Logic Programming and Meta-Logic

Dale Miller

Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104–6389 USA
dale@saul.cis.upenn.edu

Abstract. The theory of cut-free sequent proofs has been used to motivate and justify the design of a number of logic programming languages. Two such languages, lambda Prolog and its linear logic refinement, Lolli, 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) provides some primitives for concurrency but lacks abstraction mechanisms. A logic programming presentation of all of higher-order linear logic, named Forum, modularly extends these other languages and also allows for abstractions and concurrency in specifications. To illustrate the expressive strengths of Forum, we specify in it a sequent calculus proof system and the operational semantics of a programming language that incorporates side-effects.
Keywords. Logic programming, linear logic, higher-order abstract syntax, meta-logic, lambda Prolog, Forum.

1 Logic as a specification language

This section contains some non-technical observations about the roles that logic can play in the specification of computational systems. In the following sections of this chapter, a more technical presentation of a specification language based on higher-order linear logic is presented.

1.1 Two approaches to specifications

In the specification of computational systems, logics are generally used in one of two approaches. In one approach, computations are mathematical structures, containing such items as nodes, transitions, and state, and logic is used in an external sense to make statements *about* those structures. That is, computations are used as models for logical expressions. Intensional operators, such as the modals of temporal and dynamic logics or the triples of Hoare logic, are often employed to express propositions about the change in state. For example, next-time modal operators are used to describe the possible evolution of state; expressions in the Hennessey-Milner are evaluated against the transitions made by a process; and Hoare logic uses formulas to express pre- and post-conditions on a computation's state. We shall refer to this approach to using logic as *computation-as-model.* In such approaches, the fact that some identifier x has value 5 is represented as, say a pair $\langle x, 5\rangle$, within some larger mathematical structure, and logic is used to express propositions *about* such pairs: for example, $x > 3 \wedge x < 10$.

A second approach uses logical deduction to model computation. In this approach the fact that the identifier x has value 5 can be encoded as the proposition "x has value 5." Changes in state can then be modeled by changes in propositions within a derivation. Of course, changing state may require that a proposition no longer holds while a proposition that did not hold (such as "x has value 6") may hold in a new state. It is a common observation that such changes are naturally supported by linear logic and that deduction (in particular, backchaining in the sense of logic programming) can encode the evolution of a computation. As a result, it is possible to see the state of a computation as a logical formula and transitions between states as steps in the construction of a proof. We shall refer to this approach to using logic as *computation-as-deduction.*

There are many ways to contrast these two approaches to specification using logic. For example, consider their different approaches to the "frame problem." Assume that we are given a computation state described as a model, say M_1, in which it is encoded that the identifier x is bound to value 5. If we want to increment the value of x, we may need to characterize all those models M_2 in which x has value 6 and *nothing else has changed.* Specifying the precise formal meaning of this last clause is difficult computationally and conceptually. On the other hand, when derivations are used to represent computations directly, the frame problem is not solved but simply avoided: for example, backchaining over the clause

$$x \text{ has value } n \;\circ\!-\; x \text{ has value } n+1$$

might simply change the representation of state in the required fashion.

In the first approach to specification, there is a great deal of richness available for modeling computation, since, in principle, such disciplines as set theory, category theory, functional analysis, algebras, *etc.*, can be employed. This approach has had, of course, a great deal of success within the theory of computation.

In contrast, the second approach seems thin and feeble: the syntax of logical formulas and proofs contains only the most simple structures for representing computational state. What this approach lacks in expressiveness, however, is ameliorated by the fact that it is more intimately connected to computation. Deductions, for example, seldom make reference to infinity (something commonly done in the other approach) and steps within the construction of proofs are generally simple and effective computations. Recent developments in proof theory and logic programming have also provided us with logics that are surprisingly flexible and rich in their expressiveness. In particular, linear logic [10] provides flexible ways to model state, state transitions, and some simple concurrency primitives, and higher-order quantification over typed λ-terms provides for flexible notions of abstraction and encodings of object-level languages. Also, since specifications are written using logical formulas, specifications can be subjected to rich forms of analysis and transformations.

To design logics (or presentations of logics) for use in the computation-as-deduction setting, it has proved useful to provide a direct and natural operational interpretation of logical connective. To this end, the formalization of *goal-directed search* using *uniform proofs* [31, 34] associates a fixed, "search semantics" to logical connectives. When restricting to uniform proofs does not cause a loss of completeness, logical connectives can be interpreted as fixed search primitives. In this way, specifier can write declarative specifications that map directly to descriptions of computations. This analysis of goal-directed proof search has lead to the design of the logic programming languages λProlog, Lolli, LO, and Forum (see Section 3). Some simple examples with using these languages for specifications can be found in [2, 18, 31]. The recent thesis [5] provides two modest-sized Forum specifications: one being the operational semantics of a functional programming language containing references, exceptions, and continuation passing, and the other being a specification of a pipe-lined, RISC processor.

> *Observation 1.* Logic can be used to specify computation by encoding states and transitions directly using formulas and proof. This use of logic fits naturally in a logic programming setting where backchaining can denote state transition. Both linear logic and higher-order quantification can add greatly to the expressiveness of this paradigm.

1.2 An example

The following specification of reversing a list and the proof of its symmetry illustrates how the expressiveness of higher-order linear logic can provide for natural specifications and convenient forms of reasoning.

```
reverse L K :- pi rv\(
  pi X\(pi M\(pi N\(rv (X::M) N :- rv M (X::N)))) =>
    rv nil K -: rv L nil).
```

Here we use a variant of λProlog syntax: in particular, lists are constructed from the infix `::` and `nil`; `pi X\` denotes universal quantification of the variable `X`; `=>` denotes intuitionistic implication; and, `-:` and `:-` denote linear implication and its converse. This one example combines some elements of both linear logic and higher-order quantification.

To illustrate this specification, consider proving the query

```
?- reverse (a::b::c::nil) Q.
```

Backchaining on the definition of reverse above yields a goal universally quantified by `pi rv\`. Proving such a goal can be done by instantiating that quantifier with a new constant, say `rev`, and proving the result, namely, the goal

```
  pi X\(pi M\(pi N\(rev (X::M) N :- rev M (X::N)))) =>
     rev nil Q -: rev (a::b::c::nil) nil).
```

Thus, an attempt will be made to prove the goal `(rev (a::b::c::nil) nil)` from the two clauses

```
  pi X\(pi M\(pi N\(rev (X::M) N :- rev M (X::N)))).
  rev nil Q.
```

(Note that the variable `Q` in the last clause is free and not implicitly universally quantified.) Given the use of intuitionistic and linear implications, the first of these clauses can be used any number of times while the second must be used once (natural characterizations of inductive and initial cases for this example). Backchaining now leads to the following progression of goals:

```
    rev (a::b::c::nil)  nil.
    rev (b::c::nil) (a::nil).
    rev (c::nil) (b::a::nil).
    rev  nil  (c::b::a::nil).
```

and the last goal will be proved by backchaining against the initial clause and binding `Q` with `(c::b::a::nil)`.

It is clear from this specification of `reverse` that it is a symmetric relation: the informal proof simply notes that if the table of `rev` goals above is flipped horizontally and vertically, the result is the core of a computation of the symmetric version of reverse. Given the expressiveness of this logic, the formal proof of this fact directly incorporates this main idea.

Proposition. Let `l` and `k` be two lists and let $\mathcal{P}$ be a collection of clauses in which the only clause that contains an occurrence of `reverse` in its head is the one displayed above. If the goal `(reverse l k)` is provable from $\mathcal{P}$ then the goal `(reverse k l)` is provable from $\mathcal{P}$.

Proof. Assume that the goal `(reverse l k)` is provable from $\mathcal{P}$. Given the restriction on occurrences of `reverse` in $\mathcal{P}$, this goal is provable if and only if it is proved by backchaining with the above clause for `reverse`. Thus, the goal

```
pi rv\(
  pi X\(pi M\(pi N\(rv (X::M) N :- rv M (X::N)))) =>
  rv nil k -: rv l nil)
```

is provable from $\mathcal{P}$. Since this universally quantified formula is provable, any instance of it is provable. Let `rev` be a new constant not free in $\mathcal{P}$ of the same type as the variable `rv`. The formula that results from instantiating this quantified goal with the λ-term `x\y\(not (rev y x))` (where `\` is the infix symbol for λ-abstraction and `not` is the logical negation, often written in linear logic using the superscript $\perp$). The resulting formula,

```
pi X\(pi M\(pi N\(
         not (rev N (X::M)) :- not (rev (X::N) M)))) =>
not (rev k nil) -: not (rev nil l),
```

is thus provable from $\mathcal{P}$. This formula is logically equivalent to the following formula (linear implications and their contrapositives are equivalent in linear logic).

```
pi X\(pi M\(pi N\(rev (X::N) M :- rev N (X::M))))
=> rev nil l -: rev k nil
```

Since this code is provable and since the constant `rev` is not free in $\mathcal{P}$, we can universally generalize over it; that is, the following formula is also provable.

```
pi rev\(
    pi X\(pi M\(pi N\(rev (X::N) M :- rev N (X::M)))) =>
     rev nil l -: rev k nil)
```

From this goal and the definition of `reverse` (and α-conversion) we can prove `(reverse k l)`. Hence, `reverse` is symmetric. ∎

This proof should be considered elementary since it involves only simple linear logic identities and facts. Notice that there is no direct use of induction. The two symmetries mentioned above in the informal proof are captured in the higher-order substitution `x\y\(not (rev y x))`: the switching of the order of bound variables captures the vertical flip and linear logic negation (via contrapositives) captures the the horizontal flip.

1.3 Meta-programming and meta-logic

An exciting area of specification is that of specifying the meaning and behavior of programs and programming languages. In such cases, the code of a programming language must be represented and manipulated, and it is valuable to introduce the terms *meta-language* to denote the specification language and *object-language* to denote the language being specified.

Given the existence of two languages, it is natural to investigate the relationship that they may have to one another. That is, how can the meaning of object-language expressions be related to the meaning of meta-level expressions. One of the major accomplishments in mathematical logic in the first part of this century was achieved by K. Gödel by probing this kind of reflection, in this case, encoding meta-level formulas and proofs at the the object-level [12].

Although much of the work on meta-level programming in logic programming has also been focused on reflection, this focus is rather narrow and limiting: there are many other ways to judge the success of a meta-programming language apart from its ability to handle reflection. While a given meta-programming language might not be successful at providing novel encodings of itself, it might provide valuable and flexible encodings of other programming languages. For example, the π-calculus provides a revealing encoding of evaluation in the λ-calculus [35], evaluation in object-oriented programming [50], and interpretation of Prolog programs [23]. Even the semantic theory of the π-calculus can be fruitfully exploited to probe the semantics of encoded object-languages [47]. While it has been useful as a meta-language, it does not seem that the π-calculus would yield an interesting encoding of itself.

Similarly, λProlog has been successful in providing powerful and flexible specifications of functional programming languages [13, 41] and natural deduction proof systems [8]. Forum has similarly been used to specify sequent calculi and various features of programming languages [5, 31]. It is not clear, however, that λProlog or Forum would be particularly good for representing their own operational semantics.

> *Observation 2.* A meta-programming language does not need to capture its own semantics to be useful. More importantly, it should be able to capture the semantics of a large variety of languages and the resulting encoding should be direct enough that the semantics of the meta-language can provide semantically meaningful information about the encoded object-language.

A particularly important aspect of meta-programming is the choice of encodings for object-level expressions. Gödel used natural numbers and the prime factorization theorem to encode syntactic values: an encoding that does not yield a transparent nor declarative approach to object-level syntax. Because variables in logic programming range over expressions, representing object-level syntax can be a particularly simple, at least for certain expressions of the object language. For example, the meaning of a type in logic programming, particularly types as they are used in λProlog, is a set of *expressions* of a given type. In contrast, types in functional programming (say, in SML) generally denote sets of *values*. While the distinction between expressions and values can be cumbersome at times in logic programming (`2 +`

`3` is different than `5`), it can be useful in meta-programming. This is particularly true when dealing with expressions of functional type. For example, the type `int -> int` in functional programming denotes functions from integers to integers: checking equality between two such functions is not possible, in general. In logic programming, particularly in λProlog, this same type contains the code of expressions (not functions) of that type: thus it is possible to represent the syntax of higher-order operations in the meta-programming language and meaningfully compare and compute on these codes. More generally, meta-level types are most naturally used to represent object-level syntactic categories. When using such an encoding of object-level languages, meta-level unification and meta-level variables can be used naturally to probe the structure of object-level syntax.

> *Observation 3.* Since types and variables in logic programming range over expressions, the problem of naming object-level expressions is often easy to achieve and the resulting specifications are natural and declarative.

1.4 Higher-order abstract syntax

In the last observation, we used the phrase "often easy to achieve." In fact, if object-level expressions contain bound variables, it is a common observation that representing such variables using only first-order expressions is problematic since notions of bound variable names, equality up to α-conversion, substitution, *etc.*, are not addressed naturally by the structure of first-order terms. From a logic programming point-of-view this is particularly embarrassing since all of these notions are part of the meta-theory of quantification logic: since these issues exist in logic generally, it seems natural to expect a logical treatment of them for object-languages that are encoded into logic. Fortunately, the notion of *higher-order abstract syntax* is capable of declaratively dealing with these aspects of object-level syntax.

Higher-order abstract syntax involves two concepts. First, λ-terms and their equational theory should be used uniformly to represent syntax containing bound variables. Already in [6], Church was doing this to encode the universal and existential quantifiers and the definite description operator. Following this approach, instantiation of quantifiers, for example, can be specified using β-reduction.

The second concept behind higher-order abstract syntax is that operations for composing and decomposing syntax must respect at least α-conversion of terms. This appears to have first been done by Huet and Lang in [19]: they discussed the advantages of representing object-level syntax using simply typed λ-terms and manipulating such terms using matching modulo the equational rules for λ-conversion. Their approach, however, was rather weak since it only used matching (not unification more generally). That restrictions made it impossible to express all but the simplest operations on syntax. Their approach was extended by Miller and Nadathur in [33] by moving to a logic programming setting that contained $\beta\eta$-unification of simply typed λ-terms. In that paper the central ideas and advantages behind higher-order abstract syntax are discussed. In the context of theorem proving, Paulson also independently proposed similar ideas [39].

In [43] Pfenning and Elliot extended the observations in [33] by producing examples where the meta-language that incorporated λ-abstractions contained not just simple types but also product types. In that paper they coined the expression "higher-order abstract syntax." At about this time, Harper, Honsell, and Plotkin in [15] proposed representing logics in a dependent typed λ-calculus. While they did not deal with the computational treatment of syntax directly, that treatment was addressed later by considering the unification of dependent typed λ-expressions by Elliott [7] and Pym [45].

The treatment of higher-order abstract syntax in the above mentioned papers had a couple of unfortunate aspects. First, those treatments involved unification with respect to the full $\beta\eta$-theory of the λ-calculus, and this general theory is computational expensive. In [19], only second-order matching was used, an operation that is NP-complete; later papers used full, undecidable unification. Second, various different type systems were used with higher-order abstract syntax, namely simple types, product types, and dependent types. However, if abstract syntax is essentially about a treatment of bound variables in syntax, it should have a presentation that is independent from typing.

The introduction of L_λ in [29] provided solutions to both of these problems. First, L_λ provides a setting where the unification of λ-terms is decidable and has most general unifiers: it was shown by Qian [46] that L_λ-unification can be done in linear time and space (as with first-order unification). Nipkow showed that the exponential unification algorithm presented in [29] can be effectively used within theorem provers [38]. Second, it was also shown in [29] that L_λ-unification can be described for *untyped* λ-terms: that is, typing may impose additional constraints on unification but L_λ-unification can be defined without types. Thus, it is possible then to define L_λ-like unification for various typed calculi [42].

> *Observation 4.* L_λ appears to be one of the weakest settings in which higher-order abstract syntax can be supported. The main features of L_λ can be merged with various logical systems (say, λProlog and Forum), with various type systems (say, simple types and dependent types) [41], and with equational reasoning systems [37, 44].

While existing implementations of λProlog, Isabelle, Elf, and NuPRL all make use of results about L_λ, there is currently no direct implementation of L_λ. It should be a small and flexible meta-logic specification language.

2 Logic programming and linear logic

The previous section described some of the advantages of using a rich and expressive logic as the foundation of a programing language. In the next several sections, we consider how to shape higher-order linear logic into a logic programming language and discuss some of the advantages that are derived from using such a logic for specifications.

In [34] 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 [18] 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 [9]. 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 [14]. 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 [20] 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 [10] (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 $o \rightarrow o \rightarrow o$ and the quantifiers $\forall_\tau$ and $\exists_\tau$ have the type $(\tau \rightarrow o) \rightarrow o$, 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 4, 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 5 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 6, and various imperative features of a object-level programming language are specified and analyzed in Section 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 [21, 22] 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 [24, 48].

3 Designing Forum

The following generalization of the definition of uniform proof was introduced in [30] where it was shown that a certain logic specification inspired by the π-calculus [36] 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* [34], the 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 [34]: 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 super-language: 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.

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

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

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

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 [8] 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 5.

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 6 and 7 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

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

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

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

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 $\invamp$ ($p = 0$) is written as $\perp$. The formula $A_1 \invamp \cdots \invamp A_p$ 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.

4 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 $\perp$, $\invamp$, $\top$, $\&$, $\multimap$, $\Rightarrow$, ?, 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 [28].) 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 $\mathcal{A}, B \,\&\, C, \Gamma$ 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 [11] except that we have followed the tradition of [1, 17] 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, 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]. 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 [10].

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Sigma\colon \Psi; \Delta \Uparrow L}{\Sigma\colon \Psi; \Delta \Uparrow \bot, L}\ [\bot] \qquad \frac{\Sigma\colon \Psi; \Delta \Uparrow F, G, L}{\Sigma\colon \Psi; \Delta \Uparrow F \parr G, L}\ [\parr] \qquad \frac{\Sigma\colon \Psi, F; \Delta \Uparrow L}{\Sigma\colon \Psi; \Delta \Uparrow ?\, F, L}\ [?]$$

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

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

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

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

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

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

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

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

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 $i = 2$.

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.

Theorem 2 (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 $\mathcal{F}$.*

See [27] for the proof. 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.

5 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 7 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.

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

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

$$\Sigma\colon \Psi; \Delta_1 \longrightarrow G_1, \mathcal{A}_1; \Upsilon \qquad \Sigma\colon \Psi; \cdot \longrightarrow G_2; \Upsilon$$
$$\Sigma\colon \Psi; \Delta_2 \longrightarrow G_3, \mathcal{A}_2; \Upsilon \qquad \Sigma\colon \Psi; \cdot \longrightarrow G_4; \Upsilon$$

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. In particular we have the following corollary.

Corollary 3 *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 [26] 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.)

6 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

$$\mathit{left}\ B_1 \parr \cdots \parr \mathit{left}\ B_n \parr \mathit{right}\ C_1 \parr \cdots \parr \mathit{right}\ C_m.$$

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

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

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

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

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

$$?\,\mathit{left}\ B_1 \parr \cdots \parr ?\,\mathit{left}\ B_n \parr ?\,\mathit{right}\ C_1 \parr \cdots \parr ?\,\mathit{right}\ C_m.$$

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.

$$\begin{array}{lrl}
(\supset R) & \mathit{right}\ (A \supset B) & \multimap (?(\mathit{left}\ A) \parr \mathit{right}\ B). \\
(\supset L) & ?(\mathit{left}\ (A \supset B)) & \multimap \mathit{right}\ A \multimap ?(\mathit{left}\ B). \\
(\wedge R) & \mathit{right}\ (A \wedge B) & \multimap \mathit{right}\ A \multimap \mathit{right}\ B. \\
(\wedge L_1) & ?(\mathit{left}\ (A \wedge B)) & \multimap ?(\mathit{left}\ A). \\
(\wedge L_2) & ?(\mathit{left}\ (A \wedge B)) & \multimap ?(\mathit{left}\ B). \\
(\hat\forall R) & \mathit{right}\ (\hat\forall B) & \multimap \forall x(\mathit{right}\ (Bx)). \\
(\hat\forall L) & ?(\mathit{left}\ (\hat\forall B)) & \multimap ?(\mathit{left}\ (Bx)). \\
(\mathit{Initial}) & \multicolumn{2}{l}{\mathit{right}\ B \parr ?(\mathit{left}\ B).} \\
(\mathit{Cut}) & \bot & \multimap ?(\mathit{left}\ B) \multimap \mathit{right}\ B.
\end{array}$$

Figure 3: Specification of the LM_1 sequent calculus.

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 $\mathit{bool} \to \mathit{bool} \to \mathit{bool}$, and the symbols $\hat\forall$ and $\hat\exists$ of type $(i \to \mathit{bool}) \to \mathit{bool}$: 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 [9] 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 [8], Isabelle [40], and Elf [41]. 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

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

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

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

which in turns is provable if and only if the sequent

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

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

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

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:

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

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 4, 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 ($Initial$) and (Cut) clauses to see these are equivalent to the inference rules

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

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 [9].

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

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 4).

Proposition 4 (Correctness of LM_1) *The sequent $B_1, \ldots, B_n \longrightarrow B_0$ has an LM_1-proof if and only if the sequent*

$$\Sigma_1 \colon LM_1; \cdot \longrightarrow \mathit{right}\ B_0; \mathit{left}\ B_1, \ldots, \mathit{left}\ B_n$$

has a proof in $\mathcal{F}$ (here, $n \geq 0$).

The proof of the missing propositions and theorems in this section can be found in [27].

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 [8, 40] and dependent typed calculi [3, 16].

Proposition 5 (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 [8, 16, 40] 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 [18] 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 6 $\vdash LM_1 \equiv [(\otimes NM_1) \otimes \mathit{Initial} \otimes \mathit{Cut}]$.

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

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

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

Figure 4: Specification of the NM_1 natural deduction calculus.

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 10), we need a couple of technical lemmas.

Lemma 7 *If $\Sigma_1\colon NM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ has a proof in $\mathcal{F}$, then $\Sigma_1\colon LM_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ has a proof in $\mathcal{F}$.*

Proof This follows directly from Lemma 6, cut-elimination for linear logic, and the soundness and completeness results for $\mathcal{F}$. ▮

Lemma 8 *If $\Sigma_1\colon NM_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \cdot$ 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 Let Ξ be a proof in $\mathcal{F}$ of $\Sigma_1\colon NM_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \cdot$. 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

$$\Sigma\colon \mathit{NM}_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \mathcal{L}$$

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 $\Sigma_1\colon \mathit{NM}_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \cdot$. 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 $\Sigma_1\colon \mathit{NM}_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$. ▮

Lemma 9 *If* $\Sigma_1\colon \mathit{LM}_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ *has a proof in* $\mathcal{F}$*, then* $\Sigma_1\colon \mathit{NM}_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ *has a proof in* $\mathcal{F}$*.*

Proof Assume $\Sigma_1\colon \mathit{LM}_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ has a proof in $\mathcal{F}$. Using Lemma 6, cut-elimination for linear logic, and the soundness and completeness results for $\mathcal{F}$, the sequent

$$\Sigma_1\colon \mathit{NM}_1, \mathit{Cut}, \mathit{Initial}; \cdot \longrightarrow \mathit{right}\ B; \cdot$$

has a proof in $\mathcal{F}$. Now using Lemma 8, we have that $\Sigma_1\colon \mathit{NM}_1; \cdot \longrightarrow \mathit{right}\ B; \cdot$ has a proof in $\mathcal{F}$. ▮

The following theorem follows from results of Gentzen [9]. We supply a new proof here using linear logic as a meta-theory.

Theorem 10 *The sequent* $\longrightarrow B$ *has an* LM_1 *proof if and only if* B *has an* NM_1*-proof (from no assumptions).*

Proof This theorem follows immediately from Propositions 4 and 5 and Lemmas 7 and 9. ▮

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.

Using the equivalences $(\mathit{right}\ B)^\perp \equiv ?(\mathit{left}\ B)$ and $(\mathit{right}\ B) \equiv !(\mathit{right}\ B)$ with 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

$$(\mathit{right}\ A) \oplus (\mathit{right}\ B) \multimapinv \mathit{right}\ (A \vee B).$$
$$\exists x(\mathit{right}\ (Bx)) \multimapinv \mathit{right}\ (\hat{\exists} B).$$

$$\begin{array}{cr@{\ }l}
(\vee R_1) & \mathit{right}\ (A \vee B) & \multimap \mathit{right}\ A. \\
(\vee R_2) & \mathit{right}\ (A \vee B) & \multimap \mathit{right}\ B. \\
(\vee L) & ?(\mathit{left}\ (A \vee B)) & \multimap ?(\mathit{left}\ A)\ \&\ ?(\mathit{left}\ B). \\
(\hat{\exists} R) & \mathit{right}\ (\hat{\exists} B) & \multimap \mathit{right}\ (Bx). \\
(\hat{\exists} L) & ?(\mathit{left}\ (\hat{\exists} B)) & \multimap \forall x(?(\mathit{left}\ (Bx))).
\end{array}$$

Figure 5: Sequent rules for disjunction and existential quantification.

$$\begin{array}{cr@{\ }l}
(\vee I_1)' & \mathit{right}\ (A \vee B) & \multimap \mathit{right}\ A. \\
(\vee I_2)' & \mathit{right}\ (A \vee B) & \multimap \mathit{right}\ B. \\
(\vee E)' & \bot & \multimap \mathit{right}\ (A \vee B) \\
 & & \multimap (\mathit{right}\ A \Rightarrow \bot)\ \&\ (\mathit{right}\ B \Rightarrow \bot). \\
(\hat{\exists} I)' & \mathit{right}\ (\hat{\exists} B) & \multimap \mathit{right}\ (Bx). \\
(\hat{\exists} E)' & \bot & \multimap \mathit{right}\ (\hat{\exists} B) \\
 & & \multimap \forall x(\mathit{right}\ (Bx) \Rightarrow \bot).
\end{array}$$

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

$$\begin{array}{cr@{\ }l}
(\vee I_1) & \mathit{right}\ (A \vee B) & \multimap \mathit{right}\ A. \\
(\vee I_2) & \mathit{right}\ (A \vee B) & \multimap \mathit{right}\ B. \\
(\vee E) & \mathit{right}\ E & \multimap \mathit{right}\ (A \vee B) \\
 & & \multimap (\mathit{right}\ A \Rightarrow \mathit{right}\ E) \\
 & & \multimap (\mathit{right}\ B \Rightarrow \mathit{right}\ E). \\
(\hat{\exists} I) & \mathit{right}\ (\hat{\exists} B) & \multimap \mathit{right}\ (Bx). \\
(\hat{\exists} E) & \mathit{right}\ E & \multimap \mathit{right}\ (\hat{\exists} B) \\
 & & \multimap \forall x(\mathit{right}\ (Bx) \Rightarrow \mathit{right}\ E).
\end{array}$$

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

(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 6 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

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

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 10 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 [18].

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 [8], Isabelle [40], and Elf [41]. 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, [8]). 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.

7 Operational Semantics Examples

Evaluation of pure functional programs has been successfully specified in intuitionistic meta-logics [13] and type theories [4, 41] 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 $\mathit{tm} \to \mathit{tm} \to \mathit{tm}$) and abstraction (at type $(\mathit{tm} \to \mathit{tm}) \to \mathit{tm}$). 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 $\mathit{tm} \to \mathit{tm} \to o$ (see, for example, [13]).

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

In order to add side-effecting features, this specification must be made more explicit: in particular, the exact order in which M, N, and $(R\ U)$ are evaluated must be specified. Using a "continuation-passing" technique from logic programming [49], this ordering can be made explicit using the following two clauses, this time using the predicate *eval* at type $\mathit{tm} \to \mathit{tm} \to o \to o$.

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

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

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

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

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

Figure 8: Three specifications of a global counter.

argument of the predicate r. In these three specifications, the predicate r is existentially quantified over the specification in which it is used so that the atomic formula that stores the counter's value is itself local to the counter's specification (such existential quantification of predicates is a familiar technique for implementing abstract data types in logic programming [25]). 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 [18]). 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 11 *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

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

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

$$\begin{aligned}
\mathit{eval}\ (\mathit{read}\ L)\ V\ K \parr \mathit{ref}\ L\ V &\circ\!-\ K \parr \mathit{ref}\ L\ V. \\
\mathit{assign}\ L\ V\ K \parr \mathit{ref}\ L\ U &\circ\!-\ K \parr \mathit{ref}\ L\ V.
\end{aligned}$$

Figure 9: Specification of references.

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 $\{h_1, \ldots, h_n\}$. 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

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

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 $\mathit{eval}\ M\ V\ \top \parr S$ is provable: that is, the value of M is V if the store is initially S. The references specified here obey a block structured discipline 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

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

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

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

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 [36].

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

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

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

8 Some exercises

Problems 1 and 2 require proofs that will involve permutations of inferences and induction of the structure of proofs. These two problems will probably be the most difficult.

Problems 3, 4, and 5 involve analyzing and writing particular logic programs illustrating linear logic features. For related example programs, see [18].

Problem 6 concerns working out an example and proving a theorem about the π-calculus. For this problem, see [30].

8.1 Provability using Horn clauses

In the Section 8.7, a proof system and terminology is introduced (for this question only). With respect to the terms defined there, show the following. Let D and G-formulas be defined as follows (these are first-order Horn clauses).

$$G ::= A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_i x G$$
$$D ::= A \mid G \supset D \mid D_1 \wedge D_2 \mid \forall_i x D,$$

where, of course, A is a syntactic variable ranging over first-order atomic formulas. (Assume that the only domain type is i.) Now let $\mathcal{P}$ be a finite set of D-formulas and let $\mathcal{G}$ be a finite set of G-formulas. Carefully prove each of the following.

1. It is never the case that $\Sigma; \mathcal{P} \vdash_C \bot$. Notice that $\bot$ is not considered to be an atomic formula.
2. If $\Sigma; \mathcal{P} \vdash_C \mathcal{G}$ then there exists a $G \in \mathcal{G}$ such that $\Sigma; \mathcal{P} \vdash_C G$.
3. If $\Sigma; \mathcal{P} \vdash_C G$ then the sequent $\Sigma\, ;\, \mathcal{P} \longrightarrow G$ has a uniform proof (in the single-conclusion sense).
4. $\Sigma; \mathcal{P} \vdash_C G$ if and only if $\Sigma; \mathcal{P} \vdash_M G$.

8.2 A proof system for LO

The LO logic programming language is based on clauses of the following form.

$$G ::= \bot \mid \top \mid A \mid G_1 \mathbin{\&} G_2 \mid G_1 \invamp G_2$$
$$D ::= G \multimap (A_1 \invamp \ldots \invamp A_n) \mid \forall_i x D,$$

where $n \geq 1$ and, of course, A is a syntactic variable ranging over first-order atomic formulas. (Assume that the only domain type is i.) The following proof system is specialized for just LO: sequents in the proof system are such that formulas on the left of the arrow are D-formulas and formulas on the right are G-formulas.

$$\frac{}{\mathcal{P} \longrightarrow \Gamma, \top} \quad \frac{\mathcal{P} \longrightarrow \Gamma}{\mathcal{P} \longrightarrow \Gamma, \bot} \quad \frac{\mathcal{P} \longrightarrow \Gamma, G_1, G_2}{\mathcal{P} \longrightarrow \Gamma, G_1 \invamp G_2} \quad \frac{\mathcal{P} \longrightarrow \Gamma, G_1 \quad \mathcal{P} \longrightarrow \Gamma, G_2}{\mathcal{P} \longrightarrow \Gamma, G_1 \mathbin{\&} G_2}$$

$$\frac{\mathcal{P} \longrightarrow \Gamma, G}{\mathcal{P} \longrightarrow \Gamma, A_1, \ldots, A_n}$$ Provided there is a formula in $\mathcal{P}$ whose ground instance is $G \multimap (A_1 \invamp \ldots \invamp A_n)$.

Let G be a goal formula, let $\mathcal{P}$ be a finite set of D-formulas, and let Σ be the signature containing the non-logical constants in G and $\mathcal{P}$. Show that the sequent $\mathcal{P} \longrightarrow G$ has a proof in the system above if and only if $\Sigma : \mathcal{P}; \longrightarrow G$ has a proof in the linear logic proof system used in lectures.

8.3 Computing the maximum of a list

This problem concerns computing the maximum of a multiset of integers. Assume that you have the predicates `(greaterEq N M)` and `(lesser N M)` that are provable (consuming no resources) if and only if `N` is greater than or equal to `M` and (respectively) `N` is less than `M`.

1. Write a logic program $\mathcal{P}_1$ for the predicate $maxA$ such that the sequent

$$\Sigma : \mathcal{P}_1; A(n_1), \ldots, A(n_m) \longrightarrow maxA(n)$$

is provable if and only if n is the maximum of $\{n_1, \ldots, n_m\}$. (Here, as in the next problem, if $m = 0$ then set the maximum to be 0.)

2. Write a logic program $\mathcal{P}_2$ for the predicate $maxA$ such that the sequent

$$\Sigma : \mathcal{P}_2; \longrightarrow maxA(n), A(n_1), \ldots, A(n_m)$$

is provable if and only if n is the maximum of $\{n_1, \ldots, n_m\}$.

8.4 Using the left and right contexts

Below are specifications of two binary predicates.

```
pred1 L K <=  pi load\(pi unload\(pi m\(
 (pi X\(pi M\(   load (X::M) :- (m X -:   load M) ))) =>
 (pi X\(pi M\( unload (X::M) :-  m X,   unload M) ))) =>
 (load nil :- unload K) -: (unload nil) -: (load L)))).

pred2 L K <=  pi load\(pi unload\(pi m\(
 (pi X\(pi M\(  load (X::M)       :- m X | load M))) =>
 (pi X\(pi M\(unload (X::M) | m X :- unload M    ))) =>
 (load nil   :-  unload K) -: (unload nil) -: (load L)))).
```

Here, we use `pi token\` to denote universal quantification over `token` and use `|` to denote "par" (multipicative disjunction). The comma is used to denote "tensor" (multiplicative conjunction). The implication signs `-:` and `=>` associate to the right.

1. It turns out that both of these clauses specify the same relation. What is that relation? Informally justify your answer.
2. Formally prove that each of these specifications compute the same relation by a logical transformation of one to the other using a technique similar to that used in lectures to show that `reverse` is symmetric.

8.5 An example of a linear logic program

Below is the specification of two predicates. The `greaterEq` is the same of in the problem above.

```
mx N.
mx N :- a M, greaterEq N M, mx N.
sr nil.
sr (N::L) :- a N, (mx N & sr L).
```

Let $\mathcal{P}$ be the set containing these four clauses. Let $\mathcal{A}$ be the multiset of atomic formulas $\{a(i_1), \ldots, a(i_n)\}$, where $\{i_1, \ldots, i_n\}$ $(n \geq 0)$ is a multiset of positive integers. Describe when it is the case that the linear sequent

$$\Sigma : \mathcal{P}; \mathcal{A} \longrightarrow \texttt{(sr L)},$$

is provable. Explain your reason.

8.6 Encoding the π-calculus into linear logic

Consider the following two π-calculus agents.

$$P = x(y).y(w).(v).\bar{w}v.yb.nil \mid u(r).\bar{r}a.nil \mid (z).\bar{x}z.\bar{z}u.nil$$

$$Q = ((z).zb.nil) \mid (v)\bar{v}a.nil$$

1. Using the unlabeled transitions for the π-calculus, show that P reduces to Q.
2. Let P° and Q° be the formulas (over the non-logical constants *get* and *send*) that are the result of translating these agents into linear logic. Produce a proof in linear logic of the sequent

   $$\Sigma : \Pi; Q^0 \longrightarrow P^0.$$

   Here, Π is the formula

   $$\forall x \forall z \forall P \forall Q (P[z/y] \parr Q \multimap x(y).P \parr \bar{x}z.Q),$$

   and Σ is a signature containing the constants x, u, a, and b.
3. Let G and H be two linear logic formulas that are the result of translating two π-calculus agents into linear logic and let Σ be the constants contained in both G and H. Prove the following fact: If the sequent

   $$\Sigma : \Pi; G \longrightarrow H$$

   has a proof it has a proof Ξ with the following structure: there is some sequent in Ξ such that all inference rules below it are either right-introduction rules for $\bot$, $\parr$, and $\forall$ or are backchaining steps over the formula Π, and all inference rules above it are left-introduction rules for $\bot$, $\parr$ and $\forall$ or initial sequents.

8.7 Proof systems for question 1

Provability for $\mathcal{F}$ is given in terms of sequent calculus proofs. A *sequent* of $\mathcal{F}$ is a triple $\Sigma\ ;\ \Gamma \longrightarrow \Delta$, where Σ is a first-order signature over S and Γ and Δ are finite (possibly empty) sets of Σ-formulas. The set Γ is this sequent's *antecedent* and Δ is its *succedent.* The expressions Γ, B and B, Γ denote the set $\Gamma \cup \{B\}$; this notation is used even if $B \in \Gamma$. The following provisos are also attached to the four inference rules for quantifier introduction: in $\forall$-R and $\exists$-L, the constant $\mathbf{c}$ is not in Σ, and, in $\forall$-L and $\exists$-R, $\mathbf{t}$ is a Σ-term of type τ.

A *proof* of the sequent $\Sigma\ ;\ \Gamma \longrightarrow \Theta$ is a finite tree constructed using these inference rules such that the root is labeled with $\Sigma\ ;\ \Gamma \longrightarrow \Theta$ and the leaves are labeled with *initial sequents*, that is, sequents $\Sigma'\ ;\ \Gamma' \longrightarrow \Theta'$ such that either $\top$ is a member of Θ' or the intersection $\Gamma' \cap \Theta'$ contains either $\perp$ or an atomic formula.

Sequent systems generally have three *structural* rules that are not listed here. Two such rules, interchange and contraction, are not necessary here because the antecedents and succedents of sequents are sets instead of lists. Hence, the order and multiplicity of formulas in sequents are not made explicit. The third common structural rule is that of weakening: from a given sequent one may add any additional formulas to the succedent and antecedent. Weakening could be added as a derived inference rule, but it is not needed here.

Any proof is also called a **C**-*proof.* Any **C**-proof in which the succedent of every sequent in it is a singleton set is also called an **I**-*proof.* Furthermore, an **I**-proof in which no instance of the $\perp$-R inference rule appears is also called an **M**-*proof.* Sequent proofs in classical, intuitionistic, and minimal logics are represented by, respectively, **C**-proofs, **I**-proofs, and **M**-proofs. Finally, let Σ be a given first-order signature over S, let Γ be a finite set of Σ-formulas, and let B be a Σ-formula. We write $\Sigma; \Gamma \vdash_C B$, $\Sigma; \Gamma \vdash_I B$, and $\Sigma; \Gamma \vdash_M B$ if the sequent $\Sigma\ ;\ \Gamma \longrightarrow B$ has, respectively, a **C**-proof, an **I**-proof, or an **M**-proof. It follows immediately that $\Sigma; \Gamma \vdash_M B$ implies $\Sigma; \Gamma \vdash_I B$, and this in turn implies $\Sigma; \Gamma \vdash_C B$.

Acknowledgments

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.

The material in this chapter has been taken largely from the following sources. Section 1 has been taken from [32]. Sections 2 through 7 have been taken from [27], which is itself an extended version of a paper that appeared as [31] and which was also presented at the 1994 Joint Meeting of ALP and PLILP, Madrid, September 1994. Section 8 contains exercises I used in a

$$\dfrac{\Sigma\,;\,\Gamma \longrightarrow \Delta, B \qquad \Sigma\,;\,\Gamma \longrightarrow \Delta, C}{\Sigma\,;\,\Gamma \longrightarrow \Delta, B \wedge C}\ \wedge\text{-R} \qquad \dfrac{\Sigma\,;\,B, C, \Delta \longrightarrow \Theta}{\Sigma\,;\,B \wedge C, \Delta \longrightarrow \Theta}\ \wedge\text{-L}$$

$$\dfrac{\Sigma\,;\,B, \Delta \longrightarrow \Theta \qquad \Sigma\,;\,C, \Delta \longrightarrow \Theta}{\Sigma\,;\,B \vee C, \Delta \longrightarrow \Theta}\ \vee\text{-L}$$

$$\dfrac{\Sigma\,;\,\Gamma \longrightarrow \Delta, B}{\Sigma\,;\,\Gamma \longrightarrow \Delta, B \vee C}\ \vee\text{-R} \qquad \dfrac{\Sigma\,;\,\Gamma \longrightarrow \Delta, C}{\Sigma\,;\,\Gamma \longrightarrow \Delta, B \vee C}\ \vee\text{-R}$$

$$\dfrac{\Sigma\,;\,\Gamma \longrightarrow \Theta, B \qquad \Sigma\,;\,C, \Gamma \longrightarrow \Delta}{\Sigma\,;\,B \supset C, \Gamma \longrightarrow \Theta \cup \Delta}\ \supset\text{-L} \quad \dfrac{\Sigma\,;\,B, \Gamma \longrightarrow \Theta, C}{\Sigma\,;\,\Gamma \longrightarrow \Theta, B \supset C}\ \supset\text{-R}$$

$$\dfrac{\Sigma\,;\,\Gamma, [\mathbf{x} \mapsto \mathbf{t}]B \longrightarrow \Theta}{\Sigma\,;\,\Gamma, \forall_\tau x\, B \longrightarrow \Theta}\ \forall\text{-L} \qquad \dfrac{\Sigma\,;\,\Gamma \longrightarrow \Theta, [\mathbf{x} \mapsto \mathbf{t}]B}{\Sigma\,;\,\Gamma \longrightarrow \Theta, \exists x\, B}\ \exists\text{-R}$$

$$\dfrac{\Sigma \cup \{\mathbf{c}\colon \tau\}\,;\,\Gamma, [\mathbf{x} \mapsto \mathbf{c}]B \longrightarrow \Theta}{\Sigma\,;\,\Gamma, \exists_\tau \mathbf{x}\, B \longrightarrow \Theta}\ \exists\text{-L} \quad \dfrac{\Sigma \cup \{\mathbf{c}\colon \tau\}\,;\,\Gamma \longrightarrow \Theta, [\mathbf{x} \mapsto \mathbf{c}]B}{\Sigma\,;\,\Gamma \longrightarrow \Theta, \forall_\tau \mathbf{x}\, B}\ \forall\text{-R}$$

$$\dfrac{\Sigma\,;\,\Gamma \longrightarrow \Theta, \perp}{\Sigma\,;\,\Gamma \longrightarrow \Theta, B}\ \perp\text{-R}$$

Figure 10: A proof system $\mathcal{F}$ for classical, intuitionistic, and minimal logics.

course given at the University of Pisa during July 1994.

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] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, volume 355, pages 121–136. Springer-Verlag LNCS, April 1989.

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

[9] 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.

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

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

[12] Kurt Gödel. On formally undecidable propositions of the principia mathematica and related systems. I. In *Martin Davis, The Undecidable.* Raven Press, 1965.

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

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

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

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

[17] 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.

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

[19] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

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

[21] 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.

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

[23] Benjamin Li. A π-calculus specification of Prolog. In *Proc. ESOP 1994*, 1994.

[24] Patrick Lincoln and Vijay Saraswat. Higher-order, linear, concurrent constraint programming. January 1993. Available on the world-wide web at the url file://parcftp.xerox.com/pub/ccp/lcc/hlcc.dvi.

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

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

[27] Dale Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*. To appear.

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

[29] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[30] 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.

[31] 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.

[32] Dale Miller. Observations about using logic as a specification language. In M. Sessa, editor, *Proceedings of GULP-PRODE'95: Joint Conference on Declarative Programming*, Marina di Vietri (Salerno-Italy), September 1995.

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

[34] 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.

[35] Robin Milner. Functions as processes. In *Automata, Languages and Programming 17th Int. Coll.*, volume 443 of *LNCS*, pages 167–180. Springer Verlag, July 1990.

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

[37] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.

[38] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE, June 1993.

[39] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

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

[41] 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.

[42] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 74–85. IEEE, July 1991.

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

[44] Christian Prehofer. *Solving Higher-Order Equations: From Logic to Programming.* PhD thesis, Technische Universität München, 1995.

[45] David Pym. *Proofs, Search and Computation in General Logic.* PhD thesis, LFCS, University of Edinburgh, 1990.

[46] Zhenyu Qian. Linear unification of higher-order patterns. In J.-P. Jouannaud, editor, *Proc. 1993 Coll. Trees in Algebra and Programming.* Springer Verlag LNCS, 1993.

[47] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, May 1994.

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

[49] 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.

[50] David Walker. π-calculus semantics of object-oriented programming languages. LFCS Report Series ECS-LFCS-90-122, University of Edinburgh, October 1990.