable $\mathcal{A}$ to denote a (possibly empty) list of atomic formulas. Thus, the right-hand side of a sequent matches $\mathcal{A}, B \,\&\, C, \Gamma$ if it contains a formulas that is a top-level $\&$ for which only atomic formulas occur to its left. Both $\mathcal{A}$ and Γ may be empty. Left rules are applied only to the formula B that labels the sequent arrow in $\Sigma : \Psi; \Delta \xrightarrow{B} \mathcal{A}$. The notation $\mathcal{A}_1 + \mathcal{A}_2$ matches a list $\mathcal{A}$ if $\mathcal{A}_1$ and $\mathcal{A}_2$ are lists that can be interleaved to yield $\mathcal{A}$: that is, the order of members in $\mathcal{A}_1$ and $\mathcal{A}_2$ is as in $\mathcal{A}$, and (ignoring the order of elements) $\mathcal{A}$ denotes the multiset set union of the multisets represented by $\mathcal{A}_1$ and $\mathcal{A}_2$.

Notice that all the right-rules treat the context (Σ, Ψ, Γ, and $\mathcal{A}$) as black boxes: they either discard the context ($\top$-R), copy it ($\&$-R), or retain it (all other right-rules).

The following theorem yields as an immediate corollary that Forum is a logic programming language. We shall use $\vdash$ to denote provability in linear logic. In particular, $\Sigma : \Psi; \Delta \vdash \Gamma$ means that the sequent $\Sigma : !\,\Psi, \Delta \longrightarrow \Gamma$ has a proof (in linear logic); the notation $\Sigma : \Psi \vdash \Gamma$ means that the sequent $\Sigma : !\,\Psi \longrightarrow \Gamma$ has a proof; and the notation $\Sigma \vdash \Gamma$ means that the sequent $\Sigma :\longrightarrow \Gamma$ has a proof.

Theorem 1 *Let Σ be a signature and let G be a Σ-formula of linear logic all of whose logical connectives are in the set $\{\top, \&, \bot, \invamp, \multimap, \Rightarrow, \forall\}$. Then $\Sigma \vdash G$ if and only if the sequent $\Sigma :; \longrightarrow G$ is provable in the proof system in Figure 1.*

Proof Soundness follows quickly from the encoding described above of the two sequents used in Figure 1 into linear logic sequents. Completeness follows by showing that any cut-free proof in linear logic over Forum's connectives can be transformed via permutation of inference rules into a proof that corresponds directly to proofs built using the rules in Figure 1. Similar style completeness proofs can be found in [12, 15]. ∎

The completeness result could also be proved using a result of Andreoli about "focusing" proofs. Andreoli considered one-sided sequents and classified all the logical connectives of linear logic as being either asynchronous or synchronous. In our setting, an occurrence of a connective on the right of a sequent arrow is asynchronous and on the left is synchronous. As is shown in [1], asynchronous connectives can be introduced in any order without reference to context and with no need to backtrack. Here, this corresponds to the fact that the right-hand side of a sequent can be decomposed until there are only atomic formulas remaining on the right (we are, of course, reading proof rules bottom-up). Also, since the order of decomposition is not important, formulas on the right can proceed in a left-to-right fashion. Synchronous connectives can be introduced after all asynchronous connectives have been introduced, and synchronous subformulas of synchronous formulas can be process immediately: that is, when processing a synchronous formula, we can "focus" the processing on its immediate synchronous subformulas. Processing of synchronous formulas can in general require backtracking. It has been known that backchaining is a "focused" event (for example, Pfenning has described backchaining as "immediate implication"); Andreoli's results nicely formalizes and generalizes this observation. (The proof system in Figure 1 was motivated in large part by a proof system in [1].)

An analogy exists between the embedding of all of linear logic into Forum and the embedding of classical logic into intuitionistic logic via the double negation

$$\frac{}{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},\top,\Gamma}\ \top\text{-}R \qquad \frac{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},B,\Gamma \quad \Sigma:\Psi;\Delta\longrightarrow\mathcal{A},C,\Gamma}{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},B\,\&\,C,\Gamma}\ \&\text{-}R$$

$$\frac{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},\Gamma}{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},\bot,\Gamma}\ \bot\text{-}R \qquad \frac{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},B,C,\Gamma}{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},B\parr C,\Gamma}\ \parr\text{-}R$$

$$\frac{\Sigma:\Psi;B,\Delta\longrightarrow\mathcal{A},C,\Gamma}{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},B\multimap C,\Gamma}\ \multimap\text{-}R \qquad \frac{\Sigma:B,\Psi;\Delta\longrightarrow\mathcal{A},C,\Gamma}{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},B\Rightarrow C,\Gamma}\ \Rightarrow\text{-}R$$

$$\frac{y{:}\,\tau,\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},B[y/x],\Gamma}{\Sigma:\Psi;\Delta\longrightarrow\mathcal{A},\forall_\tau x.B,\Gamma}\ \forall\text{-}R \qquad \frac{\Sigma:\Psi;\Delta\xrightarrow{B}\mathcal{A}}{\Sigma:\Psi;B,\Delta\longrightarrow\mathcal{A}}\ \text{decide1} \qquad \frac{\Sigma:B,\Psi;\Delta\xrightarrow{B}\mathcal{A}}{\Sigma:B,\Psi;\Delta\longrightarrow\mathcal{A}}\ \text{decide2}$$

$$\frac{}{\Sigma:\Psi;\xrightarrow{A}A}\ \mathit{initial} \qquad \frac{}{\Sigma:\Psi;\xrightarrow{\bot}}\ \bot\text{-}L \qquad \frac{\Sigma:\Psi;\Delta\xrightarrow{B}\mathcal{A}}{\Sigma:\Psi;\Delta\xrightarrow{B\&C}\mathcal{A}}\ \&\text{-}L \qquad \frac{\Sigma:\Psi;\Delta\xrightarrow{C}\mathcal{A}}{\Sigma:\Psi;\Delta\xrightarrow{B\&C}\mathcal{A}}\ \&\text{-}L$$

$$\frac{\Sigma:\Psi;\Delta_1\xrightarrow{B}\mathcal{A}_1 \quad \Sigma:\Psi;\Delta_2\xrightarrow{C}\mathcal{A}_2}{\Sigma:\Psi;\Delta_1,\Delta_2\xrightarrow{B\parr C}\mathcal{A}_1+\mathcal{A}_2}\ \parr\text{-}L \qquad \frac{t\text{ is a }\Sigma\text{-term of type }\tau \quad \Sigma:\Psi;\Delta\xrightarrow{B[t/x]}\mathcal{A}}{\Sigma:\Psi;\Delta\xrightarrow{\forall_\tau x.B}\mathcal{A}}\ \forall\text{-}L$$

$$\frac{\Sigma:\Psi;\Delta_1\longrightarrow B,\mathcal{A}_1 \quad \Sigma:\Psi;\Delta_2\xrightarrow{C}\mathcal{A}_2}{\Sigma:\Psi;\Delta_1,\Delta_2\xrightarrow{B\multimap C}\mathcal{A}_1+\mathcal{A}_2}\ \multimap\text{-}L \qquad \frac{\Sigma:\Psi;\longrightarrow B \quad \Sigma:\Psi;\Delta\xrightarrow{C}\mathcal{A}}{\Sigma:\Psi;\Delta\xrightarrow{B\Rightarrow C}\mathcal{A}}\ \Rightarrow\text{-}L$$

Figure 1: The rule ∀-R has the proviso that y is not declared in the signature Σ.

translation. In classical logic, contraction and weakening can be used on both the left and right of the sequent arrow: in intuitionistic logic, they can only be used on the left. The familiar double negation translation of classical logic into intuitionistic logic makes it possible for the formula $B^{\bot\bot}$ on the right to be moved to the left as $B^{\bot}$, where contractions and weakening can be applied to it, and then moved back to the right as B. In this way, classical reasoning can be regained indirectly. Similarly, in linear logic when there are, for example, non-permutable right-rules, one of the logical connectives involved can be rewritten so that the non-permutability is transfer to one between a left rule above a right rule (the only kind of non-permutability in Forum proofs). For example, the bottom-up construction of a proof of the sequent $\longrightarrow a\otimes b, a^{\bot}\parr b^{\bot}$ must first introduce the $\parr$ prior to the $\otimes$: the context splitting required by $\otimes$ must be delayed until after the $\parr$ is introduced. If this sequent is translated into Forum we would have the sequent $\longrightarrow (a^{\bot}\parr b^{\bot})\multimap\bot, a^{\bot}\parr b^{\bot}$. In this case, $\multimap$ and $\parr$ can be introduced in any order, giving rise to the sequent $a^{\bot}\parr b^{\bot}\longrightarrow a^{\bot}, b^{\bot}$. Introducing the $\parr$ now causes the context to be split, but this occurs after the right-introduction of $\parr$. Thus, the encoding of some of the linear logic connectives into the set used by Forum essentially amounts to moving any "offending" non-permutabilities to where they are allowed.

Using various linear logic equivalences, all formulas in Forum are logically equivalent to formulas of the form $C_1\,\&\cdots\&\,C_n$ $(n\geq 0)$ where each C_i is of the form

$$\forall\bar{y}(G_1\hookrightarrow\cdots\hookrightarrow G_m\hookrightarrow(A_1\parr\cdots\parr A_p))\quad (m,p\geq 0).$$

Here, occurrences of $\hookrightarrow$ are either occurrences of $\multimap$ or $\Rightarrow$. An empty $\&$ is written as $\top$ and an empty $\parr$ is written as $\bot$. Formulas of this form will be called *clauses*. Given that the formulas in the Ψ portion of the sequents in Figure 1 are implicitly !'ed and given the linear logic equivalence $!(A\,\&\,B)\equiv\,!A\otimes\,!B$, we can further assume that all formulas in Ψ are clauses.

Certain occurrences of logical connectives that are not primitive to Forum can be removed from clauses using the following linear logic equivalences.

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

These equivalences can be used at times to avoid using the indirect equivalences mentioned earlier that employ negation.

We shall not discuss here practical considerations of how search for proofs using the inference rules in Figure 1 can be done, except to note a problem in using clauses with an empty head (a head that is $\bot$). For example, consider attempting to prove a sequent with right-hand side $\mathcal{A}$ and with the clause $\forall\bar{x}(G\multimap\bot)$

on the left-hand side. This clause can be used in a backchaining step, regardless of $\mathcal{A}$'s structure, yielding the new right-hand side $\theta G, \mathcal{A}$, for some substitution θ over the variables $\bar{x}$. Such a clause provides no overt clues as to when it can be effectively used to prove a given goal. See [15] for a discussion of a similar problem when negated clauses are allowed in logic programming based on minimal or intuitionistic logic. As we shall see below, the specification of the cut rule for an object-level logic employs just such a clause: the well known problems of searching for proofs involving cut thus apply equally well to the search for uniform proofs involving such clauses.

3 Specifying object-level provability

Given the proof-theoretic motivations of Forum and its inclusion of quantification at higher-order types, it is not surprising that it can be used to specify proof systems for various object-level logics. Below we illustrate how a sequent calculus proof system can be specified, and show how properties of linear logic can be used to infer properties of the object-level proof systems.

Provability in intuitionistic logic has well known presentations using sequent calculus and natural deduction, both of which were given by Gentzen in [7] as proof systems LJ and NJ, respectively. The LJ sequent $B_1, \ldots, B_n \longrightarrow B_0$ $(n \geq 0)$ can be represented by the meta-level formula

$$?\,\mathit{left}\ B_1 \parr \cdots \parr ?\,\mathit{left}\ B_n \parr \mathit{right}\ B_0,$$

where *left* and *right* are two meta-level predicates. To capture object-level contraction and weakening on the left-hand side, we employ the ? modal. Since no structural rules are available on the right-hand side of LJ sequents, no modal is used to encode that formula. Figure 2 is a specification of Gentzen's LJ calculus. (Expressions displayed as they are in Figure 2 are abbreviations for closed formulas: the intended formulas are those that result by applying ! to their universal closure.) The operational reading of these clauses is quite natural. For example, the first clause in Figure 2 encodes the right-introduction of $\supset$: operationally an occurrence of $A \supset B$ on the right is removed and replaced with an occurrence of B on the right and a (modalized) occurrence of A on the left (reading the right-introduction rule for $\supset$ from the bottom). Notice that all occurrences of the *left* predicate in Figure 2 are in the scope of ?. If occurrences of such modals in

$$
\begin{array}{rl}
\mathit{right}\ (A \supset B) & \multimapinv (?(\mathit{left}\ A) \parr \mathit{right}\ B).\\
?(\mathit{left}\ (A \supset B)) & \multimapinv \mathit{right}\ A \multimapinv ?(\mathit{left}\ B).\\
\mathit{right}\ (A \wedge B) & \multimapinv \mathit{right}\ A \mathbin{\&} \mathit{right}\ B.\\
?(\mathit{left}\ (A \wedge B)) & \multimapinv ?(\mathit{left}\ A).\\
?(\mathit{left}\ (A \wedge B)) & \multimapinv ?(\mathit{left}\ B).\\
\mathit{right}\ (B \vee C) & \multimapinv \mathit{right}\ B.\\
\mathit{right}\ (B \vee C) & \multimapinv \mathit{right}\ C.\\
?(\mathit{left}\ (B \vee C)) \parr \mathit{right}\ E & \multimapinv (?(\mathit{left}\ B) \parr \mathit{right}\ E)\\
 & \multimapinv (?(\mathit{left}\ C) \parr \mathit{right}\ E).\\
\mathit{right}\ B \parr ?(\mathit{left}\ B). & \\
\bot & \multimapinv ?(\mathit{left}\ B) \multimapinv \mathit{right}\ B.
\end{array}
$$

Figure 2: Specification of LJ: sequent calculus

the heads of clauses were dropped, it would be possible to prove meta-level goals that do not correspond to any LJ sequent: such goals could contain *left*-atoms that are not prefixed with the ? modal. (Of course, the actual Forum clauses result from replacing ? by its definition: this example and some others suggest that there are advantages to allowing ? as an additional primitive.)

Notice that with the left-introduction of $\vee$, the formula on the right (here E) must be copied: since such formulas are not under a ? modal, the inference rule must explicitly copy the right-hand formula. This is done by "synchronizing" (with a multiple-conclusion clause) both the disjunction that is being introduced and the right-hand formula, and then explicitly copying the right-hand formula within the rule (hence the two copies of *right* E on the right-side of that clause).

The penultimate clause in Figure 2 specifies the initial sequent rule while the final clause specifies the cut rule. The well known problems of searching for proofs containing cut rules are transferred to the meta-level as problems of using a clause with $\bot$ for a head within the search for cut-free proofs (see Section 2).

Let *LJ* be the set of clauses displayed in Figure 2 and let Σ_1 be the set of constants of the object-logic along with the two predicates *left* and *right*.

Proposition 2 (Correctness of LJ) *The sequent* $B_1, \ldots, B_n \longrightarrow B_0$ $(n \geq 0)$ *has an LJ proof if and only if* $\Sigma_1 : LJ \vdash ?\,\mathit{left}\ B_1 \parr \cdots \parr ?\,\mathit{left}\ B_n \parr \mathit{right}\ B_0$.

Proof For the forward direction, an LJ proof can be converted into a uniform proof of the corresponding meta-level formula by mapping the sequence of inference rules in the LJ proof to the sequence of clauses used in backchaining. Additionally, right-introductions for $\parr$ and & and weakening, contrac-

$$\begin{aligned}
\textit{right}\ (A \supset B) &\multimap (\textit{right}\ A \Rightarrow \textit{right}\ B).\\
\textit{right}\ B &\multimap \textit{right}\ A \multimap \textit{right}\ (A \supset B).\\
\textit{right}\ (A \wedge B) &\multimap \textit{right}\ A \mathbin{\&} \textit{right}\ B.\\
\textit{right}\ A &\multimap \textit{right}\ (A \wedge B).\\
\textit{right}\ B &\multimap \textit{right}\ (A \wedge B).\\
\textit{right}\ (B \vee C) &\multimap \textit{right}\ B.\\
\textit{right}\ (B \vee C) &\multimap \textit{right}\ C.\\
\textit{right}\ E &\multimap \textit{right}\ (B \vee C)\\
&\multimap (\textit{right}\ B \Rightarrow \textit{right}\ E)\\
&\multimap (\textit{right}\ C \Rightarrow \textit{right}\ E).
\end{aligned}$$

Figure 3: Specification of NJ: natural deduction

tion, and dereliction for ? will need to be inserted in a straightforward fashion. The converse direction is as simple: the sequence of backchaining steps determines the application of inference rules in a corresponding LJ proof. In the process of establishing this correspondence, it is important to observe how occurrences of atoms with the predicate *right* appear within uniform proofs: a simple induction on uniform proofs shows that if a multiple-conclusion goal is provable from *LJ*, that goal contains exactly one occurrence of *right*. ∎

So far we have only discussed the operational interpretation of the specification in Figure 2. It is delightful, however, to note that this specification has some meta-logical properties that go beyond its operational reading. In particular, the specifications for the initial and cut inference rules together are logically equivalent to the proposition $(\textit{right}\ B)^{\perp} \equiv\ ?(\textit{left}\ B)$. This equivalence implies the equivalence $(\textit{right}\ B) \equiv\ !(\textit{right}\ B)$. That is, we have the (not too surprising) fact that left and right are essentially duals, and that this is guaranteed by reference only to the specifications for the initial and cut rules. If we replace some occurrences of $?(\textit{left}\ B)$ in Figure 2 with $\textit{right}\ B$ and replace other occurrences with the equivalent $!(\textit{right}\ B)$, and rewrite the resulting clauses using linear logic equivalences, we get the clauses in Figure 3. Since the results of rewriting the last two clauses of in Figure 2 are linear tautologies, they are dropped. Figure 3 contains a specification of Gentzen's natural deduction system NJ. This specification is similar to those given using intuitionistic meta-logics [6, 19] and dependent typed calculi [11, 3]. Let *NJ* be the set of clauses displayed in Figure 3.

Proposition 3 (Correctness of NJ) *The formula B_0 has an NJ proof from the assumptions $B_1, \ldots, B_n$ $(n \geq 0)$ if and only if*

$$\Sigma_1 : NJ, \textit{right}\ B_1, \ldots, \textit{right}\ B_n \vdash \textit{right}\ B_0.$$

A proof of this Proposition can be done similar to the proof of Proposition 2. The discussion of the derivation of the natural deduction proof system from the sequent calculus proof system provides a proof of the following Proposition. For convenience, if Γ is a finite, non-empty set of formulas, let $\otimes\Gamma$ denote the formula that is the tensor of all the formula in Γ in some fixed but arbitrary order.

Proposition 4 *Let Eq be the tensor of the last two formulas in Figure 2. Then $\Sigma_1 \vdash (\otimes LJ) \equiv (\otimes NJ) \otimes Eq$.*

The following theorem, first proved by Gentzen in [7], is an almost immediate consequence of the preceding propositions.

Theorem 5 *The sequent $B_1, \ldots, B_n \longrightarrow B_0$ has an LJ proof if and only if B_0 has an NJ proof from the assumptions $B_1, \ldots, B_n$ $(n \geq 0)$.*

Proof If B_0 has an NJ proof from the assumptions $B_1, \ldots, B_n$, then by Proposition 3,

$$\Sigma_1 : NJ, \textit{right}\ B_1, \ldots, \textit{right}\ B_n \vdash \textit{right}\ B_0.$$

Using Proposition 4 and cut, we have

$$\Sigma_1 : LJ, \textit{right}\ B_1, \ldots, \textit{right}\ B_n \vdash \textit{right}\ B_0.$$

Since *Eq* follows from *LJ* and since *Eq* implies the equivalences $\forall B.(\textit{right}\ B)^{\perp} \equiv\ ?(\textit{left}\ B)$ and $\forall B.(\textit{right}\ B) \equiv\ !(\textit{right}\ B)$, additional uses of cut at the meta-level yield a proof of $\Sigma_1 : LJ \vdash ?\,\textit{left}\ B_1 \mathbin{⅋} \ldots \mathbin{⅋} ?\,\textit{left}\ B_n \mathbin{⅋} \textit{right}\ B_0$. Thus, by Proposition 2, it follows that the sequent $B_1, \ldots, B_n \longrightarrow B_0$ has an LJ proof.

For the converse assume that $B_1, \ldots, B_n \longrightarrow B_0$ has an LJ proof. Thus,

$$\Sigma_1 : LJ \vdash ?\,\textit{left}\ B_1 \mathbin{⅋} \ldots \mathbin{⅋} ?\,\textit{left}\ B_n \mathbin{⅋} \textit{right}\ B_0$$

and using cut and Proposition 4, we have

$$\Sigma_1 : NJ, Eq \vdash ?\,\textit{left}\ B_1 \mathbin{⅋} \ldots \mathbin{⅋} ?\,\textit{left}\ B_n \mathbin{⅋} \textit{right}\ B_0$$

and $\Sigma_1 : NJ, Eq, \textit{right}\ B_1, \ldots, \textit{right}\ B_n \vdash \textit{right}\ B_0$. The additional assumption of *Eq* stops us from using Proposition 3 immediately. It is straightforward to show, however, that any uniform proof that uses this additional assumption can be converted to a uniform proof that does not use that assumption. As as result, we can conclude that

$$\Sigma_1 : NJ, \mathit{right}\ B_1, \ldots, \mathit{right}\ B_n \vdash \mathit{right}\ B_0,$$

and by Proposition 3, that B_0 has an NJ proof from the assumptions $B_1, \ldots, B_n$. ∎

Most logical or type-theoretic systems that have been used for meta-level specifications of proof systems have been based on intuitionistic principles (for example, λProlog, Isabelle, LF). Although these systems have been successful at specifying numerous logical systems, they have important limitations. For example, while they can often provide elegant specifications of natural deduction proof systems, specifications of sequent calculus proofs are often unachievable without the addition of various non-logical constants for the sequent arrow and for forming lists of formulas (see, for example, [6]). Furthermore, these systems often have problems capturing substructural logics, such as linear logic, that do not contain the usual complement of structural rules. It should be clear from the above example that Forum allows for both the natural specification of sequent calculus and the possibility of handling substructural object-logics.

4 Operational Semantics Examples

Evaluation of pure functional programs has been successfully specified in intuitionistic meta-logics [9] and type theories [4, 20] using structured operational semantics and natural semantics. These specification systems are less successful at providing natural specifications of languages that incorporate references, control operators, and concurrency. We now consider how evaluation incorporating references can be specified in Forum.

Consider the presentation of call-by-value evaluation given by the following inference rules (in natural semantics style).

$$\frac{M \Downarrow (\mathit{abs}\ R) \qquad N \Downarrow U \qquad (R\ U) \Downarrow V}{(\mathit{app}\ M\ N) \Downarrow V}$$

$$\frac{}{(\mathit{abs}\ R) \Downarrow (\mathit{abs}\ R)}$$

Here, we assume that there is a type *tm* representing the domain of object-level, untyped λ-terms and that *app* and *abs* denote application (at type $tm \to tm \to tm$) and abstraction (at type $(tm \to tm) \to tm$). Object-level substitution is achieved at the meta-level by β-reduction of the meta-level application $(R\ U)$ in the above clause. A familiar way to represent these inference rules in meta-logic is to encode them as the following two clauses using the predicate *eval* of type $tm \to tm \to o$ (see, for example, [9]).

$$\begin{array}{l} \mathit{eval}\ (\mathit{app}\ M\ N)\ V \multimapinv \mathit{eval}\ M\ (\mathit{abs}\ R) \\ \qquad\qquad \multimapinv \mathit{eval}\ N\ U \multimapinv \mathit{eval}\ (R\ U)\ V. \\ \mathit{eval}\ (\mathit{abs}\ R)\ (\mathit{abs}\ R). \end{array}$$

In order to add side-effecting features, this specification must be made more explicit: in particular, the exact order in which M, N, and $(R\ U)$ are evaluated must be specified. Using a "continuation-passing" technique from logic programming [21], this ordering can be made more explicit using the following two clauses, this time using the predicate *eval* at type $tm \to tm \to o \to o$.

$$\begin{array}{l} \mathit{eval}\ (\mathit{app}\ M\ N)\ V\ K \multimapinv \\ \qquad \mathit{eval}\ M\ (\mathit{abs}\ R)\ (\mathit{eval}\ N\ U\ (\mathit{eval}\ (R\ U)\ V\ K)). \\ \mathit{eval}\ (\mathit{abs}\ R)\ (\mathit{abs}\ R)\ K \multimapinv K. \end{array}$$

From these clauses, the goal $(\mathit{eval}\ M\ V\ \top)$ is provable if and only if V is the call-by-value value of M. It is this "single-threaded" specification of evaluation that we shall modularly extend with a couple of non-functional features.

Consider adding to this specification a single global counter that can be read and incremented. To specify such a counter we add the integers to type `tm`, several simple functions over the integers, and the two symbols *read* and *inc* of type `tm`. The intended meaning of these constants is that evaluating the first returns the current value of the counter and evaluating the second increments the counter's value and returns the counter's old value. We also assume that integers are values: that is, for every integer i the clause $\forall k(\mathit{eval}\ i\ i\ k \multimapinv k)$ is part of the evaluator's specification.

$$\begin{array}{l} E_1 = \exists r[(r\ 0)^\perp \otimes \\ \ !\forall K, V(\mathit{eval}\ \mathit{read}\ V\ K \parr r\ V \multimapinv K \parr r\ V)) \otimes \\ \ !\forall K, V(\mathit{eval}\ \mathit{inc}\ V\ K \parr r\ V \multimapinv K \parr r\ (V+1))] \end{array}$$

$$\begin{array}{l} E_2 = \exists r[(r\ 0)^\perp \otimes \\ \ !\forall K, V(\mathit{eval}\ \mathit{read}\ (-V)\ K \parr r\ V \multimapinv K \parr r\ V) \otimes \\ \ !\forall K, V(\mathit{eval}\ \mathit{inc}\ (-V)\ K \parr r\ V \multimapinv K \parr r\ (V-1))] \end{array}$$

$$\begin{array}{l} E_3 = \exists r[(r\ 0) \otimes \\ \ !\forall K, V(\mathit{eval}\ \mathit{read}\ V\ K \multimapinv r\ V \otimes (r\ V \multimap K)) \otimes \\ \ !\forall K, V(\mathit{eval}\ \mathit{inc}\ V\ K \multimapinv r\ V \otimes (r\ (V+1) \multimap K)] \end{array}$$

Figure 4: Three specifications of a global counter.

Figure 4 contains three specifications, E_1, E_2, and E_3, of such a counter: all three specifications store the counter's value in a 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 [14]). The first two specifications store the counter's value on the right of the sequent arrow, and reading and incrementing a counter occur via a synchronization between evaluation and the atom storing the counter. 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 *read* and *inc* functions (counters such as these are described in [12]). 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, evaluation the *inc* symbol causes 1 to be subtracted from the counter's value: to compensate for this unusual choice, reading a counter in the second specification returns the minus of the current counter's value.

The use of $\otimes$, !, $\exists$, and negation in Figure 4, all of which are not primitive connectives of Forum, is for convenience in displaying these abstract data types. The equivalence

$$\exists r(R_1^{\perp} \otimes !\, R_2 \otimes !\, R_3) \multimap G \equiv \forall r(R_2 \Rightarrow R_3 \Rightarrow G \mathbin{⅋} R_1)$$

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.

Proposition 6 *Let Σ_2 be the signature containing* eval*, along with the constants of the object-level programming language, namely,* app*,* abs*,* inc*,* read*, the integers, and the various integer operations. We then have the following three entailments:*

$$\Sigma_2 : E_1 \vdash E_2, \qquad \Sigma_2 : E_2 \vdash E_3, \quad \textit{and} \quad \Sigma_2 : E_3 \vdash E_1.$$

Proof 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 by instantiating the right-hand existential quantifier with some term involving that eigenvariable. Assume that in all three cases, the eigenvariable selected is the predicate system s. The 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 $\Sigma_2 : E_1 \vdash eval\ M\ V\ \top$. Then by cut and the above proposition, we immediately have $\Sigma_2 : E_2 \vdash eval\ M\ V\ \top$.

It is possible to specify a more general notion of references from which a counter such as that described above can be built. Consider the specification in Figure 5. Here, the type *loc* is introduced to denote the location of references, and three constructors have been added to the object-level λ-calculus to manipulate references: one for reading a reference (*read*), one for setting a reference (*set*), and one for introducing a new reference within a particular lexical scope (*new*). For example, let m and n be expressions of type *tm* that do not contain free occurrences of r, and let F_1 be the expression

$$(new\ (\lambda r(set\ r\ (app\ m\ (read\ r))))\ n).$$

This expression represents the program that first evaluates n; then allocates a new, scoped reference cell, which is initialized with n's value; then overwrites this new reference cell with the result of applying m to the value currently stored in that cell. Since m does not contain a reference to r, it should be the case that this expression has the same operational behavior as the expression F_2 defined as

$$(app\ (abs\ \lambda x(app\ m\ x))\ n).$$

Below we illustrate the use of meta-level properties of linear logic to prove the fact that F_1 and F_2 have the same operational behaviors.

Let *Ev* be the set of formulas from Figure 5 plus the two formulas displayed above for the evaluation of *app* and *abs*, and let Σ_3 be the set of constants occurring in

$$
\begin{aligned}
\mathit{read} &: \mathit{loc} \to \mathit{tm}\\
\mathit{set} &: \mathit{loc} \to \mathit{tm} \to \mathit{tm}\\
\mathit{new} &: (\mathit{loc} \to \mathit{tm}) \to \mathit{tm} \to \mathit{tm}\\
\mathit{assign} &: \mathit{loc} \to \mathit{tm} \to o \to o\\
\mathit{ref} &: \mathit{loc} \to \mathit{tm} \to o
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{eval}\,(\mathit{set}\ L\ N)\ V\ K \multimapinv \mathit{eval}\ N\ V\ (\mathit{assign}\ L\ V\ K).\\
\mathit{eval}\,(\mathit{new}\ R\ E)\ V\ K \multimapinv\\
\qquad \mathit{eval}\ E\ U\ (\forall h(\mathit{ref}\ h\ U \parr \mathit{eval}\,(R\ h)\ V\ K)).\\
\mathit{eval}\,(\mathit{read}\ L)\ V\ K \parr \mathit{ref}\ L\ V \multimapinv K \parr \mathit{ref}\ L\ V.\\
\mathit{assign}\ L\ V\ K \parr \mathit{ref}\ L\ U \multimapinv K \parr \mathit{ref}\ L\ V.
\end{array}
$$

Figure 5: Specification of references.

Σ_2 and in Ev. An object-level program may have both a value and the side-effect of changing a store. Let S be a syntactic variable for a *store*, that is, a formula of the form $\mathit{ref}\ h_1\ u_1 \parr \ldots \parr \mathit{ref}\ h_n\ u_n$ $(n \geq 0)$, where all the constants $h_1, \ldots, h_n$ are distinct. Of course, we can think of a store as a finite function that maps locations to values stored in those locations. The *domain* of a store is the set of locations it assigns: in the above case, the domain of S is $\{h_1, \ldots, h_n\}$. A *garbaged state* is a formula of the form $\forall \bar{h}.S$, where S is a state and $\forall \bar{h}$ is the universal quantification of all the variables in the domain of S. Consider, for example, the program expression F_3 given as

$$(\mathit{new}\ \lambda r(\mathit{read}\ r)\ 5).$$

This program has the value 5 and the side-effect of leaving behind a garbaged store. More precisely, the evaluation of a program M in a store S yields a value V and new store S' and garbaged store G if the formula

$$\forall k[k \parr S' \parr G \multimap \mathit{eval}\ M\ V\ k \parr S]$$

is provable from the clauses in Ev and the signature Σ_3 extended with the domain of S. An immediate consequence of this forumula is that the formula $\mathit{eval}\ M\ V\ \top \parr S$ is provable: that is, the value of M is V if the store is initially S. The references specified here obey a block structured discipline: that is, the domains of S and S' are the same and any new references that are created in the evaluation of M are collected in the garbaged store G. For example, a consequence of the formulas in Ev is the formula

$$\forall k[k \parr \forall h(\mathit{ref}\ h\ 5) \multimap \mathit{eval}\ F_3\ 5\ k].$$

That is, evaluating expression F_3 yields the value 5 and the garbaged store $\forall h(\mathit{ref}\ h\ 5)$. An immediate consequence of this formula is the formula

$$\forall k[k \parr S \parr \forall h(\mathit{ref}\ h\ 5) \multimap \mathit{eval}\ F_3\ 5\ k \parr S];$$

that is, this expression can be evaluated in any store without changing it. Because of their quantification, garbaged stores are inexcessible: operationally (but not logically) $\forall h(\mathit{ref}\ h\ 5)$ can be considered the same as $\perp$ in a manner similar to the identification of $(x)\bar{x}y$ with the null process in the π-calculus [18].

We can now return to the problem of establishing how the programs F_1 and F_2 are related. They both contain the program phrases m and n, so we first assume that if n is evaluated in store S_0 it yields value v and mutates the store into S_1, leaving the garbaged store G_1. Similarly, assume that if m is evaluated in store S_1 it yields value $(\mathit{abs}\ u)$ and mutates the store into S_2 with garbaged store G_2. That is, assume the formulas

$$
\begin{array}{l}
\forall k[k \parr S_1 \parr G_1 \multimap \mathit{eval}\ n\ v\ k \parr S_0] \text{ and}\\
\forall k[k \parr S_2 \parr G_2 \multimap \mathit{eval}\ m\ (\mathit{abs}\ u)\ k \parr S_1].
\end{array}
$$

From these formulas and those in Ev, we can infer that

$$
\begin{array}{l}
\forall W \forall k[\mathit{eval}\,(u\ v)\ W\ k \parr S_2 \parr G_1 \parr G_2 \parr \forall h(\mathit{ref}\ h\ v)\\
\qquad \multimap \mathit{eval}\ F_1\ W\ k \parr S_0] \text{ and}\\
\forall W \forall k[\mathit{eval}\,(u\ v)\ W\ k \parr S_2 \parr G_1 \parr G_2\\
\qquad \multimap \mathit{eval}\ F_2\ W\ k \parr S_0].
\end{array}
$$

That is, if the expression $(u\ v)$ has value W in store S_2 then both expressions F_1 and F_2 yield value W in store S_1. Clearly resolution at the meta-level can be used to compose the meaning of different program fragments into the meaning of larger fragments. Hopefully, such a compositional approach to program meaning can be used to aid the analysis of programs using references.

5 Conclusions

We have given a presentation of linear logic whose proof theory modularly extends the proof theory of several known logic programming languages. The resulting specification language, named Forum, provides the abstract syntax and higher-order judgments available in intuitionistic-based meta-logics as well as primitives for synchronization and communications. We have specify directly various tasks in proof theory and the operational semantics of programming languages. Since the resulting specifications are natural and simple, properties of the meta-logic can be meaningful employed to provide interesting properties about the specified object-languages.

Acknowledgements. I benefited greatly from discussions with Jawahar Chirimar and Bob Harper and from comments of the conference reviewers and Eva Ma. The author has been funded in part by ONR N00014-93-1-1324, NSF CCR-91-02753, NSF CCR-92-09224, and DARPA N00014-85-K-0018.

References

[1] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3), 1992.

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

[3] A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *J. of Automated Reasoning*, 9:309-354, 1992.

[4] R. Burstall and F. Honsell. A natural deduction treatment of operational semantics. In *Foundations of Software Technology and Theoretical Computer Science*, pp. 250-269. Springer-Verlag LNCS 338, 1988.

[5] A. Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56-68, 1940.

[6] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *J. of Automated Reasoning*, 11(1):43-81, 1993.

[7] G. Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pp. 68-131. North-Holland Publishing Co., Amsterdam, 1969.

[8] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1-102, 1987.

[9] J. Hannan. Extended natural semantics. *J. of Functional Programming*, 3(2):123-152, 1993.

[10] J. Harland and D. Pym. On Goal-directed Provability in Classical Logic. Technical report 92/16, Dept. of Computer Science, Univ. of Melbourne, 1992.

[11] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. of the ACM*, 40(1):143-184, 1993.

[12] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *J. of Information and Computation*, 1994. (To appear).

[13] S. C. Kleene. Permutabilities of inferences in Gentzen's calculi LK and LJ. *Memoirs of the American Mathematical Society*, 10, 1952.

[14] D. Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pp. 268-283, Lisbon, Portugal, June 1989. MIT Press.

[15] D. Miller. A logical analysis of modules in logic programming. *J. of Logic Programming*, 6(1-2):79-108, 1989.

[16] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proc. of the 1992 Workshop on Extensions to Logic Programming*, Springer-Verlag LNCS 660, pp. 242-265. 1993.

[17] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125-157, 1991.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I. *Information and Computation*, pp. 1-40, September 1992.

[19] L. C. Paulson. The foundation of a generic theorem prover. *J. of Automated Reasoning*, 5:363-397, September 1989.

[20] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In LICS 1989, pp. 313-321, Monterey, CA.

[21] P. Tarau. Program transformations and WAM-support for the compilation of definite metaprograms. In *Logic Programming: Proc. of the First and Second Russian Conferences on Logic Programming*, Springer-Verlag LNAI 592, pp. 462-473, 1992.

Papers by Miller are available via anonymous ftp from `ftp.cis.upenn.edu` in `pub/papers/miller` or using WWW at `http://www.cis.upenn.edu/~dale`.