Sequent Calculus and the Specification of Computation

Lecture Notes

Draft: 5 September 2001

These notes are being used with lectures for CSE 597 at Penn State in Fall 2001. These notes had been prepared to accompany lectures given at the Marktoberdorf Summer School, 29 July – 10 August 1997 for a course titled "Proof Theoretic Specification of Computation". An earlier draft was used in courses given between 27 January and 11 February 1997 at the University of Siena and between 6 and 21 March 1997 at the University of Genoa. A version of these notes was also used at Penn State for CSE 522 in Fall 2000. For the most recent version of these notes, contact the author.

© **Dale Miller**
321 Pond Lab
Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802-6106 USA
Office: +(814)865-9505
`dale@cse.psu.edu`
`http://www.cse.psu.edu/~dale`

Contents

1 Overview and Motivation **5**
1.1 Roles for logic in the specification of computations 5
1.2 Desiderata for declarative programming 6
1.3 Relating algorithm and logic 6

2 First-Order Logic **8**
2.1 Types and signatures 8
2.2 First-order terms and formulas 8
2.3 Sequent calculus 10
2.4 Classical, intuitionistic, and minimal logics 10
2.5 Permutations of inference rules 12
2.6 Cut-elimination 12
2.7 Consequences of cut-elimination 13
2.8 Additional readings 13
2.9 Exercises 13

3 Logic Programming Considered Abstractly **15**
3.1 Problems with proof search 15
3.2 Interpretation as goal-directed search 16
3.3 The syntax of first-order Horn clauses 18
3.4 Proof search with first-order Horn clauses 19
3.5 Additional readings 21
3.6 Exercises 21

4 Hereditary Harrop Formulas **22**
4.1 Harrop formulas 22
4.2 Three presentations of *fohh* 22
4.3 The core of a logic programming language 23
4.4 Proof search with first-order hereditary Harrop formulas 24
4.5 Multiple conclusion sequents and scope extrusion 24
4.6 A Kripke model semantics 25

5 Intuitionistic Linear Logic **29**
5.1 Weaknesses of hereditary Harrop formulas 29
5.2 Sequent calculus for linear logic 29
5.3 Uniform proofs in intuitionistic linear logic 30
5.4 An embedding of hereditary Harrop formulas 32
5.5 Allowing right rules for some additional connectives 33
5.6 An example 34
5.7 Additional readings 35
5.8 Exercises 36

6 Forum **37**
6.1 Designing Forum 37
6.2 Proof system for Forum 40
6.3 Multiset rewriting as backchaining 40

6.4 Further readings 42
6.5 Exercises 42

Abstract

The sequent calculus has been used for many purposes in recent years within theoretical computer science. In these lectures, we shall highlight some of its uses in the specification of and reasoning about computation.

During the search for cut-free sequent proofs, the formulas in sequents are re-arranged and replaced with other formulas. Such changes can be used to model the dynamics of computation in a wide range of applications. For various reasons, we shall be interested in "goal-directed proof search" and will examine intuitionistic logic and linear logic for subsets that support this particularly simple form of search. We will show, quite surprisingly, that with the appropriate selection of logical connectives, goal-directed search is complete for all of linear logic. This fact leads naturally to the design of a logic programming-like language based on linear logic. The resulting specification language, called Forum, is an expressive and rich specification language suitable for a wide range of computational paradigms.

After providing an overview of sequent calculus principles, we shall develop the notion of goal directed search for a variety of logics, starting with the intuitionistic logic theory of Horn clauses and hereditary Harrop formulas. We shall provide various example specifications in these logics, especially examples that illustrate how rich forms of abstractions can be captured.

We then extend these languages with linear logic connectives and obtain first the specification language Lolli and then Forum. After developing the theory of these languages, we present numerous extended examples of how to specify computations in them. These examples will illustrate how various concurrency notions can be captured. We shall illustrate the strength of this formalism by specifying and reasoning about the following: operational semantics of imperative programming languages; sequent calculus and natural deduction proof systems for an intuitionistic object-logic; and process calculi.

No advanced knowledge of the sequent calculus or of linear logic will be assumed, although some familiarity with their elementary syntactic properties will be useful. Similarly, some acquaintance with the basic concepts of the lambda-calculus and intuitionistic logic will also be useful.

1 Overview and Motivation

1.1 Roles for logic in the specification of computations

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 Logic [HM85] 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

$$x \text{ has value } n \multimap x \text{ has value } n+1$$

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 [Gir87] 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.

1.2 Desiderata for declarative programming

The development and maintenance of quality software is a serious and difficult challenge. All too often, software is developed just to run efficiently and without too many serious errors. All other demands that are placed on software — that it evolves with changing requirements, that it works on various different hardware architectures, memory hierarchies, and software environments, that parts of it can be reused, proved formally correct, and read by others — are seldom treated as goals. If these additional goals are addressed, they are not addressed with formal and rigorous techniques that have made other branches of engineering so successful. While the problems of program development and correctness must be addressed at many levels, including, for example, various human, managerial, and process factors, formal approaches can play important roles.

The field of mathematical logic has a long tradition of dealing with formal languages and with providing deep ways to reason about logical specifications on multiple levels. For example, soundness and completeness theorems show that two remarkably different approaches to the meaning of specifications (via proofs and models) actually coincide. Similarly, many different proof styles are known to be equivalent and such equivalences provide for deep understanding of inference and computation. Logic also provides rich collections of transformations on logical specifications, as can be easily seen by looking inside modern theorem provers.

If software could be given such multiple-levels of meaning, instead of being described in terms of compilers and interpreters, then one might expect rich transformations and rich formal analyses on programs that would make software much more flexible and malleable. A premise behind the work in *declarative programming languages* is that if programming can be based on logically, well understood formalisms, such as, say, the λ-calculus or intuitionistic logic, then programs expressed in these paradigms inherit the attributes of those formalisms, and, as a result, should admit many different kinds of semantic interpretations, for example, denotational, model-theoretical, and proof-theoretic. Broad avenues for reasoning and transforming the descriptions of computations (namely, programs) should result.

Given the central position that logic can play and has played in specification and programming languages, advances in our understanding of logic should ripple throughout the world of software. Given the recent advances in linear logic, logical frameworks, higher-order logic, and type theories, this is an exciting time to be working on declarative languages.

1.3 Relating algorithm and logic

In an early paper on logic programming, Bob Kowalski wrote the follow equation relating computation and logic.

$$\text{Algorithm} = \text{Logic} + \text{Control}.$$

His point with this equation is that Logic did not contain enough information to describe algorithms fully: what was missing, in his opinion, was control. This does summarize well the early experience using Horn clauses for the specification of computation. When realizing Horn clause in an actual computer system such as Prolog, explicit information about how goals were attempted and how program clauses were selected needed to be explicitly supplied, and this extra information was not available via logic.

This equation makes the important point that there is a gap between logic specifications and algorithmic specifications. This equation, however, seems to have been greatly elaborated in the intervening years to look sometime more like the following.

$$\begin{aligned}\text{Programming} = \text{Logic} \; &+ \; \text{Control} + \text{I/O} + \\ &+ \; \text{Higher-order programming} \\ &+ \; \text{Data abstractions} \\ &+ \; \text{Modules} \\ &+ \; \text{Concurrency} + \ldots\end{aligned}$$

That is, in order to get logic programming systems to encompass a range of features common to modern programming languages, many more extension to logic needed to be considered. Generally these extensions were made in *ad hoc* fashions: logic, which was the motivation and the intriguing start, is now put in a minor ghetto of the entire system. Questions about how various features interact start to dominate the language design, and analysis, which when done on purely logical expressions was deep and rich, was severely restricted or impossible.

An important activity of those working on logic and the specification of programming language is to try to achieve the equation:

$$\text{Programming} = \text{Logic}.$$

If this equation is at all possible, then one will certainly need to rethink what is meant by "Programming" and by "Logic". Clearly, the classical first-order theory of Horn clauses is far too weak for "Logic". There is also considerable room to rethink the notion of "Programming". If by programming we only mean the process of writing code that runs fast of a given hardware and software architecture and without too many series bugs, then logic may never be useful for programming: logic would introduce an overhead into the process of building programs that might not be recovered. If, however, we think of programming as the construction of artifacts (programs) that should evolve as new demands are made of them, work on a range of hardware and software architectures, be richly analyzed and transformed by compilers, and be verified as formally correct, then logic should have an enormous role to play in the design and analysis of programs.

In these notes we shall approach this goal by exploring the following equation.

$$\text{Programming} = \text{Higher-order linear Logic} + \ldots.$$

That is, we choose to use an expressive logic, that of higher-order linear logic, for making specifications. It will be clear that this logic will allow natural descriptions of programming features such as higher-order programming, modular programming, abstract data types, and concurrency. This equation is open since this approach does not capture everything that one would desire of a high-level programming language. Possibly future advances in logic will yield logics still more expressive and useful for specifying computations.

2 First-Order Logic

As we shall see by the end of these lectures, linear logic serves a single framework in which much of what we hope to do with proof search can be directly explained. We shall, however, chose to first work with more traditional logics, namely classical, intuitionistic, and minimal logics. It will be easy later to take what is learned from these logics and move them into the linear logic framework. Furthermore, we shall focus almost entirely on first-order logics, allowing quantification of predicates occasionally in examples, For more on higher-order quantification in logic programming, see [NM90, MNPS91].

2.1 Types and signatures

Let S be a fixed, finite set of *primitive types* (also called *sorts*). We assume that the symbol o is always a member of S. Following Church [Chu40], o is the type for propositions. The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of function types, denoted by the binary, infix symbol $\rightarrow$. The Greek letters τ and σ are used as syntactic variables ranging over types. The type constructor $\rightarrow$ associates to the right: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

Let τ be the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $\tau_0 \in S$ and $n \geq 0$. (By convention, if $n = 0$ then τ is simply the type τ_0.) The types $\tau_1, \ldots, \tau_n$ are the *argument types of* τ while the type τ_0 is the *target type of* τ. The order of a type τ is defined as follows: If $\tau \in S$ then τ has order 0; otherwise, the order of τ is one greater than the maximum order of the argument types of τ. Thus if $\mathrm{ord}(\tau)$ denotes the order of type expression τ then the following two equations define ord.

$$\begin{array}{rcl} \mathrm{ord}(\tau) & = & 0 \quad \text{provided } \tau \in S \\ \mathrm{ord}(\tau_1 \rightarrow \tau_2) & = & \max(\mathrm{ord}(\tau_1) + 1, \mathrm{ord}(\tau_2)) \end{array}$$

Thus, τ has order 1 exactly when τ is of the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $n \geq 1$ and $\{\tau_0, \tau_1, \ldots, \tau_n\} \subseteq S$. We say, however, that τ *is a first-order type* if the order of τ is either 0 or 1 and that no argument type of τ is o. The target type of a first-order type may be o.

For each type τ, we assume that there are denumerably many constants and variables of that type. Constants and variables do not overlap and if two constants (or variables) have different types, they are different constants (or variables). A *signature (over S)* is a finite set Σ of constants. We often enumerate signatures by listing their members as pairs, written $a\colon \tau$, where a is a constant of type τ. Although attaching a type in this way is redundant, it makes reading signatures easier. A signature is *first-order* if all its constants are of first-order type.

2.2 First-order terms and formulas

[Mention the priority in parsing formulas.]

We can now define the first-order logic $\mathcal{F}$. The *logical constants* of $\mathcal{F}$ are the symbols $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), *true* (truth), *false* (absurdity), and for every $\tau \in S - \{o\}$, $\forall_\tau$ (universal quantification over type τ), and $\exists_\tau$ (existential quantification over type τ). Thus, $\mathcal{F}$ has only a finite number of logical constants. The negation of a formula B is written as $B \supset \mathit{false}$.

Let τ be a type of the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where τ_0 is a primitive type and $n \geq 0$. If τ_0 is o, a constant of type τ is a *predicate constant of arity* n. If τ_0 is not o, then a constant of type τ is either an *individual constant* if $n = 0$ or a *function constant of arity* n if $n \geq 1$. Similarly, we can define *predicate variable of arity* n, *individual variable*, and *function variable of arity* n.

Let τ be a primitive type different from o. A *first-order term of type* τ is either a constant or variable of type τ, or of the form $(f\ t_1 \ldots t_n)$ where f is a function constant of type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ and, for $i = 1, \ldots, n$, t_i is a term of type τ_i. In the latter case, f is the *head* and $t_1, \ldots, t_n$ are the *arguments* of this term.

A first-order formula is either *atomic* or *non-atomic*. An atomic formula is of the form $(p\, t_1 \ldots t_n)$, where $n \geq 0$, p is a predicate constant of the first-order type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow o$, and $t_1, \ldots, t_n$ are first-order terms of the types $\tau_1, \ldots, \tau_n$, respectively. The predicate constant p is the *head* of this atomic formula. Non-atomic formulas are of the form *true*, *false*, $B_1 \wedge B_2$, $B_1 \vee B_2$, $B_1 \supset B_2$, $\forall_\tau x\ B$, or $\exists_\tau x\ B$, where B, B_1, and B_2 are formulas and τ is a primitive type different from o. The usual notions of free and bound variables and of open and closed terms and formulas are assumed.

The boldface letters t, s range over terms; the roman letters B, C range over formulas; A ranges over atomic formulas; and the Greek letters Γ, Δ range over sets of formulas.

Let s be a first-order term of type τ and let x be a variable of type τ. The operation of *substituting* s *for free occurrences of* x is written as $[s/x]$. Bound variables are assumed to be changed in a systematic fashion in order to avoid variable capture. Simultaneous substitution is written as the operator $[s_1/x_1, \ldots, s_n/x_n]$.

Let Σ be a first-order signature. A Σ-*term* is a closed term all of whose constants are members of Σ. Likewise, a Σ-*formula* is a closed formula all of whose nonlogical constants are members of Σ.

We define *clausal order* of formulas using the following recursion on first-order formulas.

$$\begin{array}{rcl}
\text{clausal}(A) & = & 0 \quad \text{provided } A \text{ is atomic, } \mathit{true}\text{, or } \mathit{false} \\
\text{clausal}(B_1 \wedge B_2) & = & \max(\text{clausal}(B_1), \text{clausal}(B_2)) \\
\text{clausal}(B_1 \vee B_2) & = & \max(\text{clausal}(B_1), \text{clausal}(B_2)) \\
\text{clausal}(B_1 \supset B_2) & = & \max(\text{clausal}(B_1) + 1, \text{clausal}(B_2)) \\
\text{clausal}(\forall x.B) & = & \text{clausal}(B) \\
\text{clausal}(\exists x.B) & = & \text{clausal}(B)
\end{array}$$

This measurement on formulas will be of particular interest when we design various logic programming languages.

The *polarity* of a subformula occurrence within a formula is defined as follows. If a subformula C of B occurs to the left of an even number of occurrences of implications in a B, then C is a *positive* subformula occurrence of B. On the other hand, if a subformula C occurs to the left of an odd number of occurrences of implication in a formula B, then C is a *negative* subformula occurrence of B. More formally:

- B is a positive subformula occurrence of B.
- If C is a positive subformula occurrence of B then C is a positive subformula occurrence in $B \wedge B'$, $B' \wedge B$, $B \vee B'$, $B' \vee B$, $B' \supset B$, $\forall_\tau x.B$, and $\exists_\tau x.B$; C is also a negative subformula occurrence in $B \supset B'$.
- If C is a negative subformula occurrence of B then C is a negative subformula occurrence in $B \wedge B'$, $B' \wedge B$, $B \vee B'$, $B' \vee B$, $B' \supset B$, $\forall_\tau x.B$, and $\exists_\tau x.B$; C is also a positive subformula occurrence in $B \supset B'$.

2.3 Sequent calculus

The sequent calculus provides a setting to reason about logical truth by considering the form of deduction. In this approach to proof, there are no axioms, only inference rules. In fact, for most logical connective there are two inferences, one describing the form of an argument in which one proves a formula with a given connective and one describing the form of an argument in which one reasons from such a formula. There are also structural rules, one of which is the cut rule: this rule can be used to justify that the two introduction rules are really describing dual aspects of the same logical connective.

Provability for $\mathcal{F}$ is given using sequent calculus proofs [Gen69]. 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) multisets of Σ-formulas. The multiset Γ is this sequent's *antecedent* and Δ is its *succedent*. The expressions Γ, B and B, Γ denote the multiset union $\Gamma \cup \{B\}$. The rules for introducing the logical connectives are presented in Figure 1, the initial and cut rules are given in Figure 2, and the structural rules are given in Figure 3. The following provisos are also attached to the four inference rules for quantifier introduction: in $\forall$R and $\exists$L, the constant c is not in Σ, and, in $\forall$L and $\exists$R, 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 $\Sigma : \Gamma \longrightarrow \Theta$.

2.4 Classical, intuitionistic, and minimal logics

Any proof is also called a **C**-*proof*. Any **C**-proof in which the succedent of every sequent in it has at most one formula is also called an **I**-*proof*. Furthermore, an **I**-proof in which the succedent of every sequent in it has exactly one formula 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 $\Sigma; \Gamma \vdash_C B$.

Notice that in an **I**-proof, there will be no occurrences of contrR while in an **M**-proof, there will be no occurrences of contrR and of weakR.

The notion of provability defined here is not equivalent to the more usual presentations of classical, intuitionistic, and minimal logic [Fit69, Gen69, Pra65, Tro73] in which signatures are not made explicit and substitution terms (the terms used in $\forall$L and $\exists$R) are not constrained to be taken from such signatures. The main reason they are not equivalent is illustrated by the following example. Let S be the set $\{i, o\}$ and consider the sequent

$$\{p : i \to o\} \ : \ \forall_i x\,(px) \ \longrightarrow \exists_i x\,(px).$$

This sequent has no proof even though $\exists_i x\,(px)$ follows from $\forall_i x\,(px)$ in the traditional presentations of classical, intuitionistic, and minimal logics. The reason for this difference is that there are no $\{p : i \to o\}$-terms of type i: that is, the type i is *empty* in this signature. Thus we need an additional definition: the signature Σ *inhabits* the set of primitive types S if for every $\tau \in S$ different than o, there is a Σ-term of type τ. When Σ inhabits S, the notions of provability defined above coincide with the more traditional presentations.

$$\frac{\Sigma\ :\ B, \Delta\ \longrightarrow \Gamma}{\Sigma\ :\ B \wedge C, \Delta\ \longrightarrow \Gamma}\ \wedge\mathrm{L} \qquad \frac{\Sigma\ :\ C, \Delta\ \longrightarrow \Gamma}{\Sigma\ :\ B \wedge C, \Delta\ \longrightarrow \Gamma}\ \wedge\mathrm{L}$$

$$\frac{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B \qquad \Sigma\ :\ \Delta\ \longrightarrow \Gamma, C}{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B \wedge C}\ \wedge\mathrm{R}$$

$$\frac{\Sigma\ :\ B, \Delta\ \longrightarrow \Gamma \qquad \Sigma\ :\ C, \Delta\ \longrightarrow \Gamma}{\Sigma\ :\ B \vee C, \Delta\ \longrightarrow \Gamma}\ \vee\mathrm{L}$$

$$\frac{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B}{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B \vee C}\ \vee\mathrm{R} \qquad \frac{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, C}{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B \vee C}\ \vee\mathrm{R}$$

$$\frac{\Sigma\ :\ \Delta_1\ \longrightarrow \Gamma_1, B \qquad \Sigma\ :\ C, \Delta_2\ \longrightarrow \Gamma_2}{\Sigma\ :\ B \supset C, \Delta_1, \Delta_2\ \longrightarrow \Gamma_1, \Gamma_2}\ \supset\mathrm{L} \qquad \frac{\Sigma\ :\ B, \Delta\ \longrightarrow \Gamma, C}{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B \supset C}\ \supset\mathrm{R}$$

$$\frac{\Sigma\ :\ \Delta, B[t/x]\ \longrightarrow \Gamma}{\Sigma\ :\ \Delta, \forall_\tau x\ B\ \longrightarrow \Gamma}\ \forall\mathrm{L} \qquad \frac{\Sigma \cup \{c{:}\,\tau\}\ :\ \Delta\ \longrightarrow \Gamma, B[c/x]}{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, \forall_\tau x\ B}\ \forall\mathrm{R}$$

$$\frac{\Sigma \cup \{c{:}\,\tau\}\ :\ \Delta, B[c/x]\ \longrightarrow \Gamma}{\Sigma\ :\ \Delta, \exists_\tau x\ B\ \longrightarrow \Gamma}\ \exists\mathrm{L} \qquad \frac{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B[t/x]}{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, \exists_\tau x\ B}\ \exists\mathrm{R}$$

$$\frac{}{\Sigma\ :\ \longrightarrow \mathit{true}}\ \mathit{true}\mathrm{R} \qquad \frac{}{\Sigma\ :\ \mathit{false}\ \longrightarrow}\ \mathit{false}\mathrm{L}$$

Figure 1: Introduction rules.

$$\frac{}{\Sigma\ :\ B\ \longrightarrow B}\ \mathit{initial} \qquad \frac{\Sigma\ :\ \Delta_1\ \longrightarrow \Gamma_1, B \qquad \Sigma\ :\ B, \Delta_2\ \longrightarrow \Gamma_2}{\Sigma\ :\ \Delta_1, \Delta_2\ \longrightarrow \Gamma_1, \Gamma_2}\ \mathrm{cut}$$

Figure 2: Initial and cut rules.

$$\frac{\Sigma\ :\ \Delta\ \longrightarrow \Gamma}{\Sigma\ :\ \Delta, B\ \longrightarrow \Gamma}\ \mathrm{weakL} \qquad \frac{\Sigma\ :\ \Delta\ \longrightarrow \Gamma}{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B}\ \mathrm{weakR}$$

$$\frac{\Sigma\ :\ \Delta, B, B\ \longrightarrow \Gamma}{\Sigma\ :\ \Delta, B\ \longrightarrow \Gamma}\ \mathrm{contrL} \qquad \frac{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B, B}{\Sigma\ :\ \Delta\ \longrightarrow \Gamma, B}\ \mathrm{contrR}$$

Figure 3: Structural rules.

2.5 Permutations of inference rules

An important aspect of the structure of a sequent calculus proof system is the way in which inference rules permute or do not permute. Consider the following combination of inference rules.

$$\dfrac{\dfrac{\Sigma : \Delta, p, r \longrightarrow s, \Gamma \qquad \Sigma : \Delta, q, r \longrightarrow s, \Gamma}{\Sigma : \Delta, p \vee q, r \longrightarrow s, \Gamma}\ \vee L}{\Sigma : \Delta, p \vee q \longrightarrow r \supset s, \Gamma}\ \supset R$$

Here, implication is introduced on the right below a left introduction of a disjunction. This order of introduction can be switch, as we see in the following combination of inference rules.

$$\dfrac{\dfrac{\Sigma : \Delta, p, r \longrightarrow s, \Gamma}{\Sigma : \Delta, p \longrightarrow r \supset s, \Gamma}\ \supset R \qquad \dfrac{\Sigma : \Delta, q, r \longrightarrow s, \Gamma}{\Sigma : \Delta, q \longrightarrow r \supset s, \Gamma}\ \supset R}{\Sigma : \Delta, p \vee q \longrightarrow r \supset s, \Gamma}\ \vee L$$

Notice that in this latter proof, we need to have two occurrences of the right introduction of implication.

Sometimes inference rules can be permuted if additional structural rules are employed. Consider the following two inference rules.

$$\dfrac{\dfrac{\Sigma : \Delta_1, r \longrightarrow \Gamma_1, p \qquad \Sigma : \Delta_2, q \longrightarrow \Gamma_2, s}{\Sigma : \Delta_1, \Delta_2, p \supset q, r \longrightarrow \Gamma_1, \Gamma_2, s}\ \supset L}{\Sigma : \Delta_1, \Delta_2, p \supset q \longrightarrow \Gamma_1, \Gamma_2, r \supset s}\ \supset R$$

To switch the order of these two inference rules requires introduction some weakenings and a contraction.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma : \Delta_1, r \longrightarrow \Gamma_1, p}{\Sigma : \Delta_1, r \longrightarrow \Gamma_1, p, s}\ \text{weakR}}{\Sigma : \Delta_1 \longrightarrow \Gamma_1, p, r \supset s}\ \supset R \qquad \dfrac{\dfrac{\Sigma : \Delta_2, q \longrightarrow \Gamma_2, s}{\Sigma : \Delta_2, q, r \longrightarrow \Gamma_2, s}\ \text{weakL}}{\Sigma : \Delta_2, q \longrightarrow \Gamma_2, r \supset s}\ \supset R}{\Sigma : \Delta_1, \Delta_2, p \supset q \longrightarrow \Gamma_1, \Gamma_2, r \supset s, r \supset s}\ \supset L}{\Sigma : \Delta_1, \Delta_2, p \supset q \longrightarrow \Gamma_1, \Gamma_2, r \supset s}\ \text{contrR}$$

Notice that if the first collection of inference rules was from an **I**-proof, then Γ_1 and Γ_2 must be empty. However, the result of permuting these inference rule would necessarily be a **C**-proof since we are required to have a sequent with two copies of $r \supset s$ on the right. In general, an $\supset$R below an $\supset$L in an **I**-proof cannot always be permuted.

There rules for the binary logical connectives that have two premises can be classified as either *additive* or *multiplicative*. A rule is additive if its the context surrounding the logical formula introduced in the conclusion is the same as the context surrounding its immediate subformulas in the premise sequents. The $\vee$L and $\wedge$R rules are additive. A rule is multiplicative if the context surrounding the the logical formula introduced in the conclusion is the accumulation of as the contexts surrounding its immediate subformulas in the premise sequents. The $\supset$L rule is multiplicative. Later when we discuss linear logic in more detail, these two styles of inference rules become particularly important. Linear logic also adds an *exponential* that relates these two. As we show in an exercise below, in the presence of weakening and contraction, it is possible to move between these two styles of inference rules without changing the logic.

2.6 Cut-elimination

The main theorem concerning sequent calculus proofs is called the cut-elimination theorem.

Theorem 1 *If a sequent has a* **C***-proof (respectively,* **I***-proof and* **M***-proof) then it has a cut-free* **C***-proof (respectively,* **I***-proof and* **M***-proof).*

Proofs of this theorem can be found in various places. Gentzen's original proof [Gen69] is still quite readable. See also [Gal86, GTL89]. Constructive proofs can be given and these result in procedures that can take a proof and systematically remove cut rules.

2.7 Consequences of cut-elimination

There are many important consequences of the cut-elimination theorem for these first-order logics. We list two here.

First, it is easy to show that these logics are consistent. That is, it is easy to see that there can be no proof of *false*: the only inference rules that could be used to prove the sequent $\Sigma : \longrightarrow \mathit{false}$ would be contrR and weakR, and these do not lead to a proof.

Second, all the sequents in a cut-free proof of $\Sigma : \Delta \longrightarrow \Gamma$ contain formulas that are subformulas of a formula in Δ or in Γ. This is the so-called *subformula property.* (By subformula, we also need to admit substitution instances of subformulas).

Finally, while cuts can be eliminated from proofs, this is largely only a statement of principle: there are few *mathematically interesting* sequents that have cut-free proofs that could be written down or stored in computer memories. While it is possible to formulate domains of mathematical interest within first-order logic, the most natural proofs of sequents in such domains would involve extensive use of cut, since this is the inference rule that encompasses the use of lemmas. The cut-elimination theorem claims that, in principle, lemmas are not needed: every theorem can be proved by organizing its subformulas into a proof. Clearly, proofs where where every use of a lemma is expanded out must be a huge object. It is not hard to check the complexity of the cut-elimination procedure to see that it can produce super-exponential blow-ups in proof size. Simple sequents with proofs involving a dozen cuts have cut-free proofs that require more elementary particles believed to comprise the universe.

Then to who could cut-free proofs be of interest. Logicians use the cut-elimination theorem to tell them that certain deep symmetries exist in their sequent systems. As we shall see in these notes, cut-free proofs have a useful role in describing computation: a cut-free proof can provide an elegant and flexible notion of *computation trace.* That is, here we shall think of cut-free proofs as recording the many minuscule steps of a computation: that is, a cut-free proof is rather similar to writing down every step that a Turing machine takes during some computation. Clearly, such proofs are not of particular use in the expression of mathematically interesting proofs. They will serve us, however, as a convenient device for representing and reasoning about computation.

2.8 Additional readings

In [Kle52], Kleene presents a detailed analysis of permutability of inference rules for classical and intuitionistic sequent systems similar to those presented here.

In [Mil91], Miller argues that proof theory should be considered a rich and appropriate setting for justifying declarative programming. The literature on logic programming more generally draws its justifications from model theory considerations.

2.9 Exercises

1. Provide proofs for each of the following sequents. Provide a **I**-proof only if there is no **M**-proof, and supply a **C**-proof only if there is no **I**-proof. Assume that the signature Σ is

$\{p : o, q : o, r : i \to o, a : i, b : i\}$.

(a) $p \wedge (p \supset q) \wedge (p \wedge q \supset s) \supset s$

(b) $(p \supset q) \supset (\neg q \supset \neg p)$

(c) $(\neg q \supset \neg p) \supset (p \supset q)$

(d) $p \vee (p \supset q)$

(e) $(r\ a \wedge r\ b \supset q) \supset \exists x (r\ x \supset q)$

(f) $((p \supset q) \supset p) \supset p$

(g) $\exists y \forall x (r\ x \supset r\ y)$

2. The multiplicative version of $\wedge$R would be the inference rule

$$\frac{\Sigma : \Delta_1, \longrightarrow B, \Gamma_1 \quad \Sigma : \Delta_2 \longrightarrow C, \Gamma_2}{\Sigma : \Delta_1, \Delta_2 \longrightarrow B \wedge C, \Gamma_1, \Gamma_2}.$$

Show that a sequent is has an **C**-proof (resp. **I**-proof, **M**-proof) if and only if it has one in a proof system that results from replacing $\wedge$R with the multiplicative version. Show the same but where $\vee$L is replaced with its multiplicative version

$$\frac{\Sigma : B, \Delta_1, \longrightarrow \Gamma_1 \quad \Sigma : C, \Delta_2 \longrightarrow \Gamma_2}{\Sigma : B \vee C, \Delta_1, \Delta_2 \longrightarrow \Gamma_1, \Gamma_2}.$$

[Notice that the multiplicative form of this rule can not be used with M-proofs. Fix this.]

3. Define a sequent proof to be *atomically closed* if every instance of the initial inference rule involves only atomic formulas. Show that a sequent has a **C**-proof (respectively, **I**-proof and **M**-proof) if and only if it has an atomically closed **C**-proof (respectively, **I**-proof and **M**-proof). [Notice that this is not possible with false in M-proofs. Fix this.]

4. Let $n \geq 1$ and let $\Sigma : \Delta \longrightarrow \Gamma$ be a sequent such that every formula in Δ is of order n or less and every formula in Γ is order $n-1$ or less. Prove that every sequent in a cut-free proof of $\Sigma : \Delta \longrightarrow \Gamma$ has this same property.

5. Show that if we consider **C**-proofs, then all inference rules for propositional connectives (exclude the quantifiers) permute over each other.

6. Not all pairs of quantification introduction rules permute. Present those pairs of inference rules that do not permute.

3 Logic Programming Considered Abstractly

3.1 Problems with proof search

In order to specify computation in a logic programming language, a programmer will first specify a signature, say Σ, that contains typing declarations for the set of *non-logical constants* about which a computation will involve. These constants are then used to build a formulas that are used for two different purposes. A *logic program* is a multiset of Σ-formulas that specifies, at least partially, the meaning of the constants in Σ. A *query* or *goal* is also a Σ-formula that serves as a question to ask of a logic program: goals are used to explore consequences of the specifications given by programs. More generally, a goal can also be a multiset of formulas (this is particularly important in the linear logic setting).

Computation is then the process of attempting to prove that a given goal follows from a given logic program. If this proof attempt is unsuccessful, the result of the computation is simply an indication that there was such a failure. If the attempt was successful, the resulting proof could also be returned: however, since proofs in this setting are essentially traces of entire computations, some kind of extraction from this proof is more reasonable to returned. This extract is will be a substitution for some of the variables found in the goal formula(s). This extract is often called an *answer substitution*.

Given a sequent, there are potentially many directions to explore to build a proof. We list some below.

1. It is always possible to use the cut rule. In that case, we need to produce a lemma to be proved on one branch and to be used on the other (also called the *cut-formula*).

2. The structural rules of contractions can always be applied to make additional copies of a rule.

3. The structural rule of weakening can be used to remove any formula from a sequent.

4. We could apply a left introduction rule or a right introduction rule.

5. We can check to see if the sequent is initial.

Some of these choices produce sub-choices. For example, choosing the cut rule requires finding a cut-formula; choosing $\forall$L or $\exists$R requires knowing a term t to instantiate a quantifier, and using the $\supset$L or cut rules require splitting the multisets Γ and Δ into two pairs of multisets.

All this freedom in searching for proofs is not, however, needed, and greatly reduced sets of choices can still result in complete proof procedures. We can deal with many of these choices as follows.

- Given the cut-elimination proof, we do not need to consider the cut rule and the problem of selecting a cut-formula. Such a choice forces us to move into a domain where proofs are more like computation traces than witnesses of mathematical truths. But since our goal here is the specification of computation, we shall live inside this choice.

- Since we have the structural rules of contraction and weakening, several simplifications can be made. First, we can assume that weakening is delayed until just prior to choosing the initial rule. Also, instead of splitting the context in the $\supset$L rule, we can apply contraction to duplicate all the formulas and then place one copy on the left branch and one copy on the right branch.

- The problem of determining an approach substitution term in the ∀L and ∃R rules is a serious problem whose solution fails outside our setting here. When systems based on proof search are implemented, they generally make use of various techniques, relying on the so-called "logic variable" and on unification. We shall not discuss these matters further here.

- The choice between selecting to do a right introduction or a left introduction can also be greatly simplified and is the central issue in *goal directed search*, describe more below.

3.2 Interpretation as goal-directed search

An idealized interpreter has three components in its state: a signature Σ, a set of Σ-formulas $\mathcal{P}$ denoting a program, and a Σ-formula G denoting the goal we wish to prove from $\mathcal{P}$. We use the *sequent* notation $\Sigma : \mathcal{P} \longrightarrow G$ to denote the *state* of this idealized interpreter.

If the interpreter is in state $\Sigma : \mathcal{P} \longrightarrow G$, how should proof search proceed? The principal restriction that we shall make in restricting the search for proofs is that that search should be *goal-directed*: that is, if the goal formula has a logical connective as its toplevel symbol, then the proof should be attempted by reducing that logical connective in a specific fashion. The particular rule depends on the logical connective. In particular, we would desire that our interpreter can make the following reductions.

AND Reduce $\Sigma : \mathcal{P} \longrightarrow B_1 \wedge B_2$ to the two sequents $\Sigma : \mathcal{P} \longrightarrow G_1$ and $\Sigma : \mathcal{P} \longrightarrow B_2$. Proofs of both sequents must now be attempted.

OR Reduce $\Sigma : \mathcal{P} \longrightarrow B_1 \vee B_2$ to either $\Sigma : \mathcal{P} \longrightarrow B_1$ or $\Sigma : \mathcal{P} \longrightarrow B_2$. A proof on only one of these needs will be sufficient.

INSTAN Reduce $\Sigma : \mathcal{P} \longrightarrow \exists_\tau x.B$ to $\Sigma : \mathcal{P} \longrightarrow B[t/x]$, for some Σ-term t of type τ.

AUGMENT Reduce $\Sigma : \mathcal{P} \longrightarrow B_1 \supset B_2$ to $\Sigma : \mathcal{P} \cup \{B_1\} \longrightarrow B_2$.

GENERIC Reduce $\Sigma : \mathcal{P} \longrightarrow \forall_\tau x.B$ to $\{c : \tau\} \cup \Sigma : \mathcal{P} \longrightarrow B[c/x]$, where c is a token that is not in the current signature Σ. We shall often refer to c as a "new constant".

TRUE The sequent $\Sigma : \mathcal{P} \longrightarrow true$ is provable immediately and does not need to be reduced further.

These reduction rules are goal-directed and do not consider either the signature or the logic program. Thus logical connectives get reflected into the search for proofs in a fixed fashion that cannot be modified by a program. For example, the connectives $\wedge$ and $\vee$ are always mapped into AND and OR search steps. Logic programs are responsible for determining the meaning of only the non-logical constants that are used to build atomic formulas.

If these reduction rules are reversed, the result can be seen as inference rules. For example, if the sequent $\Sigma : \mathcal{P} \longrightarrow B[t/x]$ can be proved for some Σ-term t of type τ, then we have a justification for the sequent $\Sigma : \mathcal{P} \longrightarrow \exists_\tau x.B$. Figure 4 displays the inference rules corresponding to these reduction rules.

Since we are attempting to justify the design of a *logic* programming language, it is natural to ask to what extent are these inference rules related to logic. It is easy to see that each of these inference rules is sound: that is, if the premise sequents (the sequents above the horizontal line in the inference rule) are true (in, say classical logic), then the original sequent is true. Soundness can be established without knowing the exact nature of signatures and programs.

$$\frac{\Sigma\ :\ \mathcal{P}\ \longrightarrow B_1 \qquad \Sigma\ :\ \mathcal{P}\ \longrightarrow B_2}{\Sigma\ :\ \mathcal{P}\ \longrightarrow B_1 \wedge B_2}\ \wedge\mathrm{R}$$

$$\frac{\Sigma\ :\ \mathcal{P}\ \longrightarrow B_1}{\Sigma\ :\ \mathcal{P}\ \longrightarrow B_1 \vee B_2}\ \vee\mathrm{R} \qquad \frac{\Sigma\ :\ \mathcal{P}\ \longrightarrow B_2}{\Sigma\ :\ \mathcal{P}\ \longrightarrow B_1 \vee B_2}\ \vee\mathrm{R}$$

$$\frac{\Sigma\ :\ \mathcal{P}\ \longrightarrow B[t/x]}{\Sigma\ :\ \mathcal{P}\ \longrightarrow \exists_\tau x.B}\ \exists\mathrm{R} \qquad \frac{}{\Sigma\ :\ \mathcal{P}\ \longrightarrow \mathit{true}}\ \mathit{true}\mathrm{R}$$

$$\frac{\Sigma\ :\ \mathcal{P} \cup \{B_1\}\ \longrightarrow B_2}{\Sigma\ :\ \mathcal{P}\ \longrightarrow B_1 \supset B_2}\ \supset\mathrm{R} \qquad \frac{\{c:\tau\} \cup \Sigma\ :\ \mathcal{P}\ \longrightarrow B[c/x]}{\Sigma\ :\ \mathcal{P}\ \longrightarrow \forall_\tau x.B}\ \forall\mathrm{R}$$

Figure 4: Inference rules for provability. The rule for universal quantification has the proviso that c is not declared in Σ and the rule for existential quantification has the proviso that t is a Σ-term of type τ.

The converse property, that of logical completeness of these rules, can be phrased as follows: if a sequent with a non-atomic goal is provable, are the sequents that it reduces to also provable? Achieving completeness is more involved and, in fact, dominates the design of the logic programming languages we considered. To see that these reductions are not generally complete, consider the following examples. Here, let signature Σ contain the declarations $\{p : o, q : o, r : i \to o, a : i, b : i\}$.

1. The OR rule reduces the sequent $\Sigma\ :\ p \vee q\ \longrightarrow q \vee p$ to either $\Sigma\ :\ p \vee q\ \longrightarrow q$ or $\Sigma\ :\ p \vee q\ \longrightarrow p$. Neither of these sequents is true while the original sequent is true.

2. The OR rule reduces the sequent $\Sigma\ :\ \emptyset\ \longrightarrow p \vee (p \supset q)$ to either $\Sigma\ :\ \emptyset\ \longrightarrow p$ or $\Sigma\ :\ \emptyset\ \longrightarrow p \supset q$. The first sequent is not provable and the second sequent would reduce to $\Sigma\ :\ p\ \longrightarrow q$, which is also not provable. It is easy to see, however, that $p \vee (p \supset q)$ is a classical logic tautology: if p is true, then the disjunction $p \vee (p \supset q)$ is true and if p is false, then $p \supset q$ is true and again the disjunction is true.

3. The INSTAN rule reduces the sequent

$$\Sigma\ :\ (ra \wedge rb) \supset q\ \longrightarrow \exists_i x(r\ x \supset q)$$

to the sequent $\Sigma\ :\ (r\ a \wedge r\ b) \supset q\ \longrightarrow r\ t \supset q$, where t is a Σ-term of type i. But there is not such term which makes this sequent provable. For example, if we used a for t, we would have the sequent

$$\Sigma\ :\ (ra \wedge rb) \supset q, r\ a\ \longrightarrow q$$

and this is no longer represents a true statement. To see that the original sequent is true classically, we know that $r\ a$ is either true or false. If it is false, then $\exists_i x(r\ x \supset q)$ is true (by picking a for x). If $r\ a$ is true, then $(ra \wedge rb) \supset q$ is equivalence to $rb \supset q$, so once again we have shown $\exists_i x(r\ x \supset q)$.

To achieve completeness for our logic programming languages, we shall need either to restrict the formulas allowed to be programs and goals (so as to avoid these counterexamples) or to chose our logic carefully. In fact, we shall take both steps. As the first example above illustrates, it seems likely that we will need to avoid having disjunctions in our programs. In fact, the formulas we eventually allow in programs will also be called *definite* formulas since they do not contain the indefinite information supplied by disjunctions. The last two examples illustrate that this step will not be enough since classical logic itself has built into it a disjunctive assumption, called the *excluded middle*: for every formula B, classical logic makes the formula $B \vee \neg B$ true. Intuitionistic logic will play an important role in analyzing logic programs.

We shall make the following definitions to help formalize the above observations. A cut-free **I**-proof is a *uniform proof* if for every sequent in it with a non-atomic succedent is the conclusion of a right introduction rule. Notice that in uniform proofs, if a sequent is the conclusion of an initial or left-introduction rule then the succedent of that sequent is an atomic formula. Let $\mathcal{D}$ and $\mathcal{G}$ be a set of formula denoting, respectively, the definite clauses (program clauses) and goal formulas of an intended logic programming. Let $\vdash$ be provability in some logic, such as classical or intuitionistic logic. The the triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ is an *abstract logic programming language* if and only if for every finite subset $\mathcal{P}$ of $\mathcal{D}$ and for every $G \in \mathcal{G}$, $\Sigma; \mathcal{P} \vdash G$ if and only if the sequent $\Sigma : \mathcal{P} \longrightarrow G$ has a uniform proof.

Since these two definitions are restricted to **I**-proofs, we shall refer to them as the *single-conclusion* version of uniform proofs and abstract logic programming. For more on these definitions, see [MNPS91, NM90]. We shall later introduce a multiple-conclusion version of these definitions.

The first (abstract) logic programming language we consider is first-order Horn clauses: these are weak enough that they do not separate classical from intuitionistic provability.

3.3 The syntax of first-order Horn clauses

There are several, roughly equivalent ways to describe first-order Horn clauses (*fohc* for short). We present three here. In making these definitions, we make use of three syntactic variables: A denotes atomic formulas, G denotes goal formulas, and D denotes program formulas (also called definite formulas). Programs formulas are also called *clauses*.

A common definition of Horn clauses (see, for example, [AvE82]) is given using the following grammar.

$$\begin{aligned} G &::= \quad A \mid G \wedge G \\ D &::= \; A \mid G \supset A \mid \forall_\tau x\, D. \end{aligned} \tag{1}$$

(Here and in the rest of this chapter, we assume that the type τ is a primitive type.) That is, goal formulas are conjunctions of atomic formulas and program clauses are of the form

$$\forall_{\tau_1} x_1 \ldots \forall_{\tau_1} x_m [A_1 \wedge \ldots \wedge A_n \supset A_0]$$

for $m, n \geq 0$. (If $m = 0$ then we do not write any universal quantifiers, and if $n = 0$ then we do not write the implication.)

A richer formulation is given by the following definition.

$$\begin{aligned} G &::= \; \mathit{true} \mid A \mid G \wedge G \mid G \vee G \mid \exists_\tau x\, G \\ D &::= \quad A \mid G \supset D \mid D \wedge D \mid \forall_\tau x\, D. \end{aligned} \tag{2}$$

Here, the connectives *true*, $\vee$, and $\exists$ are permitted in goals and $\wedge$ and $\forall$ can be mixed in definite formulas. Also, the "head" of a definite clause does not need to be immediately present at the top-level of a clause: it might be buried to the right of implications and conjunctions and under universal quantifiers.

A compact presentation of Horn clauses can be given simply as

$$\begin{aligned} G &::= \quad A \\ D &::= \quad A \mid A \supset D \mid \forall_\tau x\ D. \end{aligned} \tag{3}$$

Notice that in this definition, definite clauses are composed only of implications and universal quantifiers where the nesting of implications and universal quantifiers is allowed only in the conclusion of an implication and not in a premise.

It is the D-formulas that are considered Horn clauses. A Horn clause program is then a finite set of *closed* D-formulas. The symbol $\mathcal{P}$ will often be used as a syntactic variable to denote programs.

These three ways of defining program clauses give rise to programming languages of the same expressive power: that is, a program clause in one definition is classically equivalent (also intuitionistically equivalent) to a conjunction of program clauses in another definition. This is easily shown by using suitable applications of the following classical and intuitionistic equivalences.

$$\begin{aligned} \forall x(B_1 \wedge B_2) &\equiv (\forall x\ B_1) \wedge (\forall x\ B_2) \\ B_1 \supset (B_2 \supset B_3) &\equiv (B_1 \wedge B_2) \supset B_3 \\ B_1 \wedge (B_2 \vee B_3) &\equiv (B_1 \wedge B_2) \vee (B_1 \wedge B_3) \\ B_1 \vee (B_2 \wedge B_3) &\equiv (B_1 \vee B_2) \wedge (B_1 \vee B_3) \\ (B_1 \vee B_2) \supset B_3 &\equiv (B_1 \supset B_3) \wedge (B_2 \supset B_3) \\ B_1 \supset (B_2 \wedge B_3) &\equiv (B_1 \supset B_2) \wedge (B_1 \supset B_3) \\ B_1 \supset (\forall x\ B_2) &\equiv \forall x\ (B_1 \supset B_2) \\ (\exists x\ B_2) \supset B_1 &\equiv \forall x\ (B_2 \supset B_1) \end{aligned}$$

In the last two equivalences, x is not free in B_1.

If we speak about first-order Horn clauses without qualification, then we assume we are using the richest of these three definitions, namely (2).

3.4 Proof search with first-order Horn clauses

In the setting of classical logic, goal-directed search reductions presented in Subsection 3.2 is complete. To give a more complete picture of proof search within *fohc*, we need to describe how proof search deals with atomic goals. For example, if the goal G is the atomic formula A and the program $\mathcal{P}$ contains the formula A, then we clearly have a proof and computation (search) finishes immediately with a success. If $\mathcal{P}$ contains instead a clause of the form $G' \supset A$ then we know that if we can prove G' from $\mathcal{P}$, then we have again found a proof for A: since $G' \supset A$ and G' follow from $\mathcal{P}$, then so to does A. In this case, we have reduced the problem of proving $\Sigma : \mathcal{P} \longrightarrow A$ to proving $\Sigma : \mathcal{P} \longrightarrow G'$. Using a program clause in this manner to reduce the problem of proving an atomic formula is generally called *backchaining*.

To describe backchaining we use the additional inference rules found in Figure 5. To indicate that the interpreter is attempting to prove the atomic goal A by backchaining on the program clause D, we use the expression $\Sigma; \mathcal{P} \xrightarrow{D} A$. The first of the rules in Figure 5 specifies that when reducing the problem of finding a proof of $\Sigma : \mathcal{P} \longrightarrow A$, we need first to pick a member D of $\mathcal{P}$ and then attempt to backchain on it. The second rule in this figure states the obvious: if the formula that we are using for backchaining is the formula we are attempting to prove, then we are

$$\frac{\Sigma; \mathcal{P} \xrightarrow{D} A}{\Sigma : \mathcal{P} \longrightarrow A}\ \mathit{decide} \qquad\qquad \frac{}{\Sigma; \mathcal{P} \xrightarrow{A} A}\ \mathit{initial}$$

$$\frac{\Sigma; \mathcal{P} \xrightarrow{D_1} A}{\Sigma; \mathcal{P} \xrightarrow{D_1 \wedge D_2} A}\ \wedge\mathrm{L} \qquad\qquad \frac{\Sigma; \mathcal{P} \xrightarrow{D_2} A}{\Sigma; \mathcal{P} \xrightarrow{D_1 \wedge D_2} A}\ \wedge\mathrm{L}$$

$$\frac{\Sigma : \mathcal{P} \longrightarrow G \qquad \Sigma; \mathcal{P} \xrightarrow{D} A}{\Sigma; \mathcal{P} \xrightarrow{G \supset D} A}\ \supset\mathrm{L} \qquad\qquad \frac{\Sigma; \mathcal{P} \xrightarrow{D[t/x]} A}{\Sigma; \mathcal{P} \xrightarrow{\forall_\tau x.D} A}\ \forall\mathrm{L}$$

Figure 5: Rules for backchaining. In the decide rule, D is a member of $\mathcal{P}$, and in the $\forall$L rule, t is a Σ-term of type τ.

finished. If the formula selected for backchaining is a conjunction, then reduce this attempt to one using one of the conjuncts. If the backchain formula is universally quantified, then pick a Σ-term and continue backchaining with that instance of the formula. Finally, if the the backchain formula is an implication, say $G \supset D$, then we need to do two things: we must prove G and continue using D to do backchaining.

Combining the inference rules in Figure 4 and 5 now yields a complete proof system for *fohc* with respect to classical logic. A proof of this fact can be found in [NM90] (see also the exercises below). When read bottom-up, these inference rules provide a complete set of reduction steps for finding a proof.

Consider a proof of the sequent $\Sigma : \mathcal{P} \longrightarrow G$ using these inference rules. It is easy to see that every sequent that appears in such a proof will either be of the form $\Sigma : \mathcal{P} \longrightarrow G'$ or $\Sigma; \mathcal{P} \xrightarrow{D} A$, for some A, G', and D. Notice that all of these sequents contain the same signature and program. Thus, during the search for such a proof, all goals will be attempted from the same program and all the selection of program clauses will be from that same program. This one observation has at least two important consequences.

1. First, remember that changes to sequents during the construction of a proof are used to capture the dynamics of computations. If the only changes that can occur to a sequent is to the goal formula (which could be considered to be as simple as conjunctions of atoms), then the dynamics of computations must be modeled mostly *within* atomic formulas, that is, via changes in terms within atoms. Since most of the dynamics occurs in changes to atoms, logic will be able to play little direct role in reasoning about computation.

2. If we initiate a computation (that is, a search for a proof), all the program clauses that will ever be needed to complete the proof must be present in the initial sequent. Similarly, every constant, and hence the constructors for every data structure that might every need to be built, need to be present in the initial sequent. Thus, proof search using *fohc* provides no mechanisms for hiding code or data constructors. Signatures and programs are global, flat structures that do not change over the lifetime of a computation. Thus it will not be possible in *fohc* to have auxiliary programs available only when they are needed and it will not be possible to build data structures (terms) that only certain code will be allowed to access. If any code or term constructors are ever needed they must be available from the start on equal footing with all other code and data constructors. This lack of abstraction will be one of the motivations for going beyond *fohc*.

3.5 Additional readings

A background in using logic programming language, particularly Prolog, is assumed for much of these notes. To gain such a background, see the books [CM84, SS86]. A nice introduction to theory of first-order Horn clauses, see [AvE82].

3.6 Exercises

1. Show that each of the following is true using all three definitions of first-order Horn clauses.
   (a) A definite clause is order 0 or 1.
   (b) A goal formula is order 0.
   (c) If a subformula of a definite clause occurs positively, then it is a definite clause; if it occurs negatively, it is a goal formula.
   (d) All subformula occurrences of a goal formula occur positively and are goal formulas.

2. If we restrict to using Horn clauses given by the first definition (1), then program clauses are of the form
$$\forall_{\tau_1} x_1 \ldots \forall_{\tau_1} x_m [A_1 \wedge \ldots \wedge A_n \supset A_0]$$
where $m, n \geq 0$. Show that the backchaining rule can also be simplified to the following one rule.
$$\frac{\Sigma : \mathcal{P} \longrightarrow A_1\theta \qquad \cdots \qquad \Sigma : \mathcal{P} \longrightarrow A_n\theta}{\Sigma; \mathcal{P} \xrightarrow{D} A}$$
This rule has the proviso that D be a formula of the form above and θ is a substitution such that for all $i = 1, \ldots, m$ θ maps the variable x_i to the Σ-terms t_i of type τ_i, and that A is equal to $A_0\theta$.

3. With this exercise, we show that first-order Horn clauses (using definition 2) is an abstract logic programming language.
   (a) If $n \geq 0$ and $\Sigma : \mathcal{P} \longrightarrow G_1, \ldots, G_n$ has a **C**-proof, then there is an i such that $1 \leq i \leq n$ such that $\Sigma : \mathcal{P} \longrightarrow G_i$. (Hence, $n > 0$.)
   (b) Show that programs in *fohc* are consistent: that is, that it is not the case that both $\Sigma : \mathcal{P} \longrightarrow A$ and $\Sigma : \mathcal{P} \longrightarrow \neg A$ have **C**-proofs.
   (c) If $\Sigma : \mathcal{P} \longrightarrow G$ has a **C**-proof then it has an **I**-proof with no occurrences of the $\forall$R and $\supset$R inference rules.
   (d) If $\Sigma : \mathcal{P} \longrightarrow G$ has a **C**-proof then it has using the inference rules in Figures 4 and 5.
   (e) Conclude that the classical theory of *fohc* is an abstract logic programming language.

4. Let $\mathcal{P}$ be a set of Horn clauses and let D be a Horn clause. Show that if $\Sigma : \mathcal{P} \longrightarrow D$ has a **C**-proof then it has an **I**-proof.

5. For this exercise, consider only formulas built using only *true*, $\supset$ and $\wedge$. Assume that there is a formula B that is classically provable but not intuitionistically provable.
   (a) Show that B must have order at least 3.
   (b) Show that the smallest such formula (counting logical connectives) is Pierce's formula, namely $((p \supset q) \supset p) \supset p$.

4 Hereditary Harrop Formulas

4.1 Harrop formulas

In [Har60], Harrop studied a class of formulas that can be defined as follows. Let B be a syntactic variable for arbitrary first-order formulas and let H be defined by

$$H ::= A \mid B \supset H \mid \forall_\tau x\ H \mid H_1 \wedge H_2.$$

An H-formula is often called a *Harrop formula.* The main theorem regarding these formulas is that the six reduction rules mentioned in Subsection 3.2 are intuitionistically satisfied when they are applied to sequents of the form $\Sigma : \mathcal{H} \longrightarrow B$ where $\mathcal{H}$ is a finite collection of Harrop formulas and B is an arbitrary Harrop formula. Actually, what is mentioned explicitly in [Har60] correspond to the OR and INSTAN reductions: the other four reductions are simple to show. If a set of formulas $\mathcal{H}$ satisfy the OR and INSTAN reductions, those formulas are often said to satisfy, respectively, the *disjunctive* and *existential property.*

Harrop formulas do not, however, constitute an abstract logic programming language. Assume that the $\Sigma : \mathcal{H} \longrightarrow B$ has an intuitionistic proof and that B is not atomic. Given the above property for Harrop formulas, it is the case that if this sequent if probable then the last inference can be taken to be a right introduction. For example, the sequent $\Sigma : \mathcal{H} \longrightarrow B_1 \supset B_2$ would be proved from the sequent $\Sigma : \mathcal{H}, B_1 \longrightarrow B_2$ by $\supset$R. Notice, however, that since B_1 and B_2 can be arbitrary formulas, the antecedent of this new sequent is not necessarily a collection of Harrop formulas. As a result, we will not be able to guarantee that such reductions can hold at all sequents in a proof.

To fix this problem, we only need to arrange things so that whenever a formula, such as B_1, is added to the succedent, it is again a Harrop formulas.

4.2 Three presentations of *fohh*

The *first-order hereditary Harrop formulas* (*fohh*) extend Horn clauses by allowing implications and universal quantifiers in goals (and, thus, in the body of program clauses). Parallel to the three presentations of *fohc* in Section 3.3, there are the following three presentations of goals and program clauses for *fohh*. The first presentation is similar to that of definition 1 in Section 3.3.

$$\begin{aligned} G &::= A \mid G \wedge G \mid D \supset G \mid \forall_\tau x.G \\ D &::= A \mid G \supset A \mid \forall x.D \end{aligned} \tag{4}$$

Notice now that the definitions of G- and D-formulas are mutually recursive, that a negative (positive) subformula of a G-formula is a D-formula (G-formula), and that a negative (positive) subformula of a D-formula is a G-formula (D-formula). A richer formulation is given by the following definition.

$$\begin{aligned} G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G \mid D \supset G \mid \forall x.G \\ D &::= A \mid G \supset D \mid D \wedge D \mid \forall x.D \end{aligned} \tag{5}$$

It will be this set of richer D-formulas that we shall consider the proper definition of first-order hereditary Harrop formulas.

A simple presentation of a class of definite formulas similar to the one above is given by the definition

$$D ::= A \mid D \supset D \mid D \wedge D \mid \forall x.D \tag{6}$$

Any first-order formula that does not contain occurrences of disjunction and existential quantification is an example of both a D-formula and G-formula in the sense of definitions 5 and 6. The formula $(p \vee q) \supset (p \vee q)$ is neither a D-formula and G-formula in any of the definitions above.

Classical logic does not support a goal-directed search interpretation of logical connectives for any interesting uses of $\supset$ and $\forall$ in goal formulas. For example, the formula $p \vee (p \supset q)$ is a classical tautology but it is not provable using the search operations given above: p is not provable and q does not follow from p. Similarly, if the current program $\mathcal{P}$ contains the single formula $(p\ a \wedge p\ b) \supset q$ then the formula $\exists x.(p\ x \supset q)$ is a classical conclusion but it cannot be found using the search reductions described above.

The three presentations of *fohh* given above are not related using intuitionistic equivalence. First notice that the definite formulas of definition 5 strictly contain the definite formulas of definitions 4 and 6. In particular, the formula

$$(p \supset (q \vee r)) \supset s$$

is a legal definite clause using Definition 5, but it is not logically equivalent to a formula or conjunction of formulas using either 4 or 6. While it is the case that the displayed formula above does imply the conjunction

$$((p \supset q) \supset s) \wedge ((p \supset r) \supset s),$$

the converse is not true (although the converse is a classical logic entailment). As program clauses, however, these two formulas can be used interchangeably since they can be used to prove exactly the same goal formulas.

The existential quantifiers allowed in goals in Definition 5 cannot always be eliminated as was possible with *fohc*. In *fohc*, an existential quantifier in a goal can be given a larger scope until it can be converted to a universal quantifier surrounding a Horn clause. There are two ways that an existential quantifier in a goal can be "stuck" within a goal. First, it is possible for it to be to the right of an implication, as in the goal formula $D \supset \exists x\ G$. Even if x is not free in D, this formula is not intuitionistically equivalent to $\exists x(D \supset G)$. It is also possible for an existential quantifier to be inside the scope of a universal quantifier.

For example, consider the program clauses $\forall x.((\forall y \exists z.(q\ x\ y\ z)) \supset p\ x)$. The existential quantifier for z cannot be removed by simple logical equivalences of first-order logic. It is possible, however, to introduce a new predicate constant to obtain a program that proves the same goals (that do not involve the new predicate constants). In particular, the two clauses $\forall x.((\forall y.r\ x\ y) \supset p\ x)$ and $\forall x \forall y \forall z.(q\ x\ y\ z \supset r\ x\ y)$ can be used instead of the above clause.

4.3 The core of a logic programming language

Given the distinctions we have made between the program clauses and the goal formulas of a given logic programming language it is interesting to identify that class of formulas that can be in both classes. The *core* of a logic programming language is the intersection of its goal formulas and its program clauses. For example, using the definitions of logic programming based on first-order Horn clauses given in Section 3.3, the core of *fohc* is either the set of atomic formulas (using definitions (1) or (3)) or the set of conjunctions of atomic formulas (using definition (2)).

The core of *fohh* is, however, much richer. Using either definition (5) or (6), the core is the set of formulas built from atomic formulas using $\wedge$, $\supset$, and $\forall$: only $\vee$ and $\exists$ are excluded. The core of *fohh* coincides with the definition of program clauses given by (6). Notice, however, that first-order Horn clauses defined using either (1) or (3) are contained within the core of *fohh*.

Formulas in the core of logic programming language can be both proved and used as program clause. Since the core of *fohh* contains a rich set of formulas, it will sometimes be possible to use *fohh* to reason about programs directly. In so doing, the cut-elimination theorem can be used where the cut-formula comes from the core. For example, if we have **I**-proofs for both $\Sigma : \mathcal{P} \longrightarrow M$ and $\Sigma : \mathcal{P}, M \longrightarrow G$, where M is in the core, then the cut-elimination theorem tells us that there is an **I**-proof of $\Sigma : \mathcal{P} \longrightarrow G$. Here, cut-elimination tells us that if two computations exists, a third one exists. This can be a powerful tool for reasoning, especially when M has some interesting logical structure.

4.4 Proof search with first-order hereditary Harrop formulas

Proof search for *fohh* can be proved to be complete for the combination of rules taken from Figures 4 and 5. The different from proof search with *fohc* is that during the search for proofs, both the signature and antecedent (program) can increase. A universally quantified goal can be seen to add a new constant (called an *eigen-variable*) the signature and an implicational goal can be seen to add clauses to the program.

These features of *fohh* make possible logical support for modular programs and abstract datatypes. See [Mil89b, Mil89a, Mil90] for examples of such uses of *fohh*. This observation about *fohh* has spawned a lot of research into using various kinds of implications and modal operators to structure code in logic programs. For a survey of these papers, see [BLM94].

4.5 Multiple conclusion sequents and scope extrusion

A natural notion of scoping occurs in logic programming based on single-conclusion sequents. For example, the search for a uniform proof of the sequent $\Sigma : \mathcal{P} \longrightarrow D \supset G$ reduces to the search for a uniform proof of the sequent $\Sigma : \mathcal{P}, D \longrightarrow G$. If $\mathcal{P}$ is considered to be the current program held by a logic programming interpreter, then D can be seen as a program unit that is added to the current program during a computation. A notion of modular programming for logic programming was developed in [Mil89b] based on this simple observation. To enforce that this notion of modular programming obeys the correct notion of scoping, single conclusion sequent calculus is required. Consider, for example, searching for a uniform proof of the sequent $\Sigma : \mathcal{P} \longrightarrow G_1 \vee (D \supset G_2)$ using the usual intuitionistic introduction rules for $\vee$-R and $\supset$-R [Gen69]. This search would lead to the search for proofs of either the sequent $\Sigma : \mathcal{P} \longrightarrow G_1$ or $\Sigma : \mathcal{P}, D \longrightarrow G_2$. In particular, the formula D is only available to help prove the formula G_2: its scope does not include G_1. This formula is, however, classically equivalent to $(D \supset G_1) \vee G_2$ and $D \supset (G_1 \vee G_2)$. Thus the scope of D can move in ways not supported in intuitionistic logic. In particular, $p \vee (p \supset q)$ is not provable intuitionistically but it is classically. Gentzen's characterization of the differences between intuitionistic and classical logics as arising from differences in using single and multiple conclusion sequents provides an elegant analysis of scope extrusion. Consider the following sequent proof.

$$\dfrac{\dfrac{\dfrac{}{\Sigma : p \longrightarrow p, q}\ \mathit{initial}}{\Sigma : \longrightarrow p, p \supset q}\ \supset\mathrm{R}}{\Sigma : \longrightarrow p \vee (p \supset q)}\ \vee\mathrm{R}$$

The occurrence of p in the left of the initial sequent has as its scope all the formulas on the right: in the intuitionistic case, there can only be one such formula on the right and, hence, scope cannot be liberalized in this way.

For reason such as this, to achieve a notion of modular programming with a "proper" discipline for scoping, we need to limit ourselves to intuitionistic logic. Scope extrusion is, however, a feature of the π-calculus [MPW92a, MPW92b] and such extrusion has been described using multiple conclusion sequent calculus in [Mil93].

$$\frac{\Sigma' \,:\, \mathcal{P}' \longrightarrow B \qquad \Sigma \,:\, \mathcal{P}, B \longrightarrow C}{\Sigma' \,:\, \mathcal{P}' \longrightarrow C}\ cut$$

$$\frac{\Sigma + x:\tau \,:\, \mathcal{P} \longrightarrow B \qquad t \text{ is a } \Sigma'\text{-term of type } \tau}{\Sigma' \,:\, \mathcal{P}' \longrightarrow B[t/x]}\ subst$$

Figure 6: Cut and subst rules for $\mathcal{M}$. Here, $\Sigma \subseteq \Sigma'$ and $\mathcal{P} \subseteq \mathcal{P}'$.

4.6 A Kripke model semantics

Consider the first-order intuitionistic theory for the connectives *true*, $\wedge$, $\supset$, and $\forall_\tau$. We briefly describe a model theoretic semantics for this logic.

A *dependent pair* is a pair $\langle \Sigma, \mathcal{P} \rangle$ where Σ is a signature and $\mathcal{P}$ is a set of Σ-formulas. Define $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ whenever $\Sigma \subseteq \Sigma'$ and $\mathcal{P} \subseteq \mathcal{P}'$. A *Kripke model*, $[\mathcal{W}, I]$, is the specification of a set of *worlds* $\mathcal{W}$, which is a set of dependent pairs, and a function I, called an *interpretation*, that maps pairs in $\mathcal{W}$ to sets of atomic formulas. The mapping I must satisfy the two conditions:

1. $I(\langle \Sigma, \mathcal{P} \rangle)$ is a set of atomic Σ-formulas, and
2. for all $w, w' \in \mathcal{W}$ such that $w \preceq w'$, $I(w) \subseteq I(w')$ (that is, I is order preserving).

Satisfiability (also called *forcing*) in a Kripke model is defined as follows. Let $[\mathcal{W}, I]$ be a Kripke model, let $\langle \Sigma, \mathcal{P} \rangle \in \mathcal{W}$, and let B be a Σ-formula. The three place *satisfaction* relation $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$ is defined by induction on the structure of B.

- $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$ if B is atomic and $B \in I(\langle \Sigma, \mathcal{P} \rangle)$.
- $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B \wedge B'$ if $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$ and $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B'$.
- $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B \supset B'$ if for every $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ such that $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ and $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B$ then $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B'$.
- $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x.B$ if for every $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ such that $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ and for every Σ'-terms t of type τ, the relation $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B[t/x]$ holds.

The *signature of an interpretation* I is the largest signature that is contained in all worlds of the partial order underlying I. If Σ_0 is the signature of the interpretation I and B is a Σ_0-formula, then we write $I \Vdash B$ if $I, w \Vdash B$ for all $w \in \mathcal{W}$.

Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair. The *canonical model* for $\langle \Sigma, \mathcal{P} \rangle$ is defined as the model with the set of worlds $\{\langle \Sigma', \mathcal{P}' \rangle \mid \langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle\}$ and where I is defined so that $I(\langle \Sigma', \mathcal{P}' \rangle)$ is the set of all atomic formulas A so that $\Sigma'; \mathcal{P}' \vdash A$.

For the purposes of this section, we shall assume that there are two forms of cut rules for this sequent calculus: one works with the signature of the antecedent (called the *subst* rule) and one works with the formulas of the antecedent (called simply the *cut* rule). Both rules are displayed in Figure 6. The cut-elimination theorem for this logic is the fact that both of these rules can be eliminated from proofs.

Theorem 2 *Cut-elimination holds for $\mathcal{M}$ if and only if the following holds: for every dependent pair $\langle \Sigma, \mathcal{P} \rangle$ and every Σ-formula B, $\Sigma; \mathcal{P} \vdash B$ if and only if $I \Vdash B$, where I is the canonical model for $\langle \Sigma, \mathcal{P} \rangle$.*

Proof Assume first that cut-elimination holds for $\mathcal{M}$. We now prove by induction on the structure of B that $\Sigma; \mathcal{P} \vdash B$ if and only if $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B$.

Case: B is atomic. The equivalence is trivial.

Case: B is $B_1 \wedge B_2$. This case is simple and immediate.

Case: B is $B_1 \supset B_2$. Assume first that $\Sigma; \mathcal{P} \vdash B_1 \supset B_2$. By completeness of uniform proofs, $\Sigma; \mathcal{P} \cup \{B_1\} \vdash B_2$. To show $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B_1 \supset B_2$, let $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ be such that $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ and $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B_1$. By the inductive hypothesis, $\Sigma'; \mathcal{P}' \vdash B_1$ and by cut-elimination, $\Sigma'; \mathcal{P}' \vdash B_2$. By induction again, we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B_2$. Thus, $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B_1 \supset B_2$. For the converse, assume $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B_1 \supset B_2$. Since $\Sigma; \mathcal{P} \cup \{B_1\} \vdash B_1$, the inductive hypothesis yields $I, \langle \Sigma, \mathcal{P} \cup \{B_1\} \rangle \Vdash B_1$. By the definition of satisfaction of implication we must have $I, \langle \Sigma, \mathcal{P} \cup \{B_1\} \rangle \Vdash B_2$. But by the inductive hypothesis again, $\Sigma; \mathcal{P} \cup \{B_1\} \vdash B_2$, and $\Sigma; \mathcal{P} \vdash B_1 \supset B_2$.

Case: B is $\forall_\tau x B_1$. Assume first that $\Sigma; \mathcal{P} \vdash \forall_\tau x B_1$. By completeness of uniform proofs, $\Sigma \cup \{d\}; \mathcal{P} \vdash B_1[d/x]$ for any constant d not in Σ. To show $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x B_1$, let $\langle \Sigma', \mathcal{P}' \rangle \in \mathcal{W}$ be such that $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ and t is a Σ'-term of type τ. By cut-elimination on signatures (the subst rule), we have $\Sigma'; \mathcal{P}' \vdash B_1[t/x]$. By induction we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B_1[t/x]$. Thus, $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x B_1$. For the converse, assume $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x B_1$. Let d be a constant not a member of Σ. Since d is a $\Sigma \cup \{d\}$-term, $I, \langle \Sigma \cup \{d\}, \mathcal{P} \rangle \Vdash B_1[d/x]$ by the definition of satisfaction of universal quantification. But by the inductive hypothesis again, $\Sigma \cup \{d\}; \mathcal{P} \vdash B_1[d/x]$ and $\Sigma; \mathcal{P} \vdash \forall_\tau x B_1$.

Now assume the equivalence: for every dependent pair $\langle \Sigma, \mathcal{P} \rangle$ and every Σ-formula B, $\Sigma; \mathcal{P} \vdash B$ if and only if $I \Vdash B$, where I is the canonical model for $\langle \Sigma, \mathcal{P} \rangle$. We now show that any sequent that can be proved using occurrences of the cut and subst rules can be proved without such rules. In particular, we show that if $\langle \Sigma, \mathcal{P} \rangle \preceq \langle \Sigma', \mathcal{P}' \rangle$ then each of the following holds.

1. If $\Sigma'; \mathcal{P}' \vdash B$ and $\Sigma; \mathcal{P}, B \vdash C$ then $\Sigma'; \mathcal{P}' \vdash C$.
2. If t is a Σ'-term of type τ and $\Sigma + x : \tau; \mathcal{P} \vdash B$ then $\Sigma'; \mathcal{P}' \vdash B[t/x]$ (of course, x does not occur in Σ).

From these facts, any number of occurrences of the cut and subst rules can be eliminated from a proof containing them.

To prove (1), assume that $\Sigma'; \mathcal{P}' \vdash B$ and $\Sigma; \mathcal{P}, B \vdash C$. Thus, $\Sigma; \mathcal{P} \vdash B \supset C$. By the assumed equivalence, $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B$ and $I, \langle \Sigma, \mathcal{P} \rangle \Vdash B \supset C$. By the definition of satisfaction for implication, $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash C$. By the assumed equivalence again, this yields $\Sigma'; \mathcal{P}' \vdash C$.

To prove (2), assume that t is a Σ'-term of type τ and that $\Sigma + x : \tau; \mathcal{P} \vdash C$. Thus, $\Sigma; \mathcal{P} \vdash \forall_\tau x.B$. By the assumed equivalence, $I, \langle \Sigma, \mathcal{P} \rangle \Vdash \forall_\tau x.B$. By the definition of satisfaction for universal quantification, we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash B[t/x]$. By the assumed equivalence again, this yields $\Sigma'; \mathcal{P}' \vdash B[t/x]$. ∎

Given cut-elimination for intuitionistic logic, this lemma provides an immediate proof of the following theorem.

Theorem 3 *Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair and let I be the canonical model for $\langle \Sigma, \mathcal{P} \rangle$. For all Σ-formulas B, $\Sigma; \mathcal{P} \vdash B$ if and only if $I \Vdash B$. In particular, for every $B \in \mathcal{P}$, $I \Vdash B$.*

This theorem can be sharpened using the following definition of *order* for types and for formulas.

Next we define the notion of the canonical model at a given order. Such models contain, in a sense, fewer worlds than the canonical models introduced previously.

A dependent pair $\langle \Sigma, \mathcal{P} \rangle$ is of order n if all the types in Σ are of order n or less and all the formulas in $\mathcal{P}$ are of order n or less. Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair of order n. The *canonical model of order* n for $\langle \Sigma, \mathcal{P} \rangle$ is $[\mathcal{W}, I]$ where $\mathcal{W}$ is the set of all dependent pairs $\langle \Sigma', \mathcal{P}' \rangle$ of order n such that (i) Σ' extends Σ with constants of order at most $n-2$, and (ii) $\mathcal{P}'$ extends $\mathcal{P}$ with Σ'-formulas of order at most $n-2$. The mapping I is defined as before, namely, for all $\langle\Sigma',\mathcal{P}'\rangle \in \mathcal{W}$, the set $I(\langle\Sigma',\mathcal{P}'\rangle)$ contains all atomic A so that $\Sigma';\mathcal{P}' \vdash A$.

Notice that if $\langle\Sigma,\mathcal{P}\rangle$ is of order 1 then Σ is a first-order signature (all constants are of order 0 or 1) and $\mathcal{P}$ is a set of Horn clauses. The canonical model for such a dependent pair contains just one world, namely, the pair $\langle\Sigma,\mathcal{P}\rangle$.

Lemma 4 *Cut-elimination holds for $\mathcal{M}$ if and only if the following holds: Let $n \geq 1$, let $\langle\Sigma,\mathcal{P}\rangle$ be a dependent pair of order n, let I be the canonical model of order n for $\langle\Sigma,\mathcal{P}\rangle$, and let B be a Σ-formula of order $n-1$. Then $\Sigma;\mathcal{P} \vdash B$ if and only if $I \Vdash B$.*

Proof Assume first that cut-elimination holds for $\mathcal{M}$. We now prove by induction on the structure of B that $\Sigma;\mathcal{P} \vdash B$ if and only if $I,\langle\Sigma,\mathcal{P}\rangle \Vdash B$. The forward part of this equivalence is the same as in the proof of Lemma 2. Thus we only show details of the reverse implication for the two interesting cases.

Case: B is $B_1 \supset B_2$. Thus the order of B_1 is $n-2$ or less. Assume $I,\langle\Sigma,\mathcal{P}\rangle \Vdash B_1 \supset B_2$. Since $\Sigma;\mathcal{P}\cup\{B_1\} \vdash B_1$ and $\langle\Sigma,\mathcal{P}\cup\{B_1\}\rangle \in \mathcal{W}$, the inductive hypothesis yields $I,\langle\Sigma,\mathcal{P}\cup\{B_1\}\rangle \Vdash B_1$. By the definition of satisfaction of implication we must have $I,\langle\Sigma,\mathcal{P}\cup\{B_1\}\rangle \Vdash B_2$. But by the inductive hypothesis again, $\Sigma;\mathcal{P}\cup\{B_1\} \vdash B_2$ and $\Sigma;\mathcal{P} \vdash B_1 \supset B_2$.

Case: B is $\forall_\tau x B_1$. Thus the order of τ is $n-2$ or less. Assume $I,\langle\Sigma,\mathcal{P}\rangle \Vdash \forall_\tau x B_1$. Let d be a constant not a member of Σ. Since d is a $\Sigma\cup\{d\}$-term and since $\langle\Sigma\cup\{d\}\rangle$ is a member of $\mathcal{W}$, then we have $I,\langle\Sigma\cup\{d\},\mathcal{P}\rangle \Vdash B_1[d/x]$ by the definition of satisfaction of universal quantification. But by the inductive hypothesis again, we have $\Sigma\cup\{d\};\mathcal{P} \vdash B_1[d/x]$ and $\Sigma;\mathcal{P} \vdash \forall_\tau x B_1$.

The fact that cut-elimination holds follows just as in the proof of Lemma 2, except here we need to use the equivalence at various different orders. ∎

We shall need the following technical result.

Lemma 5 *Let $\langle\Sigma,\mathcal{P}\rangle$ be a dependent pair of order $n \geq 1$, and let $[\mathcal{W},I]$ be the canonical model of order n for $\langle\Sigma,\mathcal{P}\rangle$. Let $\langle\Sigma',\mathcal{P}'\rangle \in \mathcal{W}$, and let $[\mathcal{W}',I']$ be the canonical model of order n for $\langle\Sigma',\mathcal{P}'\rangle$. For all Σ'-formulas B of order n, $I,\langle\Sigma',\mathcal{P}'\rangle \Vdash B$ if and only if $I' \Vdash B$.*

This is proved by simple induction on the structure of B.

The next theorem shows that if $\langle\Sigma,\mathcal{P}\rangle$ is a dependent pair of order n then the canonical model for $\langle\Sigma,\mathcal{P}\rangle$ of order n is, in fact, a model for $\mathcal{P}$.

Theorem 6 *Let $\langle\Sigma,\mathcal{P}\rangle$ be a dependent pair of order n and let $[\mathcal{W},I]$ be the canonical model of order n for $\langle\Sigma,\mathcal{P}\rangle$. If B is of order n or less, then $\Sigma;\mathcal{P} \vdash B$ implies $I \Vdash B$.*

Proof We prove the following by induction on the structure of B: for every $\langle\Sigma',\mathcal{P}'\rangle \in \mathcal{W}$, if $\Sigma';\mathcal{P}' \vdash B$ then $I,\langle\Sigma',\mathcal{P}'\rangle \Vdash B$.

Cases: B is atomic or B is conjunctive. These cases are simple.

Case: B is $B_1 \supset B_2$ where B_1 is of order $n-1$ or less. Let $\langle\Sigma',\mathcal{P}'\rangle \in \mathcal{W}$ and let $\langle\Sigma'',\mathcal{P}''\rangle \in \mathcal{W}$ be such that $\langle\Sigma',\mathcal{P}'\rangle \preceq \langle\Sigma'',\mathcal{P}''\rangle$ and $I,\langle\Sigma'',\mathcal{P}''\rangle \Vdash B_1$. Let $[\mathcal{W}'',I'']$ be the canonical model of order n for $\langle\Sigma'',\mathcal{P}''\rangle$. By Lemma 5, $I'' \Vdash B_1$. By Lemma 4, $\Sigma'';\mathcal{P}'' \vdash B_1$. By cut-elimination, $\Sigma'';\mathcal{P}'' \vdash B_2$. By the inductive hypothesis, we have $I,\langle\Sigma'',\mathcal{P}''\rangle \Vdash B_2$. By the definition of satisfaction, we have $I,\langle\Sigma',\mathcal{P}'\rangle \Vdash B_1 \supset B_2$.

Case: B is $\forall_\tau x.B_1$ where τ is of order $n-1$ or less. Let $\langle\Sigma',\mathcal{P}'\rangle \in \mathcal{W}$ and let $\langle\Sigma'',\mathcal{P}''\rangle \in \mathcal{W}$ be such that $\langle\Sigma',\mathcal{P}'\rangle \preceq \langle\Sigma'',\mathcal{P}''\rangle$ and let t be a Σ''-term of type τ. By cut-elimination, $\Sigma'';\mathcal{P}'' \vdash B_1[t/x]$.

By the inductive hypothesis, we have $I, \langle \Sigma'', \mathcal{P}'' \rangle \Vdash B_1[t/x]$. By the definition of satisfaction, we have $I, \langle \Sigma', \mathcal{P}' \rangle \Vdash \forall_\tau x.B_1$. ∎

If Theorem 6 is specialized to just the case for order 1, it provides the familiar "minimal model" construction for first-order Horn clause theories [AvE82]. Thus, Theorem 6 can be seen as a generalization of that model construction to arbitrary orders.

Notice that the converse to Theorem 6 is not generally true if the formula B is of order n. For example, let i be the only primitive type, let p and q be the only predicates, each of sort $\langle i \rangle$, let Σ be the signature $\{a : i\}$ and let $\mathcal{P}$ be the set of Σ-formulas

$$\{p\ a,\ \forall_i x\ (p\ x \supset q\ x)\}.$$

Then, the formula of order 1, $\forall_i x\ (q\ x \supset p\ x)$ is valid in the canonical model of order 1 for $\langle \Sigma, \mathcal{P} \rangle$ but it is not provable from Σ and $\mathcal{P}$.

It is worth making the following simple observation about how canonical models can be considered minimal. We shall say that a Kripke model $\mathcal{N}$ satisfies $\langle \Sigma, \mathcal{P} \rangle$ if Σ is contained in the signature of $\mathcal{N}$ and if for every $B \in \mathcal{P}$, $\mathcal{N} \Vdash B$.

Theorem 7 *Let $\langle \Sigma, \mathcal{P} \rangle$ be a dependent pair, and let $\mathcal{K}$ be the canonical model for $\langle \Sigma, \mathcal{P} \rangle$. If $\mathcal{N}$ is a model of $\langle \Sigma, \mathcal{P} \rangle$ then $\mathcal{K} \Vdash B$ implies $\mathcal{N} \Vdash B$.*

Proof Since $\mathcal{K} \Vdash B$ then $\Sigma; \mathcal{P} \vdash B$. By the soundness of Kripke models and the fact that $\mathcal{N}$ models $\langle \Sigma, \mathcal{P} \rangle$, we have $\mathcal{N} \Vdash B$. ∎

5 Intuitionistic Linear Logic

If we now move to linear logic, we find that sequents have more structure and more possibilities to change during the search for proofs. As a result, this logic offers a richer setting for doing logic programming. Furthermore, since linear logic is a logic "behind" classical and intuitionistic logic, our work here and the next section will improve on the work in previous sections: we will only be refining and not discarding the previous results.

Most of the material here is taken from [HM94]. The reader should also refer to that paper for a more complete presentation. The material in Section 5.6 is new.

5.1 Weaknesses of hereditary Harrop formulas

As we have seen, if the sequents $\Sigma : \mathcal{P} \longrightarrow G$ and $\Sigma' : \mathcal{P}' \longrightarrow G'$ have occurrences on the same path in a proof, with first being closer to the endsequent, then $\Sigma \subseteq \Sigma'$ and $\mathcal{P} \subseteq \mathcal{P}'$. Thus, as a computation builds a proof from the bottom up, the left-hand sides of sequents do not decrease, and the formulas in them are available for backchaining on any number of times; they represent unbounded resources for constructing proofs.

There have been a few papers written that argue that implications in goal formulas provide an important approach to solving certain scoping issues in various applications. See, for example, the notion of *gap threading* in [PM90], theorem proving using natural deduction proof systems in [FM88], and state encapsulation in object-oriented style programming [HM90]. In each of these examples, intuitionistic implications in goals were used and they supplied part of the functionality that was needed. They also illustrated some weakness of intuitionistic logic. In particular, since contexts grow during the search for proofs, it is not possible to have a formula in a context replaced or deleted, two operations that were needed in these problems.

Linear logic, with its notion that formulas are resources that can be consumed, seems a natural logic to consider next. It should allow richer possibilities to model dynamics of computations at the level of logic.

5.2 Sequent calculus for linear logic

In order to refine hereditary Harrop formulas, we consider the linear logic connectives $\top, \&, \mathbf{1}, \otimes, \multimap, !$, and $\forall$. These are related to the connectives of classical and intuitionistic logic as follow.

Classical	Linear Additive	Linear Multiplicative
true	$\top$	$\mathbf{1}$
false	$\mathbf{0}$	$\bot$
$\wedge$	$\&$	$\otimes$
$\vee$	$\oplus$	$⅋$

Here, $\mathbf{1}$ is the identity for $\otimes$, *true* is the identity for $\&$, $\bot$ is the identity for $⅋$, and $\mathbf{0}$ is the identity for $\oplus$. The implication $\supset$ also splits into two implications, namely the linear implication $\multimap$ and the "intuitionistic" implication $\Rightarrow$. The differences between these are not best described by the difference between multiplicative and additive. The following equivalences, however, do hold.

$$(p \otimes q) \multimap r \equiv p \multimap q \multimap r \qquad (p \,\&\, q) \Rightarrow r \equiv p \Rightarrow q \Rightarrow r.$$

Notice that until now, we have used the terms "multiplicative" and "additive" to describe two different styles of inference rules with multiple premises. We have now extended the use of those terms to the logical connectives that are defined using those style inference rules.

$$\frac{}{\Sigma : \Delta \longrightarrow \top}\ \top R \qquad \frac{\Sigma : \Delta \longrightarrow B}{\Sigma : \Delta, \mathbf{1} \longrightarrow B}\ \mathbf{1}L \qquad \frac{}{\Sigma : \ \longrightarrow \mathbf{1}}\ \mathbf{1}R$$

$$\frac{\Sigma : \Delta, B_i \longrightarrow C}{\Sigma : \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&L\ (i = 1, 2) \qquad \frac{\Sigma : \Delta \longrightarrow B \quad \Sigma : \Delta \longrightarrow C}{\Sigma : \Delta \longrightarrow B \,\&\, C}\ \&R$$

$$\frac{\Sigma : \Delta, B_1, B_2 \longrightarrow C}{\Sigma : \Delta, B_1 \otimes B_2 \longrightarrow C}\ \otimes L \qquad \frac{\Sigma : \Delta_1 \longrightarrow B \qquad \Sigma : \Delta_2 \longrightarrow C}{\Sigma : \Delta_1, \Delta_2 \longrightarrow B \otimes C}\ \otimes R$$

$$\frac{\Sigma : \Delta_1 \longrightarrow B \qquad \Sigma : \Delta_2, C \longrightarrow E}{\Sigma : \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap L \qquad \frac{\Sigma : \Delta, B \longrightarrow C}{\Sigma : \Delta \longrightarrow B \multimap C}\ \multimap R$$

$$\frac{\Sigma : \Delta \longrightarrow C}{\Sigma : \Delta, !B \longrightarrow C}\ !W \qquad \frac{\Sigma : \Delta, !B, !B \longrightarrow C}{\Sigma : \Delta, !B \longrightarrow C}\ !C$$

$$\frac{\Sigma : \Delta, B \longrightarrow C}{\Sigma : \Delta, !B \longrightarrow C}\ !D \qquad \frac{\Sigma : !\Delta \longrightarrow B}{\Sigma : !\Delta \longrightarrow !B}\ !R$$

$$\frac{\Sigma : \Delta, B[t/x] \longrightarrow C}{\Sigma : \Delta, \forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Sigma : \Delta \longrightarrow B[y/x]}{\Sigma : \Delta \longrightarrow \forall x.B}\ \forall R,$$

provided that y is not free in the lower sequent.

Figure 7: The proof system LL for a fragment of linear logic

$$\frac{}{\Sigma : B \longrightarrow B}\ \mathit{identity} \qquad \frac{\Sigma : \Delta \longrightarrow B \qquad \Sigma : \Delta', B \longrightarrow C}{\Sigma : \Delta, \Delta' \longrightarrow C}\ \mathit{cut}$$

Figure 8: The initial and cut rules for LL.

The intuitionistic fragment of linear logic, which we consider first in this section, is the result of removing the $\parr$ connective, and the associated $\bot$ (a 0-ary $\parr$) and ? (an "infinite-ary" $\parr$, the de Morgan dual of !). Proof rules for these connectives are given in Figure 7 and the initial and cut rules for this proof system are given in Figure 8. Here, the left-hand side of sequents are multisets of formulas. As a result, the structural rule for exchange need not be explicitly stated. The structural rules of contraction and weakening are now available for those formulas marked with !: they are $!C$ (for contraction) and $!W$ (for weakening). The syntactic variable $!\Delta$ denotes the multiset $\{!C \mid C \in \Delta\}$. We write $\Sigma; \Delta \vdash_{LL} B$ if the sequent $\Sigma : \Delta \longrightarrow B$ has a proof in the proof system of Figure 7. Because all sequents in Figure 7 are single conclusion sequents, we shall be working completely within the "intuitionistic" fragment of linear logic.

5.3 Uniform proofs in intuitionistic linear logic

It is easy to see that linear logic, even over just the logical connectives considered here, is not an abstract logic programming language. For example, the sequents

$$\begin{gathered} \Sigma : a \otimes b \longrightarrow b \otimes a, \\ \Sigma : !a \longrightarrow !a \otimes !a, \\ \Sigma : !a \,\&\, b \longrightarrow !a, \\ \Sigma : b \otimes (b \multimap !a) \longrightarrow !a, \text{ and} \end{gathered}$$

$$\Sigma \ : \ \mathbf{1} \longrightarrow \mathbf{1}$$

are all provable in intuitionistic linear logic but do not have uniform LL-proofs. The problem here is that $\mathbf{1}R$, $\otimes R$ and $!\,R$ do not permute down over all the left-introduction rules.

If we drop $\mathbf{1}$, $\otimes$, and $!$, the resulting logic will, in fact, will be complete for uniform proofs. Unfortunately, this result is far too weak to be interesting and it does not supply a generalization of previous logic programming languages that we have seen. In particular, removing the ! means that there are no "potentially infinite" computation, as one would expect from a programming language. We can reintroduce the ! by allowing an intuitionistic implications as well as a linear implication.

In particular, we introduce a new sequent and a new proof system for the connectives $\top$, &, $\multimap$, $\Rightarrow$, and $\forall_\tau$. The new sequents will be of the form $\Sigma \ : \ \Gamma \ ; \ \Delta \longrightarrow B$ where Σ is a signature, B is a Σ-formula, Γ is a set of Σ-formulas, and Δ is a multiset of Σ-formulas. Such sequents have their left-hand context divided into two parts: the unbounded part, Γ, that corresponds to the left-hand side of intuitionistic sequents, and the bounded part, Δ, which corresponds to left-hand side of sequents of the purely linear fragment of linear logic (no !'s). Contraction and weakening are allowed (implicitly) in the unbounded part of the context, but not in the bounded part. As we show below, the sequent $\Sigma \ : \ B_1, \ldots, B_n \ ; \ C_1, \ldots, C_m \longrightarrow B$ can be mapped to the linear logic sequent

$$!\,B_1, \ldots, !\,B_n, C_1, \ldots, C_m \longrightarrow B.$$

The right introduction rules for the two implications are responsible for placing formulas into two parts of the left context. The right-introduction rule for linear implication adds its assumption to the bounded part of a context, and the right-introduction rule for the intuitionistic implication adds its assumption to the unbounded part of a context. These differences are naturally related to the fact that the intended meaning of $B \Rightarrow C$ is $(!\,B) \multimap C$.

Consider a sequent in which the bounded formulas are atomic. If the only logical connectives are $\multimap$ and $\Rightarrow$ then every formula in the bounded part of the context must be used exactly once: that is, they must be accounted for in some *identity* inference rule by matching them with the same formula on the right of a sequent. Such rigid control of resources is limiting for most uses. For example, if a data base is held in the bounded part of a context, then querying the data base about an item makes that item unavailable elsewhere. Also, before a computation on the data base can be finished, it is necessary to "read" all items in this way. The connectives $\top$ and & have the ability to erase parts of the bounded context (using $\top$) and to duplicate bounded contexts (using &). Thus, non-destructively reading a value from a data base can be achieved by first making a copy of the data base from which we destructively read one item and delete the rest: the original data base is untouched.

Figure 9 contains proof rules for the logical connectives $\top$, &, $\multimap$, $\Rightarrow$, and $\forall$. It is this collection of connectives that determines the logic programming language called Lolli. Notice the form of the left-introduction rules for the two implications: for $\multimap$, the bounded context is treated in a multiplicative fashion whereas for the $\Rightarrow$, the bounded context must be empty in the premise used to proved the antecedent of the implication. Notice that the inference rules in figure 9 are "focused", in the sense that left-introduction rules only work on formulas that label sequent arrow.

Proposition 8 *The sequent $\Sigma \ : \ \Gamma \ ; \ \Delta \longrightarrow B$, where Σ is a signature, B is a Σ-formula, Γ is a set of Σ-formulas, and Δ is a multiset of Σ-formulas, has an $\mathcal{L}$-proof (Figure 9) if and only if the sequent $!\,B_1, \ldots, !\,B_n, C_1, \ldots, C_m \longrightarrow B$ has a proof in linear logic.*

For a proof of this Proposition, see [HM94].

$$\frac{}{\Sigma : \mathcal{P} ; \Delta \longrightarrow \top} \qquad \frac{\Sigma : \mathcal{P} ; \Delta \longrightarrow B_1 \qquad \Sigma : \mathcal{P} ; \Delta \longrightarrow B_2}{\Sigma : \mathcal{P} ; \Delta \longrightarrow B_1 \mathbin{\&} B_2}$$

$$\frac{\Sigma : \mathcal{P}, B_1 ; \Delta \longrightarrow B_2}{\Sigma : \mathcal{P} ; \Delta \longrightarrow B_1 \Rightarrow B_2} \qquad \frac{\Sigma : \mathcal{P} ; \Delta, B_1 \longrightarrow B_2}{\Sigma : \mathcal{P} ; \Delta \longrightarrow B_1 \multimap B_2}$$

$$\frac{c : \tau, \Sigma : \mathcal{P} ; \Delta \longrightarrow B[c/x]}{\Sigma : \mathcal{P} ; \Delta \longrightarrow \forall_\tau x.B}$$

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

$$\frac{\Sigma : \mathcal{P} ; \Delta \xrightarrow{D_1} A}{\Sigma : \mathcal{P} ; \Delta \xrightarrow{D_1 \wedge D_2} A} \qquad \frac{\Sigma : \mathcal{P} ; \Delta \xrightarrow{D_2} A}{\Sigma : \mathcal{P} ; \Delta \xrightarrow{D_1 \wedge D_2} A}$$

$$\frac{\Sigma : \mathcal{P} ; \cdot \longrightarrow G \qquad \Sigma : \mathcal{P} ; \Delta \xrightarrow{D} A}{\Sigma : \mathcal{P} ; \Delta \xrightarrow{G \Rightarrow D} A}$$

$$\frac{\Sigma : \mathcal{P} ; \Delta_1 \longrightarrow G \qquad \Sigma : \mathcal{P} ; \Delta_2 \xrightarrow{D} A}{\Sigma : \mathcal{P} ; \Delta_1, \Delta_2 \xrightarrow{G \multimap D} A}$$

$$\frac{\Sigma : \mathcal{P} ; \Delta \xrightarrow{D[t/x]} A \qquad t \text{ is a } \Sigma\text{-term of type } \tau}{\Sigma : \mathcal{P} ; \Delta \xrightarrow{\forall_\tau x.D} A}$$

Figure 9: The proof system $\mathcal{L}$. The rule for universal quantification has the proviso that c is not declared in Σ.

5.4 An embedding of hereditary Harrop formulas

Girard has presented a mapping of intuitionistic logic into linear logic that preserves not only provability but also proofs [Gir87]. On the fragment of intuitionistic logic containing *true*, $\wedge$, $\supset$, and $\forall$, the translation is given by:

$$\begin{gathered}
(A)^0 = A, \text{ where } A \text{ is atomic}, \\
(\mathit{true})^0 = \top, \\
(B_1 \wedge B_2)^0 = (B_1)^0 \mathbin{\&} (B_2)^0, \\
(B_1 \supset B_2)^0 = \,!(B_1)^0 \multimap (B_2)^0, \\
(\forall x.B)^0 = \forall x.(B)^0.
\end{gathered}$$

That is, *true* and $\wedge$ are mapped to their additive versions, $\supset$ is mapped to $\Rightarrow$, and universal quantification is left unchanged. If we are willing to focus attention on only cut-free proofs, it is possible to define a "tighter" translation based on polarities. Consider the following two translation functions.

$$\begin{gathered}
(A)^+ = (A)^- = A, \text{ where } A \text{ is atomic} \\
(\mathit{true})^+ = \mathbf{1} \\
(\mathit{true})^- = \top \\
(B_1 \wedge B_2)^+ = (B_1)^+ \otimes (B_2)^+
\end{gathered}$$

$$\frac{}{\Sigma\,:\,\Gamma\longrightarrow \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Sigma\,:\,\Gamma\longrightarrow B}{\Sigma\,:\,\Gamma\longrightarrow !B}\ !R$$

$$\frac{\Sigma\,:\,\Gamma\longrightarrow \Delta, B_i}{\Sigma\,:\,\Gamma\longrightarrow \Delta, B_1\oplus B_2}\ \oplus R\ (i=1,2)$$

$$\frac{\Sigma\,:\,\Gamma\longrightarrow \Delta, B[x/t]}{\Sigma\,:\,\Gamma\longrightarrow \Delta, \exists x.B}\ \exists R \qquad \frac{\Sigma\,:\,\Gamma\longrightarrow \Delta_1, B_1 \quad \Sigma\,:\,\Gamma\longrightarrow \Delta_2, B_2}{\Sigma\,:\,\Gamma\longrightarrow \Delta_1,\Delta_2, B_1\otimes B_2}\ \otimes R$$

Figure 10: Additional rules for positive occurrences of $\mathbf{1}, \otimes, \oplus, !$, and $\exists$.

$$\begin{aligned}(B_1\wedge B_2)^- &= (B_1)^- \mathbin{\&} (B_2)^-\\ (B_1\supset B_2)^+ &= (B_1)^- \Rightarrow (B_2)^+\\ (B_1\supset B_2)^- &= (B_1)^+ \multimap (B_2)^-\\ (\forall x.B)^+ &= \forall x.(B)^+\\ (\forall x.B)^- &= \forall x.(B)^-\end{aligned}$$

The following proposition is proved by structural induction on cut-free proofs.

Proposition 9 *Let B be a formula and Γ a set of formulas, all over the logical constants* true, $\wedge, \supset$, *and* $\forall$. *Define $\Gamma^- = \{C^- \mid C\in\Gamma\}$. Then, $\Sigma\,:\,\Gamma\longrightarrow B$ has an* **I***-proof if and only if the sequent $\Sigma\,:\,\Gamma^-\,;\,\cdot\longrightarrow B^+$ has a cut-free proof using the rules from Figure 9..*

Applying this encoding to the Horn clause $A_1\wedge\ldots\wedge A_n\supset A_0$ yields the linear logic formula $A_1\otimes\ldots\otimes A_n\multimap A_0$.

5.5 Allowing right rules for some additional connectives

Since we are only interested in searching for cut-free proofs , it is possible to permit different sets of formulas to occur on the left and right of the sequent arrow. There are at least two ways to do this. We can expand the logic by allowing some occurrences of additional logical constants (as was done with $\vee$ and $\exists$ in definition (5) for *fohh*), or we can use higher-order quantification with respect to the given logic to "define" the additional constants.

Using the first approach, consider the following definition of two classes of formulas over the logical constants $\top, \&, \mathbf{1}, \otimes, \oplus, \multimap, \Rightarrow, !, \forall$, and $\exists$.

$$\begin{aligned}D &:= \top \mid A \mid D_1 \mathbin{\&} D_2 \mid G\multimap D \mid G\Rightarrow D \mid \forall x.D\\ G &:= \top \mid A \mid G_1 \mathbin{\&} G_2 \mid D\multimap G \mid D\Rightarrow G \mid \forall x.G \mid G_1\oplus G_2 \mid \mathbf{1} \mid G_1\otimes G_2 \mid\ !G \mid \exists x.G\end{aligned}$$

Here, D-formulas can appear in either part of the context on the left of a sequent while G-formulas, called *goal formulas*, can appear on the right of sequents. Given this extension, it is necessary to add to the proof system $\mathcal{L}$ right-introduction rules for $\mathbf{1}, \oplus, \otimes, !$ and $\exists$, which are found in Figure 10.

The second approach does not extend the logic by adding these logical constants directly but instead axiomatizes their right-introduction rules using higher-order quantification. The following clauses are appropriate definitions for these constants:

$$\begin{gathered}\forall P\forall Q[P\multimap (P\oplus Q)]\\ \forall P\forall Q[Q\multimap (P\oplus Q)]\\ \forall B\forall T[(B\ T)\multimap (\exists B)]\\ \top\Rightarrow \mathbf{1}\\ \forall P\forall Q[P\multimap Q\multimap (P\otimes Q)]\\ \forall P[P\Rightarrow\ !P]\end{gathered}$$

If we assume that there are no negative occurrences of any of these constants within a proof (except in these defining formulas) then this amounts to the same restriction as in the first approach.

To describe, then, the full mapping of *fohh* into linear logic using the polarity mapping, we need to add the following two clauses to the mapping of given in Subsection 5.4.

$$(B_1 \vee B_2)^+ = (B_1)^+ \oplus (B_2)^+$$
$$(\exists x.B)^+ = \exists x.(B)^+$$

Notice that our presentation of Lolli has been reversed to that used for *fohh*. In Section 4, we presented collections of definite clauses and goal formulas based on their polarities and allowed $\vee$ and $\exists$ in goal formulas but not in definite clauses. We then noticed that the core of *fohh* was freely generated by *true*, $\wedge$, $\supset$, and $\forall$. In this section, we presented Lolli first as being freely generated from $\top$, $\&$, $\multimap$, $\Rightarrow$, and $\forall$ (that is, we worked first with the core of Lolli), and then observed that we could add occurrences of $\mathbf{1}$, $\otimes$, $\oplus$, $!$, and $\exists$ to goal formulas.

5.6 An example

In this section we present a simple example of a logic specification. In order to present it, we use the syntax of λProlog. In particular, the combination `pi x\` denotes universal quantification of the variable `x` (its type will be determined from context), `-:` denotes linear implication, `:-` denotes the converse of linear implication, `,` denotes tensor, and `nil` and `::` denote the two constructors for lists.

A specification of a computation should do more than provide an approach to performing a computation: it should also provide a basis for reasoning about the computation specified.

Consider specifying the binary relation `reverse` that relates two lists if they are reverses of each other. First, consider how to compute the reverse of a list. Make a place for two piles on a table. Make one pile the list you wish to reverse and make the other pile empty. After this initialization, repeatedly move the top element from the first pile to the top of the second pile. When the first pile is empty, the second pile is the reverse of the original list. For example, consider the following two pairs of lists.

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

In more general terms: if we wish to reverse the list `L` to get `K`, first pick a binary relation `rv` to denote the pairing of lists above (this is predicate will not denote the reverse); then start with the atom `(rv L nil)`, do a series of backchaining over the clause

```
rv (X :: P) Q :- rv P (X :: Q).
```

to get to the formula `(rv nil K)`. If this can be done, then `K` is the result of reversing `K`. The entire specification of reverse can be written as the following single formula.

```
reverse L K :-
  pi rv\((pi X\ pi P\ pi Q\  rv (X :: P) Q :- rv P (X :: Q)) =>
      rv nil K -: rv L nil)
```

Notice that the clause used for backchaining is to the left of an intuitionistic implication (so it can be used any number of times) while the formula `rv nil K` is to the left of a linear implication (can be used once). Since the base case of this iteration is used exactly once, this seems like a sensible choice.

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

```
pi rv\ (
  (pi X\ pi P\ pi Q\  rv (X :: P) Q :- rv P (X :: Q) ) =>
  rv nil K -: rv L nil)
```

is provable. Since we are in logic, we can instantiate this quantifier with any binary predicate expression and the result is still provable. So choose to instantiate it with the lambda-expression `x\y\ (not (rv y x))`. (The infix backslash denotes lambda-abstraction: in logical symbols, this substitution can be written as $\lambda x \lambda y (rv\ y\ x)^{\perp}$.) The resulting formula

```
(pi X\pi P\pi Q\ not (rv Q (X :: P)) :- not (rv (X :: Q) P)) =>
not (rv K nil) -: not (rv nil L)
```

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

```
  (pi X\ pi P\ pi Q\   rv (X :: Q) P:- rv Q (X :: P) ) =>
   rv nil L -: rv K nil
```

If we now universally generalize on `rv` we again have proved the body of the reverse clause, but this time with `L` and `K` switched.

5.7 Additional readings

This section is based on the paper [HM94], which derives from the paper [HM91] and the PhD thesis of Joshua Hodas [Hod94]. See [Hod92] for a treatment of gap in natural language parsing inside Lolli. In [HM94], a lazy way of splitting contexts for the multiplicative rules for $\otimes$-R and $\multimap$-L was developed, using the so-called "input-output model of resource consumption." This approach works well when only multiplicative connectives are used, but in the presence of additive connectives, it can be improved. In [Hod94], the treatment of the additive truth $\top$ is addressed and in [CHP96] the additive & is also addressed.

It is possible to design a dependent typed calculus around the principles of Lolli. See, for example, [CP96].

Harland and Pym have also approached the design of linear logic programming languages using the notion of goal-directed search [HP91, HP92].

A survey of various approaches to using linear logic in logic programming can be found in [Mil95].

5.8 Exercises

1. Below is the specification of two predicates. The `greaterEq` is the same of in the problem above. Assume that the predicate `(greaterEq N M)` is provable (consuming no resources) if and only if `N` is greater than or equal to `M`.

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

6 Forum

The following brief section serves mostly to introduce the design goals of Forum and a proof system for it. More extensive information can be found in [Mil96].

6.1 Designing Forum

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 $\top$. 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* [MNPS91], 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.)

- 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 !.)

Each of these logics include the other and allow for increasing richer forms of *abstraction*. As we have seen, Horn clauses are flat and do not hide anything. Hereditary Harrop formulas allow for notions of modular programming and abstract datatypes. Moving to Lolli allows also the encapsulation (hiding) of state. For all this gain in expressiveness, these languages do not offer any primitives for the specification of concurrence, that is, there are no primitive for communication or synchronization.

J-M Andreoli and R Pareschi introduced the first extension of Horn clauses using linear logic [AP90]. The formulas of their language, called LO, were 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 $\multimap$.

As we shall later see, the multiple $\parr$s in the head of LO clauses allow for specifications to address some issues of concurrency. Given the restriction on universal quantification and on implications, LO is a flat language like Horn clauses that does not admit abstractions. Of course, it would be natural to desire both elements of abstraction and concurrency within a specification language. Linear logic should contain a super-language that would allow these different features to be placed in a common language.

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.

Before proceeding, it is important to generalize the definition of uniform proofs to sequents that contain multiple conclusions (an essential aspect of sequents that contain $\parr$.) The following generalization of the definition of uniform proof was introduced in [Mil93] where it was shown that a certain logic specification inspired by the π-calculus [MPW92a] 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.*

As it turns out, Forum is a presentation of all of linear logic since it contains a complete set of connectives. A proof system for all of linear logic can be found in Figures 11 and 12. The connectives missing from Forum are directly definable using the following logical equivalences.

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

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

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

Thus, Forum can be seen as Lolli with just the addition of $\perp$.

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 $\perp$, ⅋, $\multimap$, 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.

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 [Fel93] uses a meta-logical presentation to specify full logic at the object-level. Andreoli [And92] 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 6.3.

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.

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

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

$$\frac{}{\Delta \longrightarrow \top, \Gamma}\ \top R \qquad \frac{\Delta \longrightarrow \Gamma}{\Delta, \mathbf{1} \longrightarrow \Gamma}\ \mathbf{1}L \qquad \frac{}{\longrightarrow \mathbf{1}}\ \mathbf{1}R$$

$$\frac{}{\Delta, \mathbf{0} \longrightarrow \Gamma}\ \mathbf{0}L \qquad \frac{\Delta \longrightarrow \Gamma}{\Delta \longrightarrow \bot, \Gamma}\ \bot R \qquad \frac{}{\bot \longrightarrow}\ \bot L$$

$$\frac{\Delta, B_i \longrightarrow \Gamma}{\Delta, B_1 \,\&\, B_2 \longrightarrow \Gamma}\ \& L\ (i = 1, 2) \qquad \frac{\Delta \longrightarrow B, \Gamma \qquad \Delta \longrightarrow C, \Gamma}{\Delta \longrightarrow B \,\&\, C, \Gamma}\ \& R$$

$$\frac{\Delta \longrightarrow B_i, \Gamma}{\Delta \longrightarrow B_1 \oplus B_2, \Gamma}\ \oplus R\ (i = 1, 2) \qquad \frac{\Delta, B \longrightarrow \Gamma \qquad \Delta, C \longrightarrow \Gamma}{\Delta, B \oplus C \longrightarrow \Gamma}\ \oplus L$$

$$\frac{\Delta, B_1, B_2 \longrightarrow \Gamma}{\Delta, B_1 \otimes B_2 \longrightarrow \Gamma}\ \otimes L \qquad \frac{\Delta_1 \longrightarrow B, \Gamma_1 \qquad \Delta_2 \longrightarrow C, \Gamma_2}{\Delta_1, \Delta_2 \longrightarrow B \otimes C, \Gamma_1, \Gamma_2}\ \otimes R$$

$$\frac{\Delta_1, B \longrightarrow \Gamma_1 \qquad \Delta_2, C \longrightarrow \Gamma_2}{\Delta_1, \Delta_2, B \parr C \longrightarrow \Gamma_1, \Gamma_2}\ \parr L \qquad \frac{\Delta \longrightarrow B \parr C, \Gamma}{\Delta \longrightarrow B, C, \Gamma}\ \parr R$$

$$\frac{\Delta \longrightarrow \Gamma}{\Delta, !B \longrightarrow \Gamma}\ !W \qquad \frac{\Delta, !B, !B \longrightarrow \Gamma}{\Delta, !B \longrightarrow \Gamma}\ !C \qquad \frac{\Delta, B \longrightarrow \Gamma}{\Delta, !B \longrightarrow \Gamma}\ !D$$

$$\frac{\Delta \longrightarrow \Gamma}{\Delta \longrightarrow ?B, \Gamma}\ ?W \qquad \frac{\Delta \longrightarrow ?B, ?B, \Gamma}{\Delta \longrightarrow ?B, \Gamma}\ ?C \qquad \frac{\Delta \longrightarrow B, \Gamma}{\Delta \longrightarrow ?B, \Gamma}\ ?D$$

$$\frac{!\Delta \longrightarrow B, ?\Gamma}{!\Delta \longrightarrow !B, ?\Gamma}\ !R \qquad \frac{!\Delta, B \longrightarrow ?\Gamma}{!\Delta, ?B \longrightarrow ?\Gamma}\ ?L$$

$$\frac{\Delta, B[t/x] \longrightarrow \Gamma}{\Delta, \forall x.B \longrightarrow \Gamma}\ \forall L \qquad \frac{\Delta \longrightarrow B[y/x], \Gamma}{\Delta \longrightarrow \forall x.B, \Gamma}\ \forall R$$

$$\frac{\Delta \longrightarrow B[t/x], \Gamma}{\Delta \longrightarrow \exists x.B\Gamma}\ \exists R \qquad \frac{\Delta, B[y/x] \longrightarrow \Gamma}{\Delta, \exists x.B \longrightarrow \Gamma}\ \exists L,$$

provided that y is not free in the lower sequent or $\forall R$ and $\exists L$.

$$\frac{\Delta \longrightarrow B, \Gamma}{\Delta, B^{\bot} \longrightarrow \Gamma}\ \mathit{notL} \qquad \frac{\Delta, B \longrightarrow \Gamma}{\Delta \longrightarrow B^{\bot}, \Gamma}\ \mathit{notR},$$

Figure 11: The introduction rules for linear logic.

$$\frac{}{B \longrightarrow B}\ \mathit{identity} \qquad \frac{\Delta \longrightarrow B, \Gamma \qquad \Delta', B \longrightarrow \Gamma'}{\Delta, \Delta' \longrightarrow \Gamma, \Gamma'}\ \mathit{cut}$$

Figure 12: The initial and cut rules for linear logic.

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

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

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

Formulas of the form

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

where $G_1, \ldots G_m$ are arbitrary Forum formulas and $A_1, \ldots A_m$ are atomic formulas, are called *clauses*. Here, occurrences of $\hookrightarrow$ are either occurrences of $\multimap$ or $\Rightarrow$. An empty $⅋$ $(p = 0)$ is written as $\perp$. The formula $A_1 \mathbin{⅋} \cdots \mathbin{⅋} A_p$ is the *head* of such a clause. If $p = 0$ then we say that this clause has an *empty head*. The formulas of LinLog [And92] 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.

6.2 Proof system for Forum

If we add to the Lolli language in the preceding section the connective $⅋$, its identity $\perp$, and the modal $?$, then we arrive at the *Forum* logic programming language. The proof systems for Forum is given in Figure 13 and is called $\mathcal{F}$.

Theorem 10 *The sequent* $\Sigma\colon \Psi; \Delta \longrightarrow \Gamma; \Upsilon$ *has an* $\mathcal{F}$ *proof if and only if* $!\,\Psi, \Delta \vdash \Gamma, ?\,\Upsilon$.

Proof The forward direction is soundness and can be proved by simple induction on the structure of $\mathcal{F}$ proofs. The converse is completeness and this is a harder result. One approach is to extended the similar result for Lolli, but this time accounting for the $⅋$, $\perp$, and $?$ connectives. However, Andreoli has a result in [And92] that he calls *focused proofs* and he shows that focused proofs are complete for linear logic. It is possible to translated between focused proofs and the proofs in $\mathcal{F}$. Completeness for $\mathcal{F}$ then follows from completeness for focused proofs. ∎

6.3 Multiset rewriting as backchaining

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

$$a \mathbin{⅋} b \circ\!\!- c \mathbin{⅋} d \mathbin{⅋} e.$$

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

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

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

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

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

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

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

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

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

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

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

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

Figure 13: 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 $i = 2$.

When presenting examples of Forum code we often use (as in this example) $\circ\!-$ 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 $\circ\!-$ 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 $\multimap$.

6.4 Further readings

The material in this section is taken largely from the paper [Mil96]. In his PhD thesis [Chi95], Chirimar presents specifications of the operation semantics of a programming language similar to Standard ML. He is able to give a modular specification of call-by-value evaluation, of exceptions, of references, and of continuations. He was also able to proved various identities concerning object-level programs by using the meta-theory of linear logic. He also presented a specification of the pipe-line processing of the DLX RISC processor of Hennessy and Patterson [HP90] and used linear logic to help prove its equivalence to its sequential, machine code specification.

Proof search using linear logic and/or Forum has been used to represent object-oriented programming languages [DM95, BDLM96]. Topics in concurrency have often been addresses as well: see, for example, [AP90, AP91], [BG96, Gug95, Gug96, Gug94], [KY93], and [Mil93]. See also the survey paper [Mil95].

6.5 Exercises

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

$$G ::= \bot \mid \top \mid A \mid G_1 \mathbin{\&} G_2 \mid G_1 \parr G_2$$

$$D ::= G \multimap (A_1 \parr \ldots \parr A_n) \mid \forall_i x D,$$

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.

$$\frac{}{\mathcal{P} \longrightarrow \Gamma, \top} \qquad \frac{\mathcal{P} \longrightarrow \Gamma}{\mathcal{P} \longrightarrow \Gamma, \bot} \qquad \frac{\mathcal{P} \longrightarrow \Gamma, G_1, G_2}{\mathcal{P} \longrightarrow \Gamma, G_1 \parr G_2}$$

$$\frac{\mathcal{P} \longrightarrow \Gamma, G_1 \quad \mathcal{P} \longrightarrow \Gamma, G_2}{\mathcal{P} \longrightarrow \Gamma, G_1 \;\&\; G_2}$$

$$\frac{\mathcal{P} \longrightarrow \Gamma, G}{\mathcal{P} \longrightarrow \Gamma, A_1, \ldots, A_n} \quad \text{provided a formula in } \mathcal{P} \text{ has the ground instance } G \multimap (A_1 \parr \ldots \parr A_n).$$

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.

2. 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`.

   (a) 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.)

   (b) 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 $\{n_1, \ldots, n_m\}$.

3. Below are specifications of two binary predicates.

```
pred1 L K <=  all load\all unload\all item\
  (all X\all M\   load (X::M) o- (item X -o   load M)) =>
  (all X\all M\ unload (X::M) o-  item X,   unload M)  =>
  (load nil   o-  unload K)                           -o
  unload nil                                          -o
                                                 load L.

pred2 L K <=  all load\all unload\all item\
  (all X\all M\   load (X::M)         o- item X | load M) =>
  (all X\all M\ unload (X::M) | item X o- unload M       ) =>
  (load nil   o-  unload K)                               -o
  unload nil                                              -o
                                                     load L.
```

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

(a) It turns out that both of these clauses specify the same relation. What is that relation? Informally justify your answer.

(b) 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.

References

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

[AP90] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In *Proceeding of the Seventh International Conference on Logic Programming, Jerusalem*, May 1990.

[AP91] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

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

[BDLM96] Michele Bugliesi, Giorgio Delzanno, Luigi Liquori, and Maurizio Martelli. A linear logic calculus of objects. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming.* MIT Press, September 1996.

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

[BLM94] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

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

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

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

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logic framework. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press. An extended version of this paper will appear in *Information and Computation.*

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

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[Fit69] Melvin C. Fitting. *Intuitionistic Logic Model Theory and Forcing.* North-Holland, 1969.

[FM88] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61–80, Argonne, IL, May 1988. Springer-Verlag.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row, 1986.

[Gen69] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, 1989.

[Gug94] Alessio Guglielmi. Concurrency and plan generation in a logic programming language with a sequential operator. In P. Van Hentenryck, editor, *Logic Programming, 11th International Conference, S. Margherita Ligure, Italy*, pages 240–254. Mit Press, 1994.

[Gug95] Alessio Guglielmi. Sequentiality by linear implication and universal quantification. In Jörg Desel, editor, *Structures in Concurrency Theory*, Workshops in Computing, pages 160–174. Springer-Verlag, 1995.

[Gug96] Alessio Guglielmi. *Abstract Logic Programming in Linear Logic—Independence and Causality in a First Order Calculus.* PhD thesis, Università di Pisa, 1996.

[Har60] R. Harrop. Concerning formulas of the types $A \to B \vee C,\ A \to (Ex)B(x)$ in intuitionistic formal systems. *Journal of Symbolic Logic*, pages 27–32, 1960.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *JACM*, 32(1):137–161, 1985.

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

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

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

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

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994. Available as University of Pennsylvania Technical Reports MS-CIS-92-28 or LINC LAB 269.

[HP90] J. Hennesy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufman Publishers, Inc., 1990.

[HP91] James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming (extended abstract). In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Logic Programming Symposium, San Diego*, pages 304–318, San Diego, November 1991. MIT Press.

[HP92] James Harland and David Pym. Resolution in fragments of classical linear logic (extended abstract). In A. Voronkov, editor, *Proceedings of the Russian Conference on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 30–41, St. Petersburg, July 1992. Springer-Verlag.

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

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

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

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

[Mil90] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[Mil91] Dale Miller. Proof theory as an alternative to model theory. *Newsletter of the Association for Logic Programming*, August 1991. Guest editorial.

[Mil93] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265. Springer-Verlag, 1993.

[Mil95] Dale Miller. A survey of linear logic programming. *Computational Logic: The Newsletter of the European Network in Computational Logic*, 2(2):63 – 67, December 1995. ftp://ftp.cis.upenn.edu/pub/papers/miller/ComputNet95/llsurvey.html.

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

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.

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

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

[Pra65] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge MA, 1986.

[Tro73] Anne Sjerp Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer Verlag, 1973.