Forum:
A Multiple-Conclusion Specification Logic

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

Draft: January 3, 1996

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 [15], provide for various forms of abstraction (modules, abstract data types, and higher-order programming) but lack primitives for concurrency. The logic programming language, LO (Linear Objects) [2] provides some primitives for concurrency but lacks abstraction mechanisms. In this paper we present Forum, a logic programming presentation of all of linear logic that modularly extends λProlog, Lolli, and LO. Forum, therefore, allows specifications to incorporate both abstractions and concurrency. To illustrate the new expressive strengths of Forum, we specify in it a sequent calculus proof system and the operational semantics of a programming language that incorporates references and concurrency. We also show that the meta theory of linear logic can be used to prove properties of the object-languages specified in Forum.

This paper will appear in *Theoretical Computer Science*.

1 Introduction

In [25] 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. (We assume that all the non-logical constants in G and in the formulas of 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 Prolog, λProlog, and Lolli [15] satisfy such a completeness result.

The description of logic programming above is based on single-conclusion sequents: that is, on the right of the sequent arrow in $\Sigma : \Delta \longrightarrow G$ is a single formula. This leaves open the question of how to define logic programming in the more general setting where sequents may have multiple formulas on the right-hand side [8]. 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 [12]. This choice has the disadvantage that there might be interdependencies between right-introduction rules: thus, the meaning of the logical connectives in the goal would not be reflected directly and simply into the structure of a proof, a fact that complicates the operational semantics of the logic as a programming language. A second choice requires that all possible introductions on the right can be done simultaneously. Although the sequent calculus cannot deal directly with simultaneous rule application, reference to *permutabilities* of inference rules [16] 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 introduction rules. This approach, which makes the operational interpretation of specifications simple and natural, is used in this paper.

We employ the logical connectives of Girard [9] (typeset as in that paper) and the quantification and term structures of Church's Simple Theory of Types [6]. A *signature* Σ is a finite set of pairs, written $c : \tau$, where c is a token and

τ is a simple type (over some fixed set of base types). We assume that a given token is declared at most one type in a given signature. 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 $\forall_\tau$ and $\exists_\tau$ have the type $(\tau \to o) \to o$, for any type τ. Expressions of the form $\forall_\tau \lambda x.B$ and $\exists_\tau \lambda x.B$ will be written more simply as $\forall_\tau x.B$ and $\exists_\tau x.B$, or as $\forall x.B$ and $\exists x.B$ when the type τ is either unimportant or can be inferred from context. A Σ-term B of type o is also called a Σ-*formula*. In addition to the usual connectives present in linear logic, we also add the infix symbol $\Rightarrow$ to denote intuitionistic implication; that is, $B \Rightarrow C$ is equivalent to $!B \multimap C$. The expression $B \equiv C$ abbreviates the formula $(B \multimap C) \,\&\, (C \multimap B)$: if this formula is provable in linear logic, we say that B and C are *logically equivalent*.

In the next section, the design of Forum is motivated by considering how to modularly extend certain logic programming languages that have been designed following proof theoretic considerations. In Section 3, Forum is shown to be a logic programming language using the multiple conclusion generalization of uniform proofs. The operational semantics of Forum is described in Section 4 so that the examples in the rest of the paper can be understood from a programming point-of-view as well as the declarative point-of-view. Sequent calculus proof systems for some object-level logics are specified in Section 5, and various imperative and concurrency features of a object-level programming language are specified and analyzed in Sections 6 and 7.

Although Forum extends some existing logic programming languages based on linear logic, there have been other linear logic programming languages proposed that it does not extend or otherwise relate directly. In particular, the language ACL by Kobayashi and Yonezawa [17, 18] captures simple notions of asynchronous communication by identifying the send and read primitives with two complementary linear logic connectives. Also, 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 [19, 32].

2 Design issues

The following generalization of the definition of uniform proof was introduced in [23] where it was shown that a certain logic specification inspired by the π-calculus [27] can be seen as a logic program.

Definition 1 *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.*

Definition 2 *A logic with a sequent calculus proof system is an* abstract logic programming language *if restricting to uniform proofs does not lose completeness.*

Below are several examples of abstract logic programming languages.

- *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 formulas, occurrences of $\Rightarrow$ and $\forall$ are restricted so that they do not occur to the left of the implication $\Rightarrow$. As a result of this restriction, uniform proofs involving Horn clauses do not contain right-introduction rules for $\Rightarrow$ and $\forall$.

- *Hereditary Harrop formulas* [25], 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 $\forall$. Some presentations of hereditary Harrop formulas and Horn clauses allow certain occurrences of disjunctions ($\oplus$) and existential quantifiers [25]: since such occurrences do not add much to the expressiveness of these languages (as we shall see at the end of this section), they are not considered directly here.

- 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 $\forall$. As with hereditary Harrop formulas, it is possible to also allow certain occurrences of $\oplus$ and $\exists$, as well as the tensor $\otimes$ and the modal !.

- 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 the implication $\multimap$.

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, the intended superlanguage: allow formulas to be freely generated from $\top$, &, $\bot$, $\parr$, $\multimap$, $\Rightarrow$, and $\forall$. For various reasons, it is also desirable to add the modal ? directly to this list of connectives. Clearly, Forum contains the formulas in all the above logic programming languages.

Since the logics underlying Prolog, λProlog, Lolli, LO, and Forum differ in what logical connectives are allowed, 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 $\bot$, $\parr$, $\multimap$, and ? 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.

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

The collection of connectives in Forum are not minimal. For example, $?$ and $⅋$, can be defined in terms of the remaining connectives.

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

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 [7] 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. Of course, to achieve this complete coverage, many of the logical connectives of linear logic are encoded using negations (more precisely, using "implies bottom"), a fact that causes certain operational problems, as we shall see in Section 4.

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. In Sections 5 and 6 we show some examples of how linear logic's negation and cut-elimination theorem can be used to reason about Forum specifications.

The choice of these primitives for this presentation of linear logic makes it possible 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. For example, the proof figure

$$\frac{\begin{array}{c}(A)\\ \vdots\\ B\end{array} \quad C}{D}$$

Can be written at the meta-level using implications such as $(A \Rightarrow B) \multimap C \multimap D$. Since we intend to use Forum as a specification language for type checking rules, structured operational semantics, and proof systems, the presence of implications as primitives is desirable.

The logical equivalences

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

can be used to remove certain occurrences of $\otimes$, $\oplus$, $\exists$, $!$, and 1 when they occur to the left of implications. (In the last equivalence above, assume that x is not free in H.) These equivalences are more direct than those that employ the equivalences mentioned earlier that use negation via the "implies bottom" construction. As a result, we shall allow their use in Forum specifications and employ these equivalences to remove them when necessary.

Formulas of the form

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

where $G_1, \ldots G_m$ are arbitrary Forum formulas and $A_1, \ldots A_m$ are atomic formulas, are called *clauses*. Here, occurrences of $\hookrightarrow$ are either occurrences of $\multimap$ or $\Rightarrow$. An empty $⅋$ $(p = 0)$ is written as $\perp$. The formula $A_1 \mathbin{⅋} \cdots \mathbin{⅋} A_p$ is the *head* of such a clause. If $p = 0$ then we say that this clause has an *empty head*. The formulas of LinLog [1] are essentially clauses in which $p > 0$ and the formula $G_1, \ldots, G_m$ do not contain $\multimap$ and $\Rightarrow$ and where $?$ has only atomic scope.

3 Proof Search

In this section we consider the abstract character of cut-free proofs over the connectives of Forum. Let $\mathcal{L}_1$ be the set of all formulas over the logical connectives $\perp$, $⅋$, $\top$, $\&$, $\multimap$, $\Rightarrow$, $?$, and $\forall$. If $\mathcal{C}$ is a set or multiset of formulas, the notation $!\,\mathcal{C}$ denotes the corresponding set or multiset that results from placing $!$ on each of the formula occurrences in $\mathcal{C}$: the notation $?\,\mathcal{C}$ is defined similarly.

Let $\mathcal{F}$ be the sequent proof system given in Figure 1. In this proof system, sequents have the form

$$\Sigma\colon \Psi; \Delta \longrightarrow \Gamma; \Upsilon \quad \text{and} \quad \Sigma\colon \Psi; \Delta \xrightarrow{B} \Gamma; \Upsilon,$$

where Σ is a signature, Δ is a multiset of formulas, Γ is a list of formulas, Ψ and Υ are sets of formulas, and B is a formula. All of these formulas are from $\mathcal{L}_1$ and are also Σ-formulas. (The introduction of signatures into sequents is not strictly necessary but is desirable when this proof system is used for logic programming specifications [22].) The intended meanings of these two sequents in linear logic are

$$!\,\Psi, \Delta \longrightarrow \Gamma, ?\,\Upsilon \quad \text{and} \quad !\,\Psi, \Delta, B \longrightarrow \Gamma, ?\,\Upsilon,$$

respectively. In the proof system of Figure 1, the only right rules are those for sequents of the form $\Sigma\colon \Psi; \Delta \longrightarrow \Gamma; \Upsilon$. 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 list of atomic formulas. Thus, the right-hand side of a sequent matches $\mathcal{A}, B \mathbin{\&} C, \Gamma$ if it contains a formula that is a top-level & for which at most atomic formulas can 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 $\Sigma\colon \Psi; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon$. 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 $\mathcal{A}_2$.

As in Church's Simple Theory of Types, we assume the usual rules of α, β, and η-conversion and we identify terms up to α-conversion. A term is λ-normal if it contains no β and no η redexes. All terms are λ-convertible to a term in λ-normal form, and such a term is unique up to α-conversion. All formulas in sequents are in λ-normal form: in particular, the notation $B[t/x]$, used in $\forall$L and $\forall$R, denotes the λ-normal form of the β-redex $(\lambda x.B)t$.

We use the turnstile symbol as the mathematics-level judgment that a sequent is provable: that is, $\Delta \vdash \Gamma$ means that the two-sided sequent $\Delta \longrightarrow \Gamma$ has a linear logic proof. The sequents of $\mathcal{F}$ are similar to those used in the LU proof system of Girard [10] except that we have followed the tradition of [1, 14] in writing the "classical" context (here, Ψ and Υ) on the outside of the sequent and the "linear" context (here, Δ and Γ) nearest the sequent arrow: in LU these conventions are reversed.

Given the intended interpretation of sequents in $\mathcal{F}$, the following soundness theorem can be proved by simple induction on the structure of $\mathcal{F}$ proofs.

Theorem 1 (Soundness) *If the sequent $\Sigma\colon \Psi; \Delta \longrightarrow \Gamma; \Upsilon$ has an $\mathcal{F}$ proof then $!\,\Psi, \Delta \vdash \Gamma, ?\,\Upsilon$. If the sequent $\Sigma\colon \Psi; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon$ has an $\mathcal{F}$ proof then $!\,\Psi, \Delta, B \vdash \Gamma, ?\,\Upsilon$.*

Completeness of the $\mathcal{F}$ proof system is a more difficult matter, largely because proofs can be built only in a greatly constrained fashion. In sequent proof systems generally, left and right introduction rules can be interleaved, where as,

$$\frac{}{\Sigma\colon \Psi; \Delta \longrightarrow \mathcal{A}, \top, \Gamma; \Upsilon}\ \top R$$

$$\frac{\Sigma\colon \Psi; \Delta \longrightarrow \mathcal{A}, B, \Gamma; \Upsilon \quad \Sigma\colon \Psi; \Delta \longrightarrow \mathcal{A}, C, \Gamma; \Upsilon}{\Sigma\colon \Psi; \Delta \longrightarrow \mathcal{A}, B \,\&\, C, \Gamma; \Upsilon}\ \& R$$

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

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

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

$$\frac{\Sigma\colon B, \Psi; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon}{\Sigma\colon B, \Psi; \Delta \longrightarrow \mathcal{A}; \Upsilon}\ \mathit{decide}\,! \qquad \frac{\Sigma\colon \Psi; \Delta \longrightarrow \mathcal{A}, B; B, \Upsilon}{\Sigma\colon \Psi; \Delta \longrightarrow \mathcal{A}; B, \Upsilon}\ \mathit{decide}\,?$$

$$\frac{\Sigma\colon \Psi; \Delta \xrightarrow{B} \mathcal{A}; \Upsilon}{\Sigma\colon \Psi; B, \Delta \longrightarrow \mathcal{A}; \Upsilon}\ \mathit{decide}$$

$$\frac{}{\Sigma\colon \Psi; \cdot \xrightarrow{A} A; \Upsilon}\ \mathit{initial} \qquad \frac{}{\Sigma\colon \Psi; \cdot \xrightarrow{A} \cdot; A, \Upsilon}\ \mathit{initial}?$$

$$\frac{}{\Sigma\colon \Psi; \cdot \xrightarrow{\bot} \cdot; \Upsilon}\ \bot L \qquad \frac{\Sigma\colon \Psi; \Delta \xrightarrow{B_i} \mathcal{A}; \Upsilon}{\Sigma\colon \Psi; \Delta \xrightarrow{B_1 \& B_2} \mathcal{A}; \Upsilon}\ \& L_i \qquad \frac{\Sigma\colon \Psi; B \longrightarrow \cdot; \Upsilon}{\Sigma\colon \Psi; \cdot \xrightarrow{?\,B} \cdot; \Upsilon}\ ?\,\mathrm{L}$$

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

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

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

Figure 1: The $\mathcal{F}$ proof system. The rule $\forall$R has the proviso that y is not declared in the signature Σ, and the rule $\forall$L has the proviso that t is a Σ-term of type τ. In $\& L_i$, $i = 1$ or $i = 2$.

in $\mathcal{F}$, occurrences of introduction rules are constrained so that (reading from the bottom up) right rules are used entirely until the linear part of the right-hand side (Γ) is decomposed to only atoms, and it is only when the right-hand side is a list of atoms that left introduction rules are applied. Completeness of $\mathcal{F}$ can be proved by showing that any proof in linear logic can be converted to a proof in $\mathcal{F}$ by permuting enough inference rules. Since there are many opportunities for such permutations, such a completeness proof has many cases. Fortunately, Andreoli has provided a nice packaging of the permutation aspects of linear logic within a single proof system [1]. We show that the $\mathcal{F}$ proof system is simply a variation of the proof system he provided.

Let $\mathcal{L}_2$ be the set of formulas all of whose logical connectives are from the list $\bot$, $⅋$, $\top$, $\&$, $?$, $\forall$ (those used in $\mathcal{L}_1$ minus the two implications) along with the duals of these connectives, namely, 1, $\otimes$, 0, $\oplus$, $!$, and $\exists$. Negations of atomic formulas are also allowed, and we write $B^\perp$, for non-atomic formula B, to denote the formula that results from giving negations atomic scope using the de Morgan dualities of linear logic. A formula is *asynchronous* if it has a top-level logical connective that is either $\bot$, $⅋$, $\top$, $\&$, $?$, or $\forall$, and is *synchronous* if it has a top-level logical connective that is either 1, $\otimes$, 0, $\oplus$, $!$, and $\exists$. Figure 2 contains the $\mathcal{J}$ proof system. Andreoli showed in [1] that this proof system is complete for linear logic. Although he proved this only for the first-order fragment of linear logic, it lifts to the higher-order case we are considering given Girard's proof of cut-elimination for full, higher-order linear logic [9].

The following theorem shows that the $\mathcal{F}$ and $\mathcal{J}$ proof systems are similar, and in this way, the completeness for $\mathcal{F}$ is established. Before proving the completeness of $\mathcal{F}$ we state the following technical result used in the completeness theorem.

Lemma 2 *Let $\mathcal{A}$ and $\mathcal{A}'$ be lists of atoms that are permutations of each other. If the sequent $\Sigma\colon\Psi;\Delta \longrightarrow \mathcal{A},\Gamma;\Upsilon$ has an $\mathcal{F}$ proof then so too does $\Sigma\colon\Psi;\Delta \longrightarrow \mathcal{A}',\Gamma;\Upsilon$. Similarly, if the sequent $\Sigma\colon\Psi;\Delta \xrightarrow{B} \mathcal{A};\Upsilon$ has an $\mathcal{F}$ proof then so too does $\Sigma\colon\Psi;\Delta \xrightarrow{B} \mathcal{A}';\Upsilon$.*

Proof Completed by induction on the structure of proofs in $\mathcal{F}$. ∎

Theorem 3 (Completeness) *Let Σ be a signature, Δ be a multiset of $\mathcal{L}_1$ Σ-formulas, Γ be a list of $\mathcal{L}_1$ Σ-formulas, and Ψ and Υ be sets of $\mathcal{L}_1$ Σ-formulas. If $!\,\Psi,\Delta \vdash \Gamma,?\,\Upsilon$ then the sequent $\Sigma\colon\Psi;\Delta \longrightarrow \Gamma;\Upsilon$ has a proof in $\mathcal{F}$.*

Proof Assume that $!\,\Psi,\Delta \vdash \Gamma,?\,\Upsilon$. The main result of [1], extended to the higher-order case, implies that the sequent $\Sigma\colon\Psi^\perp,\Upsilon;\Delta^\perp \Uparrow \Gamma$ has a $\mathcal{J}$ proof. We now show how to convert such a proof into a proof of $\mathcal{F}$.

When B is an $\mathcal{L}_1$ formula, we define B^0 to be the $\mathcal{L}_2$ formula that results from replacing subformulas of B of the form $C \multimap D$ with $C^\perp ⅋ D$ and of the form $C \Rightarrow D$ with $?\,C^\perp ⅋ D$. The formula B^0 is either asynchronous or atomic

$$\frac{\Sigma\colon \Psi; \Delta \Uparrow L}{\Sigma\colon \Psi; \Delta \Uparrow \bot, L}\ [\bot] \qquad \frac{\Sigma\colon \Psi; \Delta \Uparrow F, G, L}{\Sigma\colon \Psi; \Delta \Uparrow F \mathbin{\rotatebox[origin=c]{180}{\&}} G, L}\ [\mathbin{\rotatebox[origin=c]{180}{\&}}] \qquad \frac{\Sigma\colon \Psi, F; \Delta \Uparrow L}{\Sigma\colon \Psi; \Delta \Uparrow ?\, F, L}\ [?]$$

$$\frac{}{\Sigma\colon \Psi; \Delta \Uparrow \top, L}\ [\top] \qquad \frac{\Sigma\colon \Psi; \Delta \Uparrow F, L \quad \Sigma\colon \Psi; \Delta \Uparrow G, L}{\Sigma\colon \Psi; \Delta \Uparrow F \,\&\, G, L}\ [\&]$$

$$\frac{y : \tau, \Sigma\colon \Psi; \Delta \Uparrow B[y/x], L}{\Sigma\colon \Psi; \Delta \Uparrow \forall_\tau x.B, L}\ [\forall] \qquad \frac{}{\Sigma\colon \Psi; \cdot \Downarrow 1}\ [1]$$

$$\frac{\Sigma\colon \Psi; \Delta_1 \Downarrow F \quad \Sigma\colon \Psi; \Delta_2 \Downarrow G}{\Sigma\colon \Psi; \Delta_1, \Delta_2 \Downarrow F \otimes G}\ [\otimes] \qquad \frac{\Sigma\colon \Psi; \cdot \Uparrow F}{\Sigma\colon \Psi; \cdot \Downarrow !\, F}\ [!]$$

$$\frac{\Sigma\colon \Psi; \Delta \Downarrow F_i}{\Sigma\colon \Psi; \Delta \Downarrow F_1 \oplus F_2}\ [\oplus_i] \qquad \frac{\Sigma\colon \Psi; \Delta \Downarrow B[t/x]}{\Sigma\colon \Psi; \Delta \Downarrow \exists_\tau x.B}\ [\exists]$$

$$\frac{\Sigma\colon \Psi; \Delta, F \Uparrow L}{\Sigma\colon \Psi; \Delta \Uparrow F, L}\ [R \Uparrow] \quad \text{provided that F is not asynchronous}$$

$$\frac{\Sigma\colon \Psi; \Delta \Uparrow F}{\Sigma\colon \Psi; \Delta \Downarrow F}\ [R \Downarrow] \quad \text{provided that F is either asynchronous or an atom}$$

$$\frac{}{\Sigma\colon \Psi; A \Downarrow A^\bot}\ [I_1] \qquad \frac{}{\Sigma\colon \Psi, A; \cdot \Downarrow A^\bot}\ [I_2]$$

$$\frac{\Sigma\colon \Psi; \Delta \Downarrow F}{\Sigma\colon \Psi; \Delta, F \Uparrow \cdot}\ [D_1] \qquad \frac{\Sigma\colon \Psi; \Delta \Downarrow F}{\Sigma\colon \Psi, F; \Delta \Uparrow \cdot}\ [D_2]$$

Figure 2: The $\mathcal{J}$ proof system. The rule $[\forall]$ has the proviso that y is not declared in Σ, and the rule $[\exists]$ has the proviso that t is a Σ-term of type τ. In $[\oplus_i]$, $i = 1$ or $i = 2$.

while $(B^0)^\perp$ is either synchronous or the negation of an atom. Notice that if B^0 is equal to $E \parr F$, then there are formulas C and D of $\mathcal{L}_1$ so that D^0 is F, and either B is $C \parr D$ and E is C^0, B is $C \multimap D$ and E is $(C^0)^\perp$, or B is $C \Rightarrow D$ and E is $?(C^0)^\perp$.

Let Ψ be a set of formulas of the form B^0 and $(B^0)^\perp$, where B is from $\mathcal{L}_1$; let Δ be a multiset of formulas that are either atomic or of the form $(B^0)^\perp$, where B is from $\mathcal{L}_1$; let L be a list of formulas of the form B^0, where B is from $\mathcal{L}_1$; and let F be $(B^0)^\perp$ where B is an $\mathcal{L}_1$ formula. Given these restrictions on Ψ and Δ, we define the following pair of functions on sets of such formulas:

$$[\Psi]_- = \{B \mid (B^0)^\perp \in \Psi\} \qquad [\Psi]_+ = \{B \mid B^0 \in \Psi\}.$$

Clearly the resulting sets are sets of $\mathcal{L}_1$ formulas. We similarly define $[\Delta]_-$ to be the corresponding multiset of $\mathcal{L}_1$. The corresponding value for $[\Delta]_+$ would be a multiset of atomic formulas, but in the proof below, we need to consider $[\Delta]_+$ to be a list, which is the underlying multiset put in some arbitrary but fixed order. We now prove by mutual induction on the heights of $\mathcal{J}$ proofs the following two facts.

- If $\Sigma\colon \Psi; \Delta \Uparrow L$ has a $\mathcal{J}$ proof then $\Sigma\colon [\Psi]_-; [\Delta]_- \longrightarrow [\Delta]_+, L; [\Psi]_+$ has an $\mathcal{F}$ proof.

- If $\Sigma\colon \Psi; \Delta \Downarrow (B^0)^\perp$ has a $\mathcal{J}$ proof then $\Sigma\colon [\Psi]_-; [\Delta]_- \xrightarrow{B} [\Delta]_+; [\Psi]_+$ has an $\mathcal{F}$ proof.

Consider the possible last rules in the proofs of $\Sigma\colon \Psi; \Delta \Uparrow L$ and $\Sigma\colon \Psi; \Delta \Downarrow (B^0)^\perp$.

Case: $[\top]$**.** In this case, L has $\top$ as its first element. Thus the corresponding $\mathcal{F}$ proof is $\top$R.

Case: $[1]$**.** In this case, B is $\perp$ and Δ is empty, and the corresponding $\mathcal{F}$ proof is $\perp$L.

Case: $[I_1]$**.** Thus B is the atomic formula A and Δ is the multiset containing just one occurrence of A. The corresponding $\mathcal{F}$ proof is *initial*.

Case: $[I_2]$**.** Thus B is the atomic formula A, Δ is the empty multiset, and Ψ is a set containing A. The corresponding $\mathcal{F}$ proof is *initial* ?.

Case: $[\perp]$**.** In this case, the corresponding $\mathcal{F}$ proof is built using $\perp$R.

Case: $[\parr]$**.** In this case, L is a list with head $E \parr F$ and tail L'. Let B be the $\mathcal{L}_1$ formula so that B^0 is $E \parr F$. There are three subcases to consider depending on structure of B.

Subcase: B is $C \parr D$ and E is C^0. In this case, the corresponding $\mathcal{J}$ proof is built using $\parr$R.

Subcase: B is $C \multimap D$ and E is $(C^0)^\perp$. In this case,

$$\Sigma\colon \Psi; \Delta \Uparrow (C^0)^\perp, D^0, L'$$

has a smaller proof. However, the only inference rule that can yield this sequent as a conclusion is $[R\Uparrow]$. Thus, $\Sigma\colon \Psi; \Delta, (C^0)^\perp \Uparrow D^0, L'$ has a smaller $\mathcal{J}$ proof. Using the inductive hypothesis and the $\multimap$R rule provides the desired $\mathcal{F}$ proof.

Subcase: B is $C \Rightarrow D$ and E is $?(C^0)^\perp$. In this case,

$$\Sigma\colon \Psi; \Delta \Uparrow ?(C^0)^\perp, D^0, L'$$

has a smaller proof. However, the only inference rule that can yield this sequent as a conclusion is [?]. Thus, $\Sigma\colon \Psi, (C^0)^\perp; \Delta \Uparrow D^0, L'$ has a smaller $\mathcal{J}$ proof. Using the inductive hypothesis and the $\Rightarrow$R rule provides the desired $\mathcal{F}$ proof.

Case: [?]**.** In this case, the list L is of the form $?(B^0), L'$ for some B in $\mathcal{L}_1$ and the sequent $\Sigma\colon \Psi, B^0; \Delta \Uparrow L'$ has a smaller $\mathcal{J}$ proof. By the inductive hypothesis, we have that the sequent $\Sigma\colon [\Psi]_-; \Delta^\perp \longrightarrow [\Delta]_+, L; B, [\Psi]_+$ is provable in $\mathcal{F}$ and by using ?R we can obtain a proof of the desired $\mathcal{F}$ proof.

Case: [!]**.** In this case, B is a formula of the form $?\,C$ and the sequent $\Sigma\colon \Psi; \Delta \Downarrow (C^0)^\perp$ is provable in $\mathcal{J}$. But the only way this can be proved is by the $[R\Downarrow]$, so the sequent $\Sigma\colon \Psi; (C^0)^\perp \Uparrow \cdot$ has a smaller $\mathcal{J}$ proof. By the inductive hypothesis, we have that the sequent $\Sigma\colon [\Psi]_-; C \longrightarrow \cdot; [\Psi]_+$ is provable in $\mathcal{F}$. Now using ?L, we have the desired $\mathcal{F}$ proof.

Case: $[R\Downarrow]$**.** Not possible.

Case: [&] **and** [$\forall$]**.** A simple use of induction and the &R and $\forall$R rules yield the desired $\mathcal{F}$ proofs.

Case: [$\otimes$]**.** In this case, B is either of the form $C \parr D$, $C \multimap D$, or $C \Rightarrow D$.

Subcase: B is $C \parr D$. In this case, the proof of $\Sigma\colon \Psi; \Delta \Downarrow (C^0)^\perp \otimes (D^0)^\perp$ has immediate subproofs of $\Sigma\colon \Psi; \Delta_1 \Downarrow (C^0)^\perp$ and $\Sigma\colon \Psi; \Delta_2 \Downarrow (D^0)^\perp$ where Δ is the multiset union of Δ_1 and Δ_2. The inductive hypothesis provides us with proofs of the sequents

$$\Sigma\colon [\Psi]_-; [\Delta_1]_- \xrightarrow{C} [\Delta_1]_+; [\Psi]_+ \text{ and } \Sigma\colon [\Psi]_-; [\Delta_2]_- \xrightarrow{D} [\Delta_2]_+; [\Psi]_+.$$

Using $\parr$L we obtain a proof of the sequent $\Sigma\colon [\Psi]_-; [\Delta]_- \xrightarrow{C \parr D} [\Delta_1]_+ + [\Delta_2]_+; [\Psi]_+$. It might not be the case that the list $[\Delta_1]_+ + [\Delta_2]_+$ is the same list as $[\Delta]_+$ (remember the ordering here was arbitrary and fixed), so we might need to use Lemma 2 to finally obtain a proof of $\Sigma\colon [\Psi]_-; [\Delta]_- \xrightarrow{C \parr D} [\Delta]_+; [\Psi]_+$.

Subcase: B **is** $C \multimap D$**.** In this case, the proof of $\Sigma\colon \Psi; \Delta \Downarrow C^0 \otimes (D^0)^\perp$ has immediate subproofs of $\Sigma\colon \Psi; \Delta_1 \Downarrow C^0$ and $\Sigma\colon \Psi; \Delta_2 \Downarrow (D^0)^\perp$ where Δ is the multiset union of Δ_1 and Δ_2. Since C^0 is either asynchronous or atomic, the only inference rule that yields the first of these sequents is $[R \Downarrow]$. Thus the sequent $\Sigma\colon \Psi; \Delta_1 \Uparrow C^0$ has a smaller proof. Using the inductive hypotheses, we conclude that sequents

$$\Sigma\colon [\Psi]_-; [\Delta_1]_- \longrightarrow [\Delta_1]_+, C; [\Psi]_+$$

and

$$\Sigma\colon [\Psi]_-; [\Delta_2]_- \xrightarrow{D} [\Delta_2]_+; [\Psi]_+$$

have $\mathcal{F}$ proofs. Using $\multimap$L we obtain a proof of the sequent

$$\Sigma\colon [\Psi]_-; [\Delta]_- \xrightarrow{C \multimap D} [\Delta_1]_+ + [\Delta_2]_+; [\Psi]_+.$$

As above, it might not be the case that the list $[\Delta_1]_+ + [\Delta_2]_+$ is the same list as $[\Delta]_+$, so we might need to use Lemma 2 to finally obtain a proof of $\Sigma\colon [\Psi]_-; [\Delta]_- \xrightarrow{C \multimap D} [\Delta]_+; [\Psi]_+$.

Subcase: B **is** $C \Rightarrow D$**.** In this case, the proof of $\Sigma\colon \Psi; \Delta \Downarrow\, !(C^0) \otimes (D^0)^\perp$ has immediate subproofs of $\Sigma\colon \Psi; \Delta_1 \Downarrow\, !(C^0)$ and $\Sigma\colon \Psi; \Delta_2 \Downarrow (D^0)^\perp$ where Δ is the multiset union of Δ_1 and Δ_2. The only inference rule that yields the first of these sequents is $[!]$, and this is only possible only if Δ_1 is empty and Δ_2 is equal to Δ. Thus the sequent $\Sigma\colon \Psi; \cdot \Uparrow C^0$ has a smaller proof. Using the inductive hypotheses, we conclude that $\Sigma\colon [\Psi]_-; \cdot \longrightarrow C; [\Psi]_+$ and $\Sigma\colon [\Psi]_-; [\Delta]_- \xrightarrow{D} [\Delta]_+; [\Psi]_+$. Using $\Rightarrow$L we obtain a proof of the sequent $\Sigma\colon [\Psi]_-; [\Delta]_- \xrightarrow{C \Rightarrow D} [\Delta]_+; [\Psi]_+$.

Case: $[\oplus_i]$**.** This case follows using the inference rule $\&\mathrm{L}_i$.

Case: $[\exists]$**.** This case follows using the inference rule $\forall$L.

Case: $[R \Uparrow]$**.** The list L must have the atomic formula A as its first element and L' as its tail and the sequent $\Sigma\colon \Psi; \Delta, A \Uparrow L'$ has a shorter proof. By the inductive hypothesis we know that $\Sigma\colon [\Psi]_-; [\Delta]_- \longrightarrow [\Delta, A]_+, L'; [\Psi]_+$ has an $\mathcal{F}$ proof and by using Lemma 2 we know that

$$\Sigma\colon [\Psi]_-; [\Delta]_- \longrightarrow [\Delta]_+, A, L'; [\Psi]_+$$

has an $\mathcal{F}$ proof.

Case: $[D_1]$ **and** $[D_2]$**.** The $[D_1]$ case follows immediately from the use of the inductive hypothesis and the *decide* inference rules. The $[D_2]$ case follows immediately from the use of the inductive hypothesis and either the *decide*! or *decide*? inference rules.

This proof actually describes an algorithm for converting a $\mathcal{J}$ proof into a corresponding $\mathcal{F}$ proof. The corresponding proofs are nearly the same — just different connectives and different bookkeeping is involved. ∎

The completeness of $\mathcal{F}$ immediately establishes Forum as an abstract logic programming language.

Notice that the form of the ?L rule is different from the other left introduction rules in that none of the sequents in its premise contain an arrow labeled with a formula. Thus, using this rule causes the "focus" of proof construction, which for left rules is directed by the subformulas of the formula labeling the sequent arrow, to be lost. If we were to replace that rule with the rule

$$\frac{\Sigma\colon \Psi; \cdot \xrightarrow{B} \cdot; \Upsilon}{\Sigma\colon \Psi; \cdot \xrightarrow{?\,B} \cdot; \Upsilon}\ ?\,\mathrm{L}'$$

that keeps the "focus", then the resulting proof system is not complete. In particular, the linear logic theorems $?\,a \multimap ?\,a$ and $?\,a \multimap ?((a \multimap b) \multimap b)$ would not be provable. Section 5 contains an occasion (Lemma 7) when using $?\,\mathrm{L}'$ instead of ?L is complete.

4 Operational reading of programs

We shall not discuss the many issues involved with building an interpreter or theorem prover for Forum. Certainly, work done on the implementations of languages such as λProlog, Lolli, and LO would all be applicable here. For now, we attempt to give the reader an understanding of what the high-level operational behavior of proof search is like using Forum specifications. Clearly, that semantics is an extension of these other logic programming languages, so we shall focus on those features that are novel to Forum and which are needed for the examples in the following sections.

First we comment on how the impermutabilities of some inference rules of linear logic are treated in Forum. In particular, an analogy exists between the embedding of all of linear logic into Forum and the embedding of classical logic into intuitionistic logic via a double negation 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 $B^{\perp\perp}$ on the right to be moved to the left as $B^{\perp}$, 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 transferred to one between a left rule above a right rule. For example, the bottom-up construction of a proof of the sequent $\longrightarrow a \otimes b, a^{\perp} \parr b^{\perp}$ must first introduce the $\parr$ prior to the $\otimes$: the context splitting required by $\otimes$ must be delayed until after the $\parr$ is introduced. This sequent, written using the connectives of Forum, is $\longrightarrow (a^\perp \parr b^\perp) \multimap \perp, a^\perp \parr b^\perp$. In this case, $\multimap$ and $\parr$ can be introduced in any order, giving rise to the sequent $a^\perp \parr b^\perp \longrightarrow a^\perp, b^\perp$. 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.

We shall use the term *backchaining* to refer to an application of either the *decide* or the *decide*! inference rule followed by a series of applcations of left-introduction rules. This notion of backchaining generalizes the usual notion found in the logic programming literature.

Sequents in linear logic and $\mathcal{F}$ contain multisets as (part of) their right-hand and left-hand sides. If we focus on the right-hand side, then the generalization of backchaining contained in the $\mathcal{F}$ proof system can be used to do multiset rewriting. As is well known, multiset rewriting is a natural setting for the specification of some aspects of concurrent computation. Given that multiset rewriting is only one aspect of the behavior of linear logic, such concurrent specifications are greatly enriched by the rest of higher-order linear logic. In particular, Forum allows for the integration of some concurrency primitives and various abstractions mechanisms in one declarative setting (see Section 6 for such an example specification).

To illustrate how multiset rewriting is specified in Forum, consider the clause

$$a \parr b \multimapinv c \parr d \parr e.$$

When presenting examples of Forum code we often use (as in this example) $\multimapinv$ and $\Leftarrow$ to be the converses of $\multimap$ and $\Rightarrow$ since they provide a more natural operational reading of clauses (similar to the use of `:-` in Prolog). Here, $\parr$ binds tighter than $\multimapinv$ and $\Leftarrow$. Consider the sequent $\Sigma\colon \Psi; \Delta \longrightarrow a, b, \Gamma; \Upsilon$ where the above clause is a member of Ψ. A proof for this sequent can then look like the following.

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

We can interpret this fragment of a proof as a reduction of the multiset a, b, Γ to the multiset c, d, e, Γ by backchaining on the clause displayed above.

Of course, a clause may have multiple, top-level implications. In this case, the surrounding context must be manipulated properly to prove the sub-goals that arise in backchaining. Consider a clause of the form

$$G_1 \multimap G_2 \Rightarrow G_3 \multimap G_4 \Rightarrow A_1 \parr A_2$$

labeling the sequent arrow in the sequent $\Sigma\colon \Psi; \Delta \longrightarrow A_1, A_2, \mathcal{A}; \Upsilon$. An attempt to prove this sequent would then lead to attempt to prove the four sequents

$$\Sigma\colon \Psi; \Delta_1 \longrightarrow G_1, \mathcal{A}_1; \Upsilon \quad \Sigma\colon \Psi; \cdot \longrightarrow G_2; \Upsilon$$

$$\Sigma\colon \Psi; \Delta_2 \longrightarrow G_3, \mathcal{A}_2; \Upsilon \quad \Sigma\colon \Psi; \cdot \longrightarrow G_4; \Upsilon$$

where Δ is the multiset union of Δ_1 and Δ_2, and $\mathcal{A}$ is $\mathcal{A}_1 + \mathcal{A}_2$. In other words, those subgoals immediately to the left of an $\Rightarrow$ are attempted with empty bounded contexts: the bounded contexts, here Δ and $\mathcal{A}$, are divided up and used in attempts to prove those goals immediately to the left of $\multimap$.

Although the innermost right-hand context of sequents in $\mathcal{F}$ is formally treated as a list, the order in the list is not "semantically" important: that list structure is only used to allow for a more constrained notion of proof search. It is easy to prove that a more general version of Lemma 2 holds.

Corollary 4 *Let Γ and Γ' be lists of formulas that are permutations of each other. If $\Sigma\colon \Psi; \Delta \longrightarrow \Gamma; \Upsilon$ has an $\mathcal{F}$ proof then so too does $\Sigma\colon \Psi; \Delta \longrightarrow \Gamma'; \Upsilon$.*

Proof This corollary can be proved by either referring to the soundness and completeness of $\mathcal{F}$ and the commutativity of $\parr$ or showing that all right-introduction rules in $\mathcal{F}$ permute over each other. ∎

A particularly difficult aspect of Forum to imagine implementing directly is backchaining over clauses with empty heads. For example, consider attempting to prove a sequent with right-hand side $\mathcal{A}$ and with the clause $\forall \bar{x}(G \multimap \bot)$ labeling the sequent arrow. This clause can be used in a backchaining step, regardless of $\mathcal{A}$'s structure, yielding the new right-hand side $\mathcal{A}, \theta G$, 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: backchaining using a clause with an empty head is always successful. See [21] 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 in the next section, 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 $\mathcal{F}$ proofs involving such clauses. Also, the encoding of various linear logic connectives into Forum involve clauses with empty heads. (Notice that clauses with empty heads are not allowed in LO.)

5 Specifying object-level sequent proofs

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 sequent calculus proof systems can be specified using the multiple conclusion aspect of Forum and show how properties of linear logic can be used to infer properties of the object-level proof systems. We shall use the terms *object-level logic* and *meta-level logic* to distinguish between the logic whose proof system is being specified and the logic of Forum.

Consider the well known, two-sided sequent proof systems for classical, intuitionistic, minimal, and linear logic. The distinction between these logics can be described, in part, by where the structural rules of thinning and contraction can be applied. In classical logic, these structural rules are allowed on both sides of the sequent arrow; in intuitionistic logic, only thinning is allowed on the right of the sequent arrow; in minimal logic, no structural rules are allowed on the right of the sequent arrow; and in linear logic, they are not allowed on either side of the arrow. This suggests the following representation of sequents in these four systems. Let *bool* be the type of object-level propositional formulas and let *left* and *right* be two meta-level predicates of type $bool \to o$. Sequents in these four logics can be specified as follows.

Linear: The sequent $B_1, \ldots, B_n \longrightarrow C_1, \ldots, C_m$ $(n, m \geq 0)$ can be represented by the meta-level formula

$$left\ B_1 \mathbin{⅋} \cdots \mathbin{⅋} left\ B_n \mathbin{⅋} right\ C_1 \mathbin{⅋} \cdots \mathbin{⅋} right\ C_m.$$

Minimal: The sequent $B_1, \ldots, B_n \longrightarrow C$ $(n \geq 0)$ can be represented by the meta-level formula

$$?\, left\ B_1 \mathbin{⅋} \cdots \mathbin{⅋} ?\, left\ B_n \mathbin{⅋} right\ C.$$

Intuitionistic: Intuitionistic logic contains the sequents of minimal logic and sequents with empty right-hand sides, *i.e.*, of the form $B_1, \ldots, B_n \longrightarrow$, where $n \geq 0$. These additional sequents can represented by the meta-level formula

$$?\, left\ B_1 \mathbin{⅋} \cdots \mathbin{⅋} ?\, left\ B_n.$$

Classical: The sequent $B_1, \ldots, B_n \longrightarrow C_1, \ldots, C_m$ $(n, m \geq 0)$ can be represented by the meta-level formula

$$?\, left\ B_1 \mathbin{⅋} \cdots \mathbin{⅋} ?\, left\ B_n \mathbin{⅋} ?\, right\ C_1 \mathbin{⅋} \cdots \mathbin{⅋} ?\, right\ C_m.$$

The *left* and *right* predicates are used to identify which object-level formulas appear on which side of the sequent arrow, and the ? modal is used to mark the formulas to which weakening and contraction can be applied.

We shall focus only on an object-logic that is minimal in this section. To denote first-order object-level formulas, we introduce the binary, infix symbols $\wedge$, $\vee$, and $\supset$ of type $bool \to bool \to bool$, and the symbols $\hat{\forall}$ and $\hat{\exists}$ of type

$$
\begin{array}{lrl}
(\supset R) & \mathit{right}\ (A \supset B) & \circ\!- (?(\mathit{left}\ A) \parr \mathit{right}\ B). \\
(\supset L) & ?(\mathit{left}\ (A \supset B)) & \circ\!- \mathit{right}\ A \circ\!- ?(\mathit{left}\ B). \\
(\wedge R) & \mathit{right}\ (A \wedge B) & \circ\!- \mathit{right}\ A \circ\!- \mathit{right}\ B. \\
(\wedge L_1) & ?(\mathit{left}\ (A \wedge B)) & \circ\!- ?(\mathit{left}\ A). \\
(\wedge L_2) & ?(\mathit{left}\ (A \wedge B)) & \circ\!- ?(\mathit{left}\ B). \\
(\hat{\forall} R) & \mathit{right}\ (\hat{\forall} B) & \circ\!- \forall x(\mathit{right}\ (Bx)). \\
(\hat{\forall} L) & ?(\mathit{left}\ (\hat{\forall} B)) & \circ\!- ?(\mathit{left}\ (Bx)). \\
(\mathit{Initial}) & \mathit{right}\ B \parr ?(\mathit{left}\ B). & \\
(\mathit{Cut}) & \bot & \circ\!- ?(\mathit{left}\ B) \circ\!- \mathit{right}\ B.
\end{array}
$$

Figure 3: Specification of the LM_1 sequent calculus.

$(i \to \mathit{bool}) \to \mathit{bool}$: the type i will be used to denote object-level individuals. Figure 3 is a specification of minimal logic provability using the above style of sequent encoding for just the connectives $\wedge$, $\supset$, and $\hat{\forall}$. (The connectives $\vee$ and $\hat{\exists}$ will be addressed later.) Expressions displayed as they are in Figure 3 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 3 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 3 are in the scope of ?. If occurrences of such modals in the heads of clauses were dropped, it would be possible to prove meta-level goals that do not correspond to any minimal logic sequent: such goals could contain *left*-atoms that are not prefixed with the ? modal.

We say that the object-level sequent $B_0, \ldots, B_n \longrightarrow B$ has an LM_1*-proof* if it has one in the sense of Gentzen [8] using the corresponding object-level inference rules $(\supset R)$, $(\supset L)$, $(\wedge R)$, $(\wedge L_1)$, $(\wedge L_2)$, $(\hat{\forall} R)$, $(\hat{\forall} L)$, $(\mathit{Initial})$, (Cut).

Let LM_1 be the set of clauses displayed in Figure 3 and let Σ_1 be the set of constants containing object-logical connectives $\hat{\forall}$, $\supset$, and $\wedge$ along with the two predicates *left* and *right* and any non-empty set of constants of type i (denoting members of the object-level domain of individuals). Notice that object-level quantification is treated by using a constant of second order, $\hat{\forall} : (i \to \mathit{bool}) \to \mathit{bool}$, in concert with meta-level quantification: in the two clauses $(\hat{\forall} R)$ and $(\hat{\forall} L)$, the type of B is $i \to \mathit{bool}$. This style representation of quantification is familiar from Church [6] and has been used to advantage in computer systems such as λProlog [7], Isabelle [29], and Elf [30]. This style of representing object-level syntax is often called *higher-order abstract syntax.*

To illustrate how these clauses specify the corresponding object-level inference rule, consider in more detail the first two clauses in Figure 3. Backchaining on the $\mathcal{F}$ sequent

$$\Sigma_1 : LM_1; \cdot \longrightarrow \mathit{right}(B_0 \supset C_0); \mathit{left}(B_1), \ldots, \mathit{left}\ B_n$$

using the ($\supset R$) clause in LM_1 (i.e., use *decide*!, $\forall$L twice, and $\multimap$L) yields the sequent

$$\Sigma_1 : LM_1; \cdot \longrightarrow ?(\mathit{left}\ B_0) \parr \mathit{right}\ C_0; \mathit{left}(B_1), \ldots, \mathit{left}\ B_n,$$

which in turns is provable if and only if the sequent

$$\Sigma_1 : LM_1; \cdot \longrightarrow \mathit{right}\ C_0; \mathit{left}\ B_0, \ldots, \mathit{left}\ B_n$$

is provable. Thus, proving the object-level sequent $B_1, \ldots, B_n \longrightarrow B_0 \supset C_0$ has been successfully reduced to proving the sequent $B_0, \ldots, B_n \longrightarrow C_0$. Now consider the sequent

$$\Sigma_1 : LM_1; \cdot \longrightarrow \mathit{right}(C); \mathit{left}(C_0 \supset B_0), \mathit{left}(B_1), \ldots, \mathit{left}\ B_n.$$

Using the *decide*! inference rule to select the ($\supset L$) clause, and using two instances of $\forall$L, we get the sequent whose right-hand and left-hand sides have not changed but where the sequent arrow is labeled with

$$?\,\mathit{left}\ B_0 \multimap \mathit{right}(C_0) \multimap ?\,\mathit{left}(C_0 \supset B_0).$$

Using $\multimap$L twice yields the following three sequents:

$$\begin{array}{r}
\Sigma_1 : LM_1; \cdot \longrightarrow \mathit{right}(C); \mathit{left}(C_0 \supset B_0), \mathit{left}\ B_0, \ldots, \mathit{left}\ B_n \\
\Sigma_1 : LM_1; \cdot \longrightarrow \mathit{right}(C_0); \mathit{left}(C_0 \supset B_0), \mathit{left}(B_1), \ldots, \mathit{left}\ B_n \\
\Sigma_1 : LM_1; \cdot \xrightarrow{?\,\mathit{left}(C_0 \supset B_0)} \cdot; \mathit{left}(C_0 \supset B_0), \mathit{left}(B_1), \ldots, \mathit{left}\ B_n
\end{array}$$

The last sequent is immediately provable using the ?L, *decide*, and *initial*? inference rules. Notice that the formula $\mathit{right}(C_0)$ could have moved to either the first or second sequent: if it had moved to the first sequent, no proof in $\mathcal{F}$ of that sequent is possible (provable $\mathcal{F}$ sequents using LM_1 contain at most one *right* formula in the right, inner-most context). Thus, we have succeeded in reducing the provability of the object-level sequent $C_0 \supset B_0, B_1, \ldots, B_n \longrightarrow C$ to the provability of the sequents

$$C_0 \supset B_0, B_1, \ldots, B_n \longrightarrow C_0 \quad \text{and} \quad C_0 \supset B_0, B_0, \ldots, B_n \longrightarrow C.$$

As we shall show in the proof of Proposition 5, these are the only possible reductions available using the clauses in LM_1.

In a similar fashion, we can trace the use of *decide*! on the ($\mathit{Initial}$) and (Cut) clauses to see these are equivalent to the inference rules

$$\overline{\Sigma_1 : LM_1; \cdot \longrightarrow \mathit{right}\ B; \mathit{left}\ B, \mathcal{L}}$$

and

$$\frac{\Sigma_1\colon LM_1;\cdot \longrightarrow \mathit{right}\ C;\mathcal{L} \quad \Sigma_1\colon LM_1;\cdot \longrightarrow \mathit{right}\ B;\mathit{left}\ C,\mathcal{L}}{\Sigma_1\colon LM_1;\cdot \longrightarrow \mathit{right}\ B;\mathcal{L}},$$

respectively, where $\mathcal{L}$ is a syntactic variable denoting a finite set of *left*-atoms.

In many ways, this style presentation of inference rules for LM_1 can be judged superior to the usual presentation using inference figures. For example, consider the following inference figures for $\wedge$R and $\supset$L taken from [8].

$$\frac{\Gamma \longrightarrow \Theta, A \quad \Gamma \longrightarrow \Theta, B}{\Gamma \longrightarrow \Theta, A \wedge B}\ \wedge R \qquad \frac{\Gamma \longrightarrow \Theta, A \quad B, \Delta \longrightarrow \Lambda}{A \supset B, \Gamma, \Delta \longrightarrow \Theta, \Lambda}\ \supset L$$

In these inference rules, the context surrounding the formulas being introduced must be explicitly mentioned and managed: in the $\wedge$R figure, the context is copied, while in the $\supset$L, the context is split to different branches (again, reading these inference figure bottom up). In the Forum specification, the context is manipulated implicitly via the use of the meta-level conjunctions: context copying is achieved using the additive conjunction & and context splitting is achieved using iterated $\circ\!-$ (i.e., using the multiplicative conjunction $\otimes$). Similarly, the structural rules of contraction and thinning can be captured together using the ? modal. Since the meta-logic captures so well many of the structural properties of the object-level proof system we can reason about properties of the object-level system using meta-level properties of Forum and linear logic. Of course, this approach to sequent calculus is also limited since Forum cannot naturally capture a number of features that are captured by conventional sequent figures: for example, the structural rule of exchange.

Notice that the well known problems with searching for proofs containing cut rules are transferred to the meta-level as problems of using a clause with $\perp$ for a head within the search for cut-free proofs (see Section 3).

Proposition 5 (Correctness of LM_1) *The sequent $B_1, \ldots, B_n \longrightarrow B_0$ ($n \geq 0$) has an LM_1-proof if and only if $\Sigma_1\colon LM_1;\cdot \longrightarrow \mathit{right}\ B_0;\mathit{left}\ B_1, \ldots, \mathit{left}\ B_n$ has a proof in $\mathcal{F}$.*

Proof For the forward direction, an LM_1-proof can be converted into a $\mathcal{F}$ proof of the corresponding meta-level formula by mapping the sequence of inference rules in the LM_1-proof into the sequence of clauses used in backchaining in the proof from $\mathcal{F}$. Additionally, right-introductions for $\invamp$, &, $\forall$, and ? will need to be inserted in a straightforward fashion.

For the reverse direction we need to read off the series of decide! inference rules in a proof from $\mathcal{F}$ of $\Sigma_1\colon LM_1;\cdot \longrightarrow \mathit{right}\ C_0;\mathit{left}\ C_1, \ldots, \mathit{left}\ C_n$ to construct an LM_1-proof of $C_1, \ldots, C_n \longrightarrow C_0$. Using the multiset rewriting notion of proof construction given in Section 4 makes this reading from proofs in $\mathcal{F}$ particular simple. However, that multiset rewriting paradigm was only described for clauses with atoms in the head: clauses in LM_1 contain formulas of the form ?(*left* B) in their head. We need to extend this notion of multiset rewriting to include such formulas. To do this, we first show the following two technical lemmas about the structure of proofs in $\mathcal{F}$ based on LM_1. The term LM_1*-derivative* is defined as follows:

1. If $D \in LM_1$ then D is an LM_1-derivative.
2. If $\forall_\tau x.D$ is an LM_1-derivative and t is a Σ_1-term of type τ, then $D[t/x]$ is an LM_1-derivative.
3. If $D_1 \parr D_2$ is an LM_1-derivative then D_1 and D_2 are LM_1-derivatives.
4. If $G \multimap D$ is an LM_1-derivative then D is an LM_1-derivative.
5. If $?\,D$ is an LM_1-derivative then D is an LM_1-derivative.
6. Nothing else is an LM_1-derivative.

Notice that all atoms of the form *right* A and *left* A are LM_1-derivatives and if $?\,D$ is an LM_1-derivative, then D is of the form *left* A.

Lemma 6 *If* $n \geq 0$ *and the sequent* $\Sigma_1 : LM_1; \cdot \longrightarrow \mathit{right}\ C_0; \mathit{left}\ C_1, \ldots, \mathit{left}\ C_n$ *has a proof in* $\mathcal{F}$*, then every sequent in that proof is of the form*

$$\Sigma : LM_1; \mathcal{L}_1 \longrightarrow \Gamma; \mathcal{L}_2 \quad \mathit{or} \quad \Sigma : LM_1; \mathcal{L}_1 \xrightarrow{D} \Gamma; \mathcal{L}_2$$

where Σ *is a signature containing* Σ*,* $\mathcal{L}_1$ *is a multiset of left-atoms containing at most one formula,* $\mathcal{L}_2$ *is a set of left-atoms containing* $\{\text{left } C_1, \ldots, \text{left } C_n\}$*,* D *is an* LM_1*-derivative, and* Γ *is a list of formulas of the form* $?(\mathit{left}\ A_1) \parr \mathit{right}\ A_2$*,* $\forall x(\mathit{right}\ (Bx))$*,* $\mathit{right}\ A$*,* $?(\mathit{left}\ A)$*,* $\mathit{left}\ A$*, and* $\perp$*. Here,* A*,* A_1*, and* A_2 *are* Σ*-terms of type* bool*, and* B *is a* Σ*-term of type* $i \rightarrow \mathit{bool}$*.*

Proof This lemma can be proved by induction on the distance of sequents from the root sequent. Clearly the root itself satisfies the condition on sequents. Next, we need to show that if the conclusion of an inference rule satisfies these conditions, the premises of that inference rule also satisfies these conditions. The only inference rules that can have such a sequents as conclusions are &R, $\perp$R, $\forall$R, ?R, *decide* ?, *decide*!, *decide*, *initial*, *initial* ? $\multimap$L, $\forall$L, and ?L. An examination of these rules confirms the inductive step of the proof. ∎

Lemma 7 *Let* $\mathcal{L}$ *be a set of left-atoms,* D *be an* LM_1*-derivative,* $\mathcal{A}$ *be a list of* Σ_1*-atomic formulas, and* A *be a* Σ_1*-term of type* bool*. If either*

$$\Sigma_1 : LM; \mathit{left}\ A \longrightarrow \mathcal{A}; \mathcal{L} \quad \mathit{or} \quad \Sigma_1 : LM; \mathit{left}\ A \xrightarrow{D} \mathcal{A}; \mathcal{L}$$

has a proof in $\mathcal{F}$ *then* $\mathcal{A}$ *is the one element list containing* left A *or* $\mathcal{A}$ *is empty and* left $A \in \mathcal{L}$*.*

Proof Assume not. Then one of these sequents is provable but the conditions on $\mathcal{A}$ and $\mathcal{L}$ are not satisfied. Pick the proof in $\mathcal{F}$ that has the smallest height and consider the various inference rules that can terminate that proof. Clearly, that proof does not end in a right-introduction rule or an initial rule. If the last rule was any of the other rules (the *decide* rules or the left-introduction rules), then one of the premises would contain a sequent that similarly does not satisfy the conditions in the lemma. This is then a contradiction since that sequent has a proof of smaller height. ∎

The implication of this lemma is that the inference rule $?\mathrm{L}'$ can be used in place of ?L (see the end of Section 3) when using the LM_1 theory. More specifically, if the sequent $\Sigma_1\colon LM_1;\cdot \xrightarrow{?\,\mathit{left}\ B} \cdot;\mathcal{L}$ is provable then $\mathit{left}\ B \in \mathcal{L}$. Given this result, the multiset rewriting paradigm can be extended to those clause in LM_1.

The correctness proof for LM_1 is now easy to finish: given a proof in $\mathcal{F}$ of

$$\Sigma_1\colon LM_1;\cdot \longrightarrow \mathit{right}\ C_0; \mathit{left}\ C_1,\ldots,\mathit{left}\ C_n,$$

just read off of it the series of decide! rules that are used to select members of the LM_1 set of clauses. Applying the sequent inference rules that correspond to those clauses yields an LM_1 proof of $C_1,\ldots,C_n \longrightarrow C_0$. ∎

So far we have only discussed the operational interpretation of the specification in Figure 3. 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 imply the equivalences $(\mathit{right}\ B)^\perp \equiv ?(\mathit{left}\ B)$ and $(\mathit{right}\ B) \equiv !(\mathit{right}\ B)$. That is, we have the (not too surprising) fact that *left* and *right* are related by a meta-level negation, and that this is guaranteed by reference only to the specifications for the initial and cut rules. Given these equivalences, it is possible to eliminate references to *left* in the LM_1 specification. The result would be a specification quite similar to one for specifying a natural deduction proof system for minimal logic. To this end, consider the specification of the NM_1 natural deduction proof system given in Figure 4. The specification there is similar to those given using intuitionistic meta-logics [7, 29] and dependent typed calculi [3, 13].

Proposition 8 (Correctness of NM_1) *The formula B_0 has an NM_1 proof from the assumptions $B_1,\ldots,B_n$ $(n \geq 0)$ if and only if*

$$\Sigma_1\colon NM_1, \mathit{right}\ B_1,\ldots,\mathit{right}\ B_n;\cdot \longrightarrow \mathit{right}\ B_0;\cdot$$

has a proof in $\mathcal{F}$.

Proof The correctness proof for natural deduction based on intuitionistic logic and type theories that can be found in [7, 13, 29] can be used here as well. The only difference is that in Figure 4, certain occurrences of $\Leftarrow$ are replaced with occurrences of $\circ\!-$. This replacement can be justified using Proposition 6 of

$$
\begin{array}{lr@{\;}l}
(\supset I) & \mathit{right}\ (A \supset B) & \circ\!\!- (\mathit{right}\ A \Rightarrow \mathit{right}\ B). \\
(\supset E) & \mathit{right}\ B & \circ\!\!- \mathit{right}\ A \circ\!\!- \mathit{right}\ (A \supset B). \\
(\wedge I) & \mathit{right}\ (A \wedge B) & \circ\!\!- \mathit{right}\ A \circ\!\!- \mathit{right}\ B. \\
(\wedge E_1) & \mathit{right}\ A & \circ\!\!- \mathit{right}\ (A \wedge B). \\
(\wedge E_2) & \mathit{right}\ B & \circ\!\!- \mathit{right}\ (A \wedge B). \\
(\hat{\forall} I) & \mathit{right}\ (\hat{\forall} B) & \circ\!\!- \forall x(\mathit{right}\ (Bx)). \\
(\hat{\forall} E) & \mathit{right}\ (Bx) & \circ\!\!- \mathit{right}\ (\hat{\forall} B).
\end{array}
$$

Figure 4: Specification of the NM_1 natural deduction calculus.

[15] in which it is shown that when translating an intuitionistic theory to linear logic, positive occurrences of intuitionistic implications can be translated using by $\multimap$ while negative occurrences can be translated using $\Rightarrow$. It follows that these two presentations of NM_1 prove the same sequents of the form displayed in this Proposition. ∎

We can now supply a meta-logical proof that NM_1 and LM_1 prove the same object-level theorems. The following two lemmas supply the necessary implications.

Lemma 9 $\vdash LM_1 \equiv [(\otimes NM_1) \otimes \mathit{Initial} \otimes \mathit{Cut}]$.

Proof As we remarked before the formulas *Initial* and *Cut* in LM_1 entail the equivalences $(\mathit{right}\ B)^\perp \equiv ?(\mathit{left}\ B)$ and $(\mathit{right}\ B) \equiv\ !(\mathit{right}\ B)$. If we apply these two equivalences along with the linear logic equivalences

$$p^\perp \circ\!\!- q^\perp \equiv q \circ\!\!- p \qquad (!\,p)^\perp \parr q \equiv p \Rightarrow q \qquad (p^\perp \mathbin{\&} q^\perp)^\perp \equiv p \oplus q$$

to the first seven clauses in Figure 3, we get the seven clauses in Figure 4. (The last two clauses of LM_1 become linear logic theorems.) Clearly, $LM_1 \vdash (\otimes NM_1)$. The proof of the converse entailment follows by simply reverse the steps taking above: we can work backwards from NM_1 to LM_1 by equivalences. ∎

Before we establish that LM_1 and NM_1 prove the same object-level formulas (Theorem 13), we need a couple of technical lemmas.

Lemma 10 *If* $\Sigma_1\!: NM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ *has a proof in* $\mathcal{F}$*, then* $\Sigma_1\!: LM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ *has a proof in* $\mathcal{F}$*.*

Proof This follows directly from Lemma 9, cut-elimination for linear logic, and the soundness and completeness results for $\mathcal{F}$. ∎

Lemma 11 *If* $\Sigma_1\!: NM_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \cdot$ *has a proof in* $\mathcal{F}$*, then* $\Sigma_1\!: NM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ *has a proof in* $\mathcal{F}$*.*

Proof Let Ξ be a proof in $\mathcal{F}$ of $\Sigma_1\!: NM_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \cdot$. We show we can always eliminate occurrences of *decide*! rules in Ξ that select the

Cut clause. Once they have all been eliminated, the *Initial* clause is also not selected.

Consider the sequent that occurs the highest in Ξ that is also the conclusion of a *decide*! rule that select *Cut*. As we noted earlier, that sequent is of the form

$$\Sigma\colon NM_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \mathcal{L}$$

and it has above it subproofs Ξ_1 and Ξ_2 of the sequents

$$\Sigma\colon NM_1; \cdot \longrightarrow \mathit{right}\ C; \mathcal{L} \quad \text{and} \quad \Sigma\colon NM_1; \cdot \longrightarrow \mathit{right}\ B; \mathit{left}\ C, \mathcal{L},$$

respectively. We can now transform Ξ_2 into Ξ_2' as follows: first remove *left* C from the right-most context of all of its sequents and for every occurrence of the initial rule in Ξ_2 of the form

$$\overline{\Sigma_1\colon NM_1; \cdot \longrightarrow \mathit{right}\ C; \mathit{left}\ C, \mathcal{L}},$$

replace that subproof in Ξ_2 with Ξ_1. The resulting Ξ_2' is a proof of

$$\Sigma\colon NM_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \mathcal{L}$$

and, since Ξ_1 and Ξ_2 do not contain occurrences of *decide*! that selected *Cut*, neither does Ξ_2'. In this way, we have reduced the number of backchainings using *Cut* in Ξ by one.

Continuing in this fashion, we can eliminate all such uses of the *Cut* clause in proving the sequent $\Sigma_1\colon NM_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \cdot$. Since backchaining on *Cut* introduces *left*-atoms and backchaining on *Initial* eliminates such atoms (reading from bottom-up), if there there are no such occurrences of *Cut*, then there are no such occurrences of *Initial*. Hence, we have described a proof in $\mathcal{F}$ of $\Sigma_1\colon NM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$. ∎

Lemma 12 *If $\Sigma_1\colon LM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ has a proof in $\mathcal{F}$, then $\Sigma_1\colon NM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ has a proof in $\mathcal{F}$.*

Proof Assume $\Sigma_1\colon LM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ has a proof in $\mathcal{F}$. Using Lemma 9, cut-elimination for linear logic, and the soundness and completeness results for $\mathcal{F}$, the sequent

$$\Sigma_1\colon NM_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \cdot$$

has a proof in $\mathcal{F}$. Now using Lemma 11, we have that $\Sigma_1\colon NM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ has a proof in $\mathcal{F}$. ∎

The following theorem follows from results of Gentzen [8]. We supply a new proof here using linear logic as a meta-theory.

Theorem 13 *The sequent $\longrightarrow B$ has an LM_1 proof if and only if B has an NM_1-proof (from no assumptions).*

$$
\begin{array}{lrl}
(\vee R_1) & \mathit{right}\ (A \vee B) & \circ\!\!- \mathit{right}\ A. \\
(\vee R_2) & \mathit{right}\ (A \vee B) & \circ\!\!- \mathit{right}\ B. \\
(\vee L) & ?(\mathit{left}\ (A \vee B)) & \circ\!\!- ?(\mathit{left}\ A)\ \&\ ?(\mathit{left}\ B). \\
(\hat{\exists} R) & \mathit{right}\ (\hat{\exists} B) & \circ\!\!- \mathit{right}\ (Bx). \\
(\hat{\exists} L) & ?(\mathit{left}\ (\hat{\exists} B)) & \circ\!\!- \forall x(?(\mathit{left}\ (Bx))).
\end{array}
$$

Figure 5: Sequent rules for disjunction and existential quantification.

$$
\begin{array}{lrl}
(\vee I_1)' & \mathit{right}\ (A \vee B) & \circ\!\!- \mathit{right}\ A. \\
(\vee I_2)' & \mathit{right}\ (A \vee B) & \circ\!\!- \mathit{right}\ B. \\
(\vee E)' & \bot & \circ\!\!- \mathit{right}\ (A \vee B) \\
 & & \circ\!\!- (\mathit{right}\ A \Rightarrow \bot)\ \&\ (\mathit{right}\ B \Rightarrow \bot). \\
(\hat{\exists} I)' & \mathit{right}\ (\hat{\exists} B) & \circ\!\!- \mathit{right}\ (Bx). \\
(\hat{\exists} E)' & \bot & \circ\!\!- \mathit{right}\ (\hat{\exists} B) \\
 & & \circ\!\!- \forall x(\mathit{right}\ (Bx) \Rightarrow \bot).
\end{array}
$$

Figure 6: Equivalent forms of the clauses in Figure 5.

$$
\begin{array}{lrl}
(\vee I_1) & \mathit{right}\ (A \vee B) & \circ\!\!- \mathit{right}\ A. \\
(\vee I_2) & \mathit{right}\ (A \vee B) & \circ\!\!- \mathit{right}\ B. \\
(\vee E) & \mathit{right}\ E & \circ\!\!- \mathit{right}\ (A \vee B) \\
 & & \circ\!\!- (\mathit{right}\ A \Rightarrow \mathit{right}\ E) \\
 & & \circ\!\!- (\mathit{right}\ B \Rightarrow \mathit{right}\ E). \\
(\hat{\exists} I) & \mathit{right}\ (\hat{\exists} B) & \circ\!\!- \mathit{right}\ (Bx). \\
(\hat{\exists} E) & \mathit{right}\ E & \circ\!\!- \mathit{right}\ (\hat{\exists} B) \\
 & & \circ\!\!- \forall x(\mathit{right}\ (Bx) \Rightarrow \mathit{right}\ E).
\end{array}
$$

Figure 7: Natural deduction rules for disjunction and existential quantification.

Proof This theorem follows immediately from Propositions 5 and 8 and Lemmas 10 and 12. ∎

Now consider adding to our object-logic disjunction and existential quantification. Let Σ_2 be Σ_1 with the constants $\vee$ and $\hat{\exists}$ added. Let LM_2 be the sequent system that results from adding the five clauses in Figure 5 to LM_1. Note the use of & in the specification of $(\vee L)$: this conjunction is needed since the right-hand of the object-level sequent is copied in this inference rule.

Applying the equivalences $(\mathit{right}\ B)^\perp \equiv\ ?(\mathit{left}\ B)$ and $(\mathit{right}\ B) \equiv\ !(\mathit{right}\ B)$ to the clauses displayed in Figure 5, we get the formulas in Figure 6. The clauses for $(\vee E)'$ and $(\hat{\exists} E)'$ could also be written more directly as the linear logic formulas

$$
\begin{array}{r}
(\mathit{right}\ A) \oplus (\mathit{right}\ B) \circ\!\!- \mathit{right}\ (A \vee B). \\
\exists x(\mathit{right}\ (Bx)) \circ\!\!- \mathit{right}\ (\hat{\exists} B).
\end{array}
$$

(using the equivalence $(\mathit{right}\ B) \equiv\ !(\mathit{right}\ B)$).

Figure 7 contains the usual introduction and elimination rules for natural deduction for $\vee$ and $\hat{\exists}$. The only difference between the clauses in that Figure and those in Figure 6 is that the natural deduction rules for disjunction and existential quantification use the atom *right* E instead of $\perp$ in the elimination rules for $\vee$ and $\hat{\exists}$. While this difference does not allow us to directly generalize Lemma 9 to include these two connectives, it is possible to show that the clauses in Figure 6 or in Figure 7 prove the same object-level theorems. For example, let NM_2' be the set of clauses formed by adding the clauses in Figure 6 to NM_1 and consider using *decide*! rule with the $(\vee E)'$ clause to prove the $\mathcal{F}$ sequent

$$\Sigma_2 \colon NM_2', \mathcal{R}; \cdot \longrightarrow \mathit{right}\ E; \cdot.$$

This would lead to subproofs of the form

$$\Sigma_2 \colon NM_2', \mathit{right}\ A, \mathcal{R}; \cdot \longrightarrow \mathit{right}\ E; \cdot \quad \text{and} \quad \Sigma_2 \colon NM_2', \mathit{right}\ A, \mathcal{R}; \cdot \longrightarrow \mathit{right}\ E; \cdot.$$

Here, we assume that $\mathcal{R}$ is a set of *right*-atoms containing *right* $(A \vee B)$. This is, of course, the same reduction in proof search if $(\vee E)$ (from Figure 7) was used instead. A similar observation holds for using either $(\exists E)'$ or $(\exists E)$. Given these observations, we could prove the generalization of Theorem 13 using LM_2 and NM_2. Notice that the specifications of NM_1 and NM_2 avoid using either ⅋ or $\perp$, and as a result, they can be modeled using on intuitionistic linear logic, in fact, a simple subset of that like Lolli [15].

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 [7], Isabelle [29], and Elf [30]. 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, [7]). 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 examples that Forum allows for both the natural specification of sequent calculus and the possibility of handling some substructural object-logics.

6 Operational Semantics Examples

Evaluation of pure functional programs has been successfully specified in intuitionistic meta-logics [11] and type theories [4, 30] using structured operational semantics and natural semantics. These specification systems are less successful at providing natural specifications of languages that incorporate references and concurrency. In this section, we consider how evaluation incorporating references can be specified in Forum; specification of concurrency primitives will be addressed in the following section.

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

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

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 inference rule. 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, [11]).

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

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 [33], this ordering can be made explicit using the following two clauses, this time using the predicate $eval$ at type $tm \to tm \to o \to o$.

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

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 non-functional features.

Consider adding to this specification a single global counter that can be read and incremented. To specify such a counter we place the integers into type `tm`, add several simple functions over the integers, and introduce the two symbols get and inc of type `tm`. The intended meaning of these two 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 $\forall k(\mathit{eval}\ i\ i\ k \circ\!-\ k)$ is part of the evaluator's specification.

Figure 8 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

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

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

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

Figure 8: Three specifications of a global counter.

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 [20]). 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 [15]). 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 8, all of which are not primitive connectives of Forum, is for convenience in displaying these abstract data types. The equivalence

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

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.

Proposition 14 *The three entailments $E_1 \vdash E_2$, $E_2 \vdash E_3$, and $E_3 \vdash E_1$ are provable in linear logic.*

Proof 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 $E_2 \vdash \mathit{eval}\ M\ V\ \top$.

It is possible to specify a more general notion of reference from which a counter such as that described above can be built. Consider the specification in Figure 9. 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

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

This expression represents the program that first evaluates n; then allocates a new, scoped reference cell that 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

$$(\mathit{app}\ (\mathit{abs}\ \lambda x(\mathit{app}\ m\ x))\ n).$$

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 9 plus the two formulas displayed above for the evaluation of *app* and *abs*. 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. A store is essentially 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

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

$$\begin{aligned}
\mathit{eval}\ (\mathit{set}\ L\ N)\ V\ K &\multimap\!\!\!- \ \mathit{eval}\ N\ V\ (\mathit{assign}\ L\ V\ K).\\
\mathit{eval}\ (\mathit{new}\ R\ E)\ V\ K &\multimap\!\!\!- \ \mathit{eval}\ E\ U\ (\forall h(\mathit{ref}\ h\ U \mathbin{⅋} \mathit{eval}\ (R\ h)\ V\ K)).\\
\mathit{eval}\ (\mathit{read}\ L)\ V\ K \mathbin{⅋} \mathit{ref}\ L\ V &\multimap\!\!\!- \ K \mathbin{⅋} \mathit{ref}\ L\ V.\\
\mathit{assign}\ L\ V\ K \mathbin{⅋} \mathit{ref}\ L\ U &\multimap\!\!\!- \ K \mathbin{⅋} \mathit{ref}\ L\ V.
\end{aligned}$$

Figure 9: Specification of references.

$\{h_1, \ldots, h_n\}$. 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. Given the specification of the evaluation of *new* in Figure 9, new locations are modeled at the meta-level using the eigen-variables that are introduced by the $\forall R$ inference rule of $\mathcal{F}$.

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, a new store S', and a garbaged store G if the formula

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

is provable from the clauses in *Ev* and the signature extended with the domain of S. An immediate consequence of this formula is that the formula $\mathit{eval}\ M\ V\ \top \mathbin{⅋} S$ 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 in the sense that 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.

A consequence of the formulas in *Ev* is the formula

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

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

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

in other words, this expression can be evaluated in any store without changing it. Because of their quantification, garbaged stores are inaccessible: 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 [27].

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

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

From these formulas and those in Ev, we can infer the following formulas.

$$\begin{array}{ll} \forall W \forall k[\mathit{eval}\ (u\ v)\ W\ k \parr S_2 \parr G_1 \parr G_2 \parr \forall h(\mathit{ref}\ h\ v) & \multimap \mathit{eval}\ F_1\ W\ k \parr S_0] \\ \forall W \forall k[\mathit{eval}\ (u\ v)\ W\ k \parr S_2 \parr G_1 \parr G_2 & \multimap \mathit{eval}\ F_2\ W\ k \parr S_0] \end{array}$$

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. The only difference in their evaluations is that F_1 leaves behind an additional garbaged store. Since the continuation k is universally quantified in these formulas, F_1 and F_2 have these behaviors in any evaluation context.

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.

7 Specification of Concurrency primitives

Concurrency primitives similar to those found in the π-calculus were shown in [23] to be expressible naturally in a fragment of linear logic subsumed by Forum. Below we show how concurrency primitives, inspired by those found in Concurrent ML (CML) [31] can be specified in Forum. Consider the specification in Figure 10. The first eight clauses specify the straightforward evaluation rules for the corresponding eight data constructors. The next three clauses defined the meaning of the three special forms *sync*, *spawn*, and *newchan*. The remaining clauses specify the *event* predicate.

This specification allows for multiple threads of evaluation. Evaluation of the *spawn* function initiates a new evaluation thread. The *newchan* function causes the meta-logic to pick a new eigen-variable (via the $\forall c$ quantification) and then to assume that that eigen-variable is a value (via the assumption $\forall I(\mathit{eval}\ c\ c\ I \multimapinv I)$): such a new value can be used to designate new channels for use in synchronization. The name-restriction operator of the π-calculus can be modeled using universal quantification in a similar fashion [23].

The *sync* primitive allows for synchronization between threads: its use causes an "evaluation thread" to become an "event thread." The behaviors of event threads are described by the remaining clauses in Figure 10. The primitive events are *transmit* and *receive* and they represent two halves of a synchronization between two event threads. Notice that the clause describing their meaning is the only clause in Figure 10 that has a head with more than one atom. The non-primitive events *choose*, *wrap*, *guard*, and *poll* are reduced to other calls to *event* and *eval*. The *choice* event is implemented as a local, non-deterministic choice. (Specifying global choice, as in CCS [26], would be much more involved.) The *wrap* and *guard* events chain together evaluation and synchronization but in direct orders.

The only use of & and $\top$ in any of our evaluators is in the specification of polling: in an attempt to synchronize with (*poll* E) (with the continuation K) the goal (*event* E U $\top$) & K is attempted (for some unimportant term U). Thus, a copy of the current evaluation threads is made and (*event* E U $\top$) is attempted in one of these copies. This atom is provable if and only if there is a complementary event for E in the current environment, in which case, the continuation $\top$ brings us to a quick completion and the continuation K is attempted in the original and unspoiled context of threads. If such a complementary event is not present, then the other clause for computing a polling event can be used, in which case, the result of the poll is *none*, which signals such a failure. The semantics of polling, unfortunately, is not exactly as intended in CML since it is possible to have a polling event return *none* even if the event being tested could be synchronized. This analysis of polling is similar to the analysis of testing in process calculus as described in [23]. As is discussed there, this problem with polling can be addressed if the meta-logic allows certain forms of negation-as-failure.

About this specification, we shall not prove anything formal, although it should be clear that the approach to reasoning about object-level programs using meta-level resolution, as in the last section, should be applicable here as well.

In the [5], Chirimar presents a specification of a programming language motivated by Standard ML [28]. In particular, a specification for the call-by-value λ-calculus is provided, and then modularly extended with the specifications of references, exceptions, and continuations: each of these features is specified without complicating the specifications of other the features.

8 Conclusions

We have given a presentation of linear logic that modularly extends the proof theory of several known logic programming languages. The resulting specification language, named Forum, provides the abstractions and higher-order judgments available in intuitionistic-based meta-logics as well as primitives for

$$
\begin{aligned}
\text{eval}, \text{event} &: \text{tm} \to \text{tm} \to o \to o.\\
\mathit{none} &: \text{tm}.\\
\mathit{guard}, \mathit{poll}, \mathit{receive}, \mathit{some}, \mathit{sync} &: \text{tm} \to \text{tm}.\\
\mathit{choose}, \mathit{transmit}, \mathit{wrap} &: \text{tm} \to \text{tm} \to \text{tm}.\\
\mathit{spawn}, \mathit{newchan} &: (\text{tm} \to \text{tm}) \to \text{tm}.
\end{aligned}
$$

$$
\begin{aligned}
\text{eval}\ \mathit{none}\ \mathit{none}\ K &\multimap K.\\
\text{eval}\ (\mathit{guard}\ E)\ (\mathit{guard}\ V)\ K &\multimap \text{eval}\ E\ V\ K.\\
\text{eval}\ (\mathit{poll}\ E)\ (\mathit{poll}\ V)\ K &\multimap \text{eval}\ E\ V\ K.\\
\text{eval}\ (\mathit{receive}\ E)\ (\mathit{receive}\ V)\ K &\multimap \text{eval}\ E\ V\ K.\\
\text{eval}\ (\mathit{some}\ E)\ (\mathit{some}\ V)\ K &\multimap \text{eval}\ E\ V\ K.\\
\text{eval}\ (\mathit{choose}\ E\ F)\ (\mathit{choose}\ U\ V)\ K &\multimap \text{eval}\ E\ U\ (\text{eval}\ F\ V\ K).\\
\text{eval}\ (\mathit{transmit}\ E\ F)\ (\mathit{transmit}\ U\ V)\ K &\multimap \text{eval}\ E\ U\ (\text{eval}\ F\ V\ K).\\
\text{eval}\ (\mathit{wrap}\ E\ F)\ (\mathit{wrap}\ U\ V)\ K &\multimap \text{eval}\ E\ U\ (\text{eval}\ F\ V\ K).
\end{aligned}
$$

$$
\begin{aligned}
\text{eval}\ (\mathit{sync}\ E)\ V\ K &\multimap \mathit{eval}\ E\ U\ (\text{event}\ U\ V\ K).\\
\text{eval}\ (\mathit{spawn}\ R)\ \mathit{unit}\ K &\multimap (\text{eval}\ (R\ \mathit{unit})\ \mathit{unit}\ \bot) \parr K.\\
\text{eval}\ (\mathit{newchan}\ R)\ V\ K &\multimap \forall c(\forall I(\text{eval}\ c\ c\ I \multimap I) \Rightarrow \text{eval}\ (R\ c)\ V\ K).
\end{aligned}
$$

$$
\text{event}\ (\mathit{receive}\ C)\ V\ K \parr \text{event}\ (\mathit{transmit}\ C\ V)\ \mathit{unit}\ L \multimap K \parr L.
$$

$$
\begin{aligned}
\text{event}\ (\mathit{choose}\ E\ F)\ V\ K &\multimap \text{event}\ E\ V\ K.\\
\text{event}\ (\mathit{choose}\ E\ F)\ V\ K &\multimap \text{event}\ F\ V\ K.\\
\text{event}\ (\mathit{wrap}\ E\ F)\ V\ K &\multimap \text{event}\ E\ U\ (\text{eval}\ (\mathit{app}\ F\ U)\ V\ K).\\
\text{event}\ (\mathit{guard}\ F)\ V\ K &\multimap \text{eval}\ (\text{app}\ F\ \mathit{unit})\ U\ (\text{event}\ U\ V\ K).\\
\text{event}\ (\mathit{poll}\ E)\ (\mathit{some}\ E)\ K &\multimap (\text{event}\ E\ U\ \top)\ \&\ K.\\
\text{event}\ (\mathit{poll}\ E)\ \mathit{none}\ K &\multimap K.
\end{aligned}
$$

Figure 10: Specifications of some primitives similar to those found in Concurrent ML.

synchronization and communications. We have illustrated the possible uses of Forum by providing example specifications of object-level sequent systems and of the operational semantics of programming languages. Since the resulting specifications are natural and simple, properties of the meta-logic can be meaningful employed to prove properties about the specified object-languages.

Forum appears to be useful for other kinds of semantic specifications as well. In [5], Chirimar specified the operation semantics of a prototypical RISC machine: he specified its sequential and pipelined operational semantics, along with optimizations such as call-forwarding and early branch resolution. He also proved that sequential and pipelined specification were equivalent, and he addressed the problem of code equivalence and analyzed the problem of code rescheduling. His proofs and analyses made use of familiar aspects of proof theory and linear logic.

Acknowledgments. This paper is an extended version of a paper that appeared as [24] and which was also presented at the 1994 Joint Meeting of ALP and PLILP, Madrid, September 1994. I benefited from discussions with Jawahar Chirimar and Bob Harper. Thanks also to Bow-Yaw Wang for his help in proofreading this paper. The author has been funded in part by ONR N00014-93-1-1324, NSF CCR-92-09224, NSF CCR-94-00907, and ARO DAAH04-95-1-0092. Papers by Miller listed in the bibliography are available via anonymous ftp from `ftp.cis.upenn.edu` in `pub/papers/miller` or using WWW at `http://www.cis.upenn.edu/~dale`.

References

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

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

[3] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[4] R. Burstall and Furio Honsell. A natural deduction treatment of operational semantics. In *Proceedings of the 8th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume LNCS, Vol. 338, pages 250–269. Springer-Verlag, 1988.

[5] Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, February 1995. Available as `ftp://ftp.cis.upenn.edu/pub/papers/chirimar/phd.ps.gz`.

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

[7] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

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

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

[10] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[11] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[12] James Harland and David Pym. On goal-directed provability in classical logic. Technical Report 92/16, Dept of Comp Sci, Uni. of Melbourne, 1992.

[13] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

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

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

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

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

[18] Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *Proceedings of OOPSLA'94*, 1994. To appear.

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

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

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

[22] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

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

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

[25] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

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

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

[28] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[29] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, September 1989.

[30] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.

[31] John H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

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

[33] Paul Tarau. Program transformations and WAM-support for the compilation of definite metaprograms. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in LNAI, pages 462–473. Springer-Verlag, 1992.