Proof Search and Computation

A Monograph

Draft: April 6, 2011

These lecture notes extracted from a larger set of lecture notes: material on linear logic and linear logic programming have been dropped.

© **Dale Miller**
INRIA Saclay - Île-de-France and
Laboratoire d'Informatique (LIX), École Polytechnique
Rue de Saclay
91128 PALAISEAU Cedex FRANCE
dale.miller at inria.fr

Contents

Preface

This monograph develops a foundation for viewing computation as "proof search". The sequent calculus is used as a framework for presenting classical, intuitionistic, and linear logics and for describing the normal form theorems that are used to describe and analyze computation in logic programming. Goal-directed proof search is formalized using the technical notions of *uniform proofs* and *backchaining*. These results are applied to logic programming languages based on Horn clauses, hereditary Harrop formulas, and linear logic.

This monograph is largely self-contained. The reader should be familiar with the basic syntactic properties of first-order logic and the λ-calculus, but nothing much is needed beyond the first definitions in these topics. No background in the formal representation of proofs is needed although such a background is useful in the earliest chapters.

We shall occasionally present example logic programs to help illustrate proof theoretic concepts. While some familiarity with Prolog is useful for understanding our examples, it is also likely that a knowledge of Prolog's advanced and mostly non-logical features will be a barrier to understanding the full role that logic can play in the specification of computation. When we present examples of logic programs, we shall use the syntactic conventions of the λProlog variant of Prolog.

The search for proofs has many dimensions that we shall not address here. For example, we shall not discuss the unification of terms, although this is central to most implementations of proof search systems. We shall also not consider the more general problems involved with the specification of interactive and automatic theorem provers.

The scope of this volume is purposely narrowed: no attempt has been made to consider a significant part of related literature. We have choosen to concentrate on how rather simple and natural structures in the sequent calculus can be used to illuminate the nature and possibilities for logic programming.

Acknowledgments. Versions of this monograph have been used in a number of graduate level courses in Paris, Copenhagen, and Venice. Many students have suggested improvements to earlier versions of this monograph and I thank them for their comments and criticisms.

1

Introduction

Since logic can be applied to computing and logic programming in a number of ways, it is worth providing an overview of the roles that logic often plays, if only to help us see the particular niche that is our focus here.

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 the *computation-as-model* approach, computations are encoded as mathematical structures, containing such items as nodes, transitions, and state. 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. This use of logic to represent and reason about computation is probably the oldest and most broadly successful use of logic for representing computation.

The *computation-as-deduction* approach, uses directly pieces of logic's syntax (such as formulas, terms, types, and proofs) as elements of the specified computation. In this much more rarefied setting, there are two rather different approaches to how computation is modeled.

The *proof normalization* approach views the state of a computation as a proof term and the process of computing as normalization (know variously as β-reduction or cut-elimination). Functional programming can be explained using proof-normalization as its theoretical basis [ML82] and has been used to justify the design of new functional programming languages [Abr93].

The *proof search* approach views the state of a computation as a sequent (a structured collection of formulas) and the process of computing as the process of searching for a proof of a sequent: the changes that take place in sequents capture the dynamics of computation. Logic programming can be explained using proof search as its theoretical basis [MNPS91] and has been used to justify the design of new logic programming languages, some of which are discussed later.

The divisions proposed above are informal and suggestive: such a classification is helpful in pointing out different sets of concerns represented by these two broad approaches (reductions, confluence, etc, versus unification, backtracking search, etc). Of course, a real advance in computation logic might allow us merge or reorganize this classification.

1.2 Proof search as an approach to logic programming

The term *proof search* as it is used in this text has a number of parallel with the term *logic programming*. In the late 1980's and early 1990's, the proof theory for classical, intuitionistic, and linear logic was used to motivate the design of logic programming languages that significantly extended the expressive strength of those languages based on first-order Horn clauses. In principle, proof search in higher-order intuitionistic logic allows for abstractions such as modular programming and higher-order programming as well as providing declarative treatments of data-structures that contained binders. When embracing linear logic as well, richer dynamics of computation can be captured using logical formulas instead of non-logical terms and data-structures. Besides presenting the proof theory justifications for the design of some of these richer logic programming languages, we shall also present numerous examples of programming in these languages.

2

Terms, formulas, sequents

In matters of the presentation of the syntax of terms and formulas, we follow Alonzo Church's Simple Theory of Types [Chu40] since it provides a simple means to integrate propositional logic, (multi-sorted) first-order logic, and a higher-order logic all within the same framework.

2.1 Syntactic expressions as λ-expressions

The untyped λ-calculus will serve as our most primitive notion of syntactic expression, allowing us to uniformly represent types, terms, formulas, and sequents. While the untyped λ-calculus is rather complex, we shall limit our use of it to β-normal expressions: in this case, much of that complexity disappears. In fact, we shall rely mostly on those β-normal forms that can be given a simple type. To reinforce our use of the untyped λ-calculus for representing only syntax (and not functional programs intended for computation), we shall refer to untyped λ-terms as untyped λ-expressions.

One advantage in choosing λ-expressions as a starting point is that they provide us with a universal notion of binding and substitution that all syntactic objects directly inherit. We shall assume that the reader is familiar with the most basic notions behind the untyped λ-calculus. We review a few such notions here.

We shall assume the existence of a fixed and denumerably infinite set of tokens, which are primitive syntactic expressions. There are two other means for building other syntactic expressions, following the usual formation rules of the λ-term. Given two syntactic structures, say M and N, their application is (MN) (applications is thus the infix juxtaposition operation and it associates to the left). Given a syntactic structure M and a token x, the abstraction of x over M is $(\lambda x.M)$. Here, the token x is a bound variable with scope M.

The usual notions of free and bound variables are assumed, as is the concept of alphabetic conversion of bound variables (via the α-conversion rule): we identify two syntactic expressions up to α-conversion. A term is *β-normal* if it is of the form $\lambda x_1 \dots \lambda x_n.(ht_1 \dots t_m)$ where $n, m \geq 0$, $h, x_1, \dots, x_n$ are tokens, and the terms $t_1, \dots, t_m$ are all in β-normal form. In this case, we call the list $x_1, \dots, x_n$ the *binder*, the token h the *head*, and the list $t_1, \dots, t_m$ the *arguments* of the term. Reduction following the β-rule replaces a *β-redex*, that is, a subexpression of the form $(\lambda x.M)N$, with the capture avoiding substitution of N for x in M. Reduction following the η-rule replaces a subexpression of the form $\lambda x(Mx)$ with M, provided that x is not free in M. When we use the terms *β-conversion* and *$\beta\eta$-conversion*, we shall always assume the α-conversion rule is implicit.

Our main use of β-reduction will be to formalize substitution. In particular, the notation $M[N/x]$ denotes the β-normal form of $(\lambda x.M)N$.

Exercise 1. Is there an expression N such that $(\lambda x.w)[N/w]$ is equal to $\lambda y.y$ (modulo α-conversion, of course)?

2.2 Types

The token o is reserved and is used as the type of formulas (to be defined in Section 2.4). This type must not be confused with a type for boolean values: objects of type o are syntactic expressions. Let S be a fixed, non-empty set of tokens that does not contain o. The types in $S \cup \{o\}$ are *primitive types* (also called *sorts*). The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of "arrow 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 $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. (Using the terminology of Section 2.1, $\rightarrow$ is a token now declared with a specific role and the expression $\tau_1 \rightarrow \tau_2$ is the infix presentation of the expression $((\rightarrow \tau_1)\tau_2)$.)

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

$$\mathrm{ord}(\tau) = 0 \quad \text{provided } \tau \in S \cup \{o\}$$
$$\mathrm{ord}(\tau_1 \rightarrow \tau_2) = \max(\mathrm{ord}(\tau_1) + 1, \mathrm{ord}(\tau_2))$$

Notice that τ has order 0 or 1 if and only if all the argument types of τ are primitive types. 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.

2.3 Signatures and terms

Signatures are used to formally *declare* that certain tokens are of a certain type. In particular, a *signature (over S)* is a set Σ (possibly empty) of pairs, written as $x{:}\,\tau$, where τ is a type and x is a token. We require a signature to be *functional* in the sense that for every token x, if $x{:}\,\tau$ and $x{:}\,\sigma$ are members of Σ then $\tau = \sigma$. More generally, we use signatures in judgments such as $\Sigma \blacksquare\!\!\vdash t{:}\,\tau$ where the variables in Σ are considered bindings over the entire judgment. Here also, t is a term and τ is a type. If we provided a more literal encoding of such a typing judgment as an untyped λ-expression, the judgment, for example, $x : \tau_1, y : \tau_2 \blacksquare\!\!\vdash t{:}\,\tau$ could be encoded as the λ-expression

$$\mathit{loc}\ \tau_1\ (\lambda x.\ \mathit{loc}\ \tau_2\ (\lambda y.\ \blacksquare\!\!\vdash\ (:\ x\ \tau)))$$

where *loc* is a token introduced to indicate that a binder is added to a judgment, $\blacksquare\!\!\vdash$ is a token used to separate the binders from the target judgment, and : is a token used to pair a term with a type. Thus, two judgments are identified if they differ by systematic renames of declared tokens. We shall not generally care to be so literal in our encodings of judgments, but it is usual to see at least once.

If we were to allow non-normal λ-terms to have types, then the proof system including the following three rules

$$\frac{}{\Gamma, x{:}\,t \blacksquare\!\!\vdash x{:}\,t} \qquad \frac{\Sigma \blacksquare\!\!\vdash t{:}\,\sigma \rightarrow \tau \quad \Sigma \blacksquare\!\!\vdash s{:}\,\sigma}{\Sigma \blacksquare\!\!\vdash (ts){:}\,\tau} \qquad \frac{\Sigma, x{:}\,\tau \blacksquare\!\!\vdash M{:}\,\sigma}{\Sigma \blacksquare\!\!\vdash (\lambda x.M){:}\,\tau \rightarrow \sigma}$$

$$\frac{}{\Gamma, x{:}\,t \nvdash x{:}\,t} \qquad \frac{\Sigma \nvdash N{:}\,\sigma \quad \Sigma, x{:}\,\sigma' \nvdash M{:}\,\tau}{\Sigma, f{:}\,\sigma \to \sigma' \nvdash M[(fN)/x]{:}\,\tau} \qquad \frac{\Sigma, x{:}\,\tau \nvdash M{:}\,\sigma}{\Sigma \nvdash (\lambda x.M){:}\,\tau \to \sigma}$$

Fig. 2.1. Typing judgment for Σ-terms of type τ.

would suffice. We shall, instead, adopt the inference rules in Figure 2.1 as formal definition of the proof system for typing, since it gives types only to β-normal terms. If the judgment $\Sigma \nvdash t{:}\,\tau$ is provable then we say that t is a *Σ-term of type τ*.

Exercise 2. Prove that if t is a Σ-term of type τ, then t is β-normal.

Exercise 3. Fix a set of sorts S and a signature Σ over S. Prove that if there are *primitive* types τ and τ' such that $\Sigma \nvdash t{:}\,\tau$ and $\Sigma \nvdash t{:}\,\tau'$, then $\tau = \tau'$. Show that this statement is not true if we allow τ and τ' to be non-primitive.

Exercise 4. Fix a set of sorts S and a signature Σ over S. Prove that if t is a Σ-term of type τ and τ is primitive then the binder of t is empty, the head of t is given a type, say, $\tau_1 \to \cdots \to \tau_m \to \tau_0$ and for $i = 1, \ldots, m$, the i^{th} argument of t is a Σ-term of type τ_i.

2.4 Formulas

Important constants to declare when presenting a logic are those denoting the connectives and quantifiers. These *logical constants* are declared by a signature in which all token are declared a type that has target type o. These constants are generally fixed for a given logic. For example, in Chapter 3, classical and intuitionistic logics are considered using the following signature for declaring the logical constants.

$$\{\top{:}\,o,\ \bot{:}\,o,\ \wedge{:}\,o \to o \to o,\ \vee{:}\,o \to o \to o,\ \supset{:}\,o \to o \to o\} \cup \{\forall_\tau{:}\,(\tau \to o) \to o \mid \tau \in S\} \cup \{\exists_\tau{:}\,(\tau \to o) \to o \mid \tau \in S\}$$

Notice that this signature contains types of order 0, 1, and 2. We will follow the usual conventions in writing expressions with these symbols: The binary symbols $\wedge$, $\vee$, and $\supset$ are written in infix notions with $\wedge$ and $\vee$ associating to the left and $\supset$ associating to the right and $\wedge$ has higher priority than $\vee$ which has higher priority than $\supset$. The expressions $\forall_\tau \lambda x.B$ and $\exists_\tau \lambda x.B$ are abbreviated as $\forall_\tau x.B$ and $\exists_\tau x.B$, respectively, or as simply $\forall x.B$ and $\exists x.B$ if the value of the type subscript is not important or can easily be inferred from context.

After fixing the declaration of logical constants, say Σ_0, we fix the set of non-logical symbols, say Σ. Such symbols may be used as constants or variables depending on the context. Let $c{:}\,\tau_1 \to \cdots \to \tau_n \to \tau_0 \in \Sigma_1$, where τ_0 is a primitive type and $n \geq 0$. If τ_0 is o, then c is a *predicate symbol of arity n*. If $\tau_0 \in S$ (i.e., τ_0 is not o), then c is a *function symbol of arity n*; if $n = 0$, we also say that c is an *individual symbol*.

A $\Sigma_0 \cup \Sigma$-term of type o is also called a $\Sigma_0 \cup \Sigma$*-formula*, or more usually either a Σ*-formula* (since Σ_0 is usually fixed) or just a *formula* (if Σ is understood). Notice that in this presentation of logic, formulas are special cases of terms.

A logic is *propositional* if all the logical constants have types that are order 0 or 1. A logic is *first-order* if all the logical constants have types that are order 0, 1, or 2. If a logic contains constants with order greater than 2, the logic is said to be a *higher-order logic*.

A signature is *propositional* if all its constants are of type o. A signature is *first-order* if all its constants are of first-order type. If Σ_0 is the declaration for a propositional logic and Σ is a propositional signature, then a $\Sigma_0 \cup \Sigma$-formula is a *propositional*

formula. Similarly, if Σ_0 is the declaration for a first-order logic and Σ is a first-order signature, then a $\Sigma_0 \cup \Sigma$-formula is a *first-order formula*. If Σ_0 is the declaration for a higher-order logic and Σ is a signature, then a $\Sigma_0 \cup \Sigma$-formula is a *higher-order formula*.

Assume that Σ_0 declares logical connectives for a first-order logic and that Σ is a first-order signature. Let τ be a primitive type different from o. A first-order term t of type τ is either a token of type τ or it is of the form $(f\ t_1 \dots t_n)$ where f is a function symbol of type $\tau_1 \to \cdots \to \tau_n \to \tau$ and, for $i = 1, \dots, n$, t_i is a term of type τ_i. In the latter case, f is the head and $t_1, \dots, t_n$ are the arguments of this term. Similarly, a first-order formula either has a logical symbol as its head, in which case, it is said to be *non-atomic*, or a non-logical symbol at its head, in which case it is *atomic*.

2.5 Sequents

We shall not attempt to define completely the notion of sequent, inference rule, and proof. Rather we outline of number of characteristics that we shall find common in the sequent calculi examined in this text.

Proof are seldom of a single formula but more generally of judgments that relate various formulas. Example judgments might be that the formula B follows from the assumptions in Γ or that one of the formulas in Δ is provable. Sequents are intended to collect together such formulas in such a judgment and to allow reasoning steps to be applied to formulas within a surrounding context. Typically, sequents are constructed in many ways: we outline here the few major differences in sequents that we shall study here.

Sequents will contain the special symbol $\vdash$. Collections of formulas in sequents will be either lists or multisets or sets. Sequents can also be *one-sided* or *two-sided*. One-sided sequents are usually written as $\vdash \Delta$ and two-sided sequents are usually written as $\Gamma \vdash \Delta$, where Γ and Δ are one of the three kinds of collections of formulas mentioned above. Sometimes we shall see multiple collections of formulas, separated by a semicolon, on both the left and right sides of sequents; for example, $\Gamma; \Gamma \vdash \Delta; \Delta'$ and $\vdash \Delta; \Delta'; \Delta$. In the two-sided sequent $\Sigma\colon \Gamma \vdash \Delta$, we shall say that Γ is this sequent's *antecedent* or *left-hand side* and that Δ is its *succedent* or *right-hand side*.

When sequents are used for quantificational logic, they will also have a signature prefixing the sequent, such as, $\Sigma\colon \vdash \Delta$ and $\Sigma\colon \Gamma \vdash \Delta$ in order to declare certain symbols appearing in quantificational sequents (usually, so called, eigenvariables). Sequents will also satisfy the following property with respect to any prefixed signature: If Σ_L is the signature declaring a logical constants and Σ_C is the signature declaring non-logical constants, then all formulas in any list or multiset or set in a sequent prefixed with Σ will be a $\Sigma_L \cup \Sigma_C \cup \Sigma$-formula.

When presenting a particular notion of sequents, say, for example $\Sigma; \Gamma \vdash \Delta$, we will specify that Γ and Δ are either lists, multisets, or sets of formulas. In order to encode such objects into λ-expressions, we can do the following. First, introduce constructors for an empty collection, singleton collection, and union of collections. Enforcing various equalities on these constructors yield lists (associativity, identity), multisets (associativity, commutativity, identity), and sets (associativity, commutativity, idempotency, identity). The exact details of such an encoding are not particularly important here. We do note the following issues with respect to matching expressions with schematic variables. For example, let B denote a formula and let Γ and Γ' denote collections of formulas. Considering what it means to match the expression B, Γ' and Γ', Γ'' to a given collection, which we assume to contain $n \geq 0$ formulas.

If the given collection is a list, then B, Γ' matches if the list is non-empty and B is the first formula and Γ' is the remaining list. The expression Γ', Γ'' matches if

Γ' is some prefix and Γ'' is the remaining suffix of that list: there are $n+1$ possible matches.

If the given collection is a multiset then B, Γ' matches if the multiset is non-empty and B is a formula in the multiset and Γ' is the multiset resulting in deleting one occurrence of B. The expression Γ', Γ'' matches if the multiset union of Γ' and Γ'' is Γ: there can be as many as 2^n possible matches since each member of Γ can be placed in either Γ' or Γ''.

If the given collection is a set then B, Γ' matches if the set is non-empty and B is a formula in the set and Γ' is either the given set or the set resulting from removing B from the set. The expression Γ', Γ'' matches if the set union of Γ' and Γ'' is Γ: there can be as many as 3^n possible matches, since each member of Γ can be placed in either Γ' or Γ'' or in both.

2.6 Inference rules

Inference rules will have a single sequent as a conclusion and zero or more sequents as premises. Of the numerous inference rules used in present various sequent calculus, three board classes of rules can be identified. These are the *structural rules*, the *identity rules*, and the *introduction rules*.

Since sequents describe relationships among formulas, the natural of a context in which a formula is located is an important element of that formula's meaning and role in proof. In order to analyze in detail the interplay between a formula and its context, it is sometimes desirable to not hide the structural differences between lists, multisets, and sets within some equality theory for the constructors of such collections, as described in the preceding section. Instead, one might assume that inference rules are used to permute items in a context or to replace two occurrences of the same formula with one occurrences. There are three common structural rules, called *exchange*, *contraction*, and *weakening* and they are illustrated in Figure 2.2 in both left and right side versions.

The exchange rules, xL and xR, can be used on lists to exchange any two consecutive elements: this structural rule does not modify a multisets or sets context. The contraction rules, cL and cR, can be used on lists and multisets to replace two occurrences of the same formula with one occurrence: this structural rule does not modify sets context (hence, it does not seem sensible to use an explicit contraction rule when the context is a set). The weakening rules, wL and wR, can be used to insert a formula into a context. If used with a list, these rule inserts the new occurrence only at the end of context: it is simple to write a version of the weakening rules that allow for inserting a formula into any position of the list.

$$\frac{\Sigma\colon \Gamma', B, C, \Gamma'' \vdash \Delta}{\Sigma\colon \Gamma', C, B, \Gamma'' \vdash \Delta}\ xL \qquad \frac{\Sigma\colon \Gamma \vdash \Delta', B, C, \Delta''}{\Sigma\colon \Gamma \vdash \Delta', C, B, \Delta''}\ xR$$

$$\frac{\Sigma\colon \Gamma, B, B \vdash \Delta}{\Sigma\colon \Gamma, B \vdash \Delta}\ cL \qquad \frac{\Sigma\colon \Gamma \vdash \Delta, B, B}{\Sigma\colon \Gamma \vdash \Delta, B}\ cR$$

$$\frac{\Sigma\colon \Gamma \vdash \Delta}{\Sigma\colon \Gamma, B \vdash \Delta}\ wL \qquad \frac{\Sigma\colon \Gamma \vdash \Delta}{\Sigma\colon \Gamma \vdash \Delta, B}\ wR$$

Fig. 2.2. Structural rules.

Exercise 5. Let Δ' be a permutation of the list Δ. Show that a sequence of xR rules can transform the sequent $\Sigma\colon \Gamma \vdash \Delta$ into the sequent $\Sigma\colon \Gamma \vdash \Delta'$.

The group of identity rules generally contains the *initial* and the *cut* rules, examples of which are displayed in Figure 2.3. Whereas the structural rules imply properties of the contexts used in forming sequents, the cut and initial rules explain the meaning of the sequent symbol $\vdash$. In particular, these two rules can be seen as stating that $\vdash$ is reflexive and transitive. It is possible to see these two rules as describing dual aspects of $\vdash$, although this is easier to see when more declarative presentations of sequent calculus is are used. Notice also that these rules contain repeated occurrences of schema variables: in the initial rule, the variable B is repeated in the conclusion and in the cut rule, the variable B is repeated in the premise.

It might be natural to refer to an inference rule with zero premises as an "axiom". We shall not do this here since the term "axiom" usually refers to a formula that is accepted as starting point. Since sequents are not formulas, we use other names for such starting points of sequent calculus proofs.

$$\frac{}{\Sigma\colon B \vdash B}\ \mathit{init} \qquad \frac{\Sigma\colon \Gamma_1 \vdash \Delta_1, B \qquad \Sigma\colon B, \Gamma_2 \vdash \Delta_2}{\Sigma\colon \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Fig. 2.3. Example of the identity rules: initial and cut.

When an inference rule has two premises, there are two general and natural ways to relate the contexts in the two premises with the context in the conclusion. An inference rules is *multiplicative* if contexts in the premises are merged to form the context in the conclusion. The cut rule illustrated above is an example of a *multiplicative* rule. A rule is *additive* if the contexts for both premises and the conclusion are equal. An additive version of the cut inference rule can be written as

$$\frac{\Sigma\colon \Gamma \vdash \Delta, B \qquad \Sigma\colon B, \Gamma \vdash \Delta}{\Sigma\colon \Gamma \vdash \Delta}$$

The final group of inference rules that we highlight here are the *introduction* rules, so called because they introduce one occurrence of a logical connective into the conclusion of the inference rule. Usually, a logical connective is introduced two ways. If the sequents employed are two-sided, then there is usually a *left-introduction* rule that introduces the new occurrence of the connective into a context on the left and a *right-introduction* rule that introduces the new occurrence of the connective into a context on the right of the $\vdash$. If the sequent is one-sided, then the corresponding left-introduction rule is usually replaced by a right-introduction for the connective that is its de Morgan dual (if it has one). Figure 2.4 presents a few examples of introduction rules for some logical connectives.

$$\frac{\Sigma\colon B, \Gamma \vdash \Delta}{\Sigma\colon B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L} \qquad \frac{\Sigma\colon C, \Gamma \vdash \Delta}{\Sigma\colon B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L}$$

$$\frac{\Sigma\colon \Gamma \vdash \Delta, B \qquad \Sigma\colon \Gamma \vdash \Delta, C}{\Sigma\colon \Gamma \vdash \Delta, B \wedge C}\ \wedge\mathrm{R}$$

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

$$\frac{\Sigma \vdash t\colon \tau \qquad \Sigma\colon \Gamma, B[t/x] \vdash \Delta}{\Sigma\colon \Gamma, \forall_\tau x\ B \vdash \Delta}\ \forall\mathrm{L} \qquad \frac{\Sigma, y\colon \tau\ :\ \Gamma \vdash \Delta, B[y/x]}{\Sigma\colon \Gamma \vdash \Delta, \forall_\tau x\ B}\ \forall\mathrm{R}$$

Fig. 2.4. Examples of left and right introduction rules.

Notice that conjunction is given two left introduction rules and one right introduction rules: this last rule is an example of an additive inference rule. Implication is

given one left and one right introduction rule: the left introduction rule is an example of a multiplicative rule.

The signature Σ plays a direct role in the specification of the quantifier rules. In particular, the introduction of the universal quantifier $\forall$ in the left uses the signature to determine which are appropriate substitution terms for the quantifier. The right introduction rule for $\forall$ changes the signature from $\Sigma \cup \{c\colon \tau\}$ above the line to Σ below the line. Notice that if we were to think of signatures as lists of pairs containing distinct variable names, then we must maintain that the symbol y is not free in any formula in the conclusion of the rule. If we think of signatures as binding structures within a sequent, then we view the $\forall$R as specifying that a sequent-level binding (namely, for y) can move to a formula-level binding (namely, for x). Such a sequent-level bound variable is generally called an *eigenvariable*. By viewing quantifiers as bindings in formulas and signatures as binders for sequents, then the inference rule $\forall R$ essentially effects the *mobility* of a binder: reading this proof down, a binder *moves* from the sequent level (the binder for y) to the formula level (the binder for x). At no point is the binder replaced with a "free variable". Of course, this movement of the binder is only allowed if no occurrences of the bound variable above the line are unbound below the line: thus all occurrences of y in the upper sequent must appear in the displayed occurrence of $B[y/x]$.

The premise $\Sigma \blacksquare\!\!\!\vdash t\colon \tau$ for the $\forall$L rule should actually be written as $\Sigma \cup \Sigma_C \cup \Sigma_L \blacksquare\!\!\!\vdash t\colon \tau$ where Σ_L and Σ_C are the signatures for the logical and non-logical constants, respectively. We shall choose to write this condition with the smaller signature for convenience. Also, one has the choice whether or not this typing judgment is used as a formal part of the proof (hence, the proof of the typing judgment is a subproof of a proof of the conclusion to this rule) or as a side condition, namely, the requirement that premise is provable (in this case, the proof of that fact is not incorporated into the sequent proof).

Exercise 6. Write the multiplicative version of the $\wedge$R rule and the additive version of the $\supset$L rule. Assume that both the left and right side contexts are multisets. Show that additive and multiplicative rules can be derived from one another if weakening and contraction structural rules are used.

2.7 Sequent calculus proofs

Derivations and proofs will not formally be encoded as untyped λ-expressions (as introduced in Section 2.1). This is largely because at the level of proof the nature of abstraction does not need to play an important role and, hence, we shall make use the simpler notion of labeled trees to represent these structures. This choice is in contrast to the usual Curry-Howard Isomorphism approach to encoding natural deduction proofs as (typed) λ-expressions. The vocabulary associated to labeled trees (subtree, leaf, etc) seems a bit more natural here than that for terms (sub-term, free variable, etc).

Assume that a signature of logical constant Σ_L is given and that a collection of inference rules are specified. Let $\mathcal{S}$ be a sequent.

Derivations and proofs will be represented by finite trees with labeled nodes and edges, containing at least one edge. Nodes are labeled by occurrences of inference rules or by two *improper rules*, *open* and *root*. All trees contain exactly one node labeled *root*, called the *root node*. Let N be another node in the tree. The edge leading from N to the root node is called its *out-arc* while the other $n \geq 0$ arcs terminating at N are called its *in-arcs*: in this case, n is the *in-degree* of the node N. If N is labeled with *open*, then N must have zero in-arcs. If N is labeled by an occurrence of a proper inference rule, the out-arc must be labeled with the conclusion of the inference

rule occurrence and the in-arcs must be labeled with the premise sequents. Of course, sequent labels are determined to be equal using the rules of λ-expression.

A *derivation for* $\mathcal{S}$ is such a labeled tree in which the in-arc to the root is labeled with $\mathcal{S}$. The smallest derivation for $\mathcal{S}$ is a tree with two nodes, one labeled with *root* and one labeled with *open* and with the edge between them labeled with $\mathcal{S}$. A derivation for $\mathcal{S}$ without any nodes labeled *open* is a *proof of* $\mathcal{S}$. In these cases, the sequent $\mathcal{S}$ is also called the *endsequent* of the derivation or the proof.

When we write derivation trees: leaves with no line over them are taken as ending in an open node. If there is a line, then we assume that there are zero premises: in other words, the tree ends with a proper inference rule that has an in-degree of zero.

Given a particular sequent calculus proof system, say $\mathcal{X}$, we shall write $\Sigma\colon \Gamma \vdash_{\mathcal{X}} \Delta$ to denote the fact that the sequent $\Sigma\colon \Gamma \vdash \Delta$ has a proof in $\mathcal{X}$. If Σ is empty, we write just $\Gamma \vdash_{\mathcal{X}} \Delta$. If Γ is also empty, we write $\vdash_{\mathcal{X}} \Delta$. If the proof system is assumed, the subscript $\mathcal{X}$ is not written. Thus, $\vdash \Delta$ will mean the sequent $\cdot\colon\cdot \vdash \Delta$ is provable.

Exercise 7. Let Ξ be a sequent calculus proof containing just cut and initial inference rules. Show that all cuts can be removed to yield a proof of the same endsequent. Describe what can be proved using only initial and cut.

2.8 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.

Assume that the following three inference rules are part of the presentation of a logic. Here, the left and right hand contexts are assumed to be multisets.

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

$$\frac{\Sigma\colon B, \Gamma \vdash \Delta \qquad \Sigma\colon C, \Gamma \vdash \Delta}{\Sigma\colon B \vee C, \Gamma \vdash \Delta}\ \vee\mathrm{L}$$

Notice that the $\supset$L rule is given in its multiplicative form and the $\vee$L rule is given in its additive form. Now consider the following small derivation.

$$\frac{\dfrac{\Sigma\colon \Gamma, p, r \vdash s, \Delta \qquad \Sigma\colon \Gamma, q, r \vdash s, \Delta}{\Sigma\colon \Gamma, p \vee q, r \vdash s, \Delta}\ \vee\mathrm{L}}{\Sigma\colon \Gamma, p \vee q \vdash r \supset s, \Delta}\ \supset\mathrm{R}$$

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

$$\frac{\dfrac{\Sigma\colon \Gamma, p, r \vdash s, \Delta}{\Sigma\colon \Gamma, p \vdash r \supset s, \Delta}\ \supset\mathrm{R} \qquad \dfrac{\Sigma\colon \Gamma, q, r \vdash s, \Delta}{\Sigma\colon \Gamma, q \vdash r \supset s, \Delta}\ \supset\mathrm{R}}{\Sigma\colon \Gamma, p \vee q \vdash r \supset s, \Delta}\ \vee\mathrm{L}$$

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 derivation containing two inference rules.

$$\frac{\dfrac{\Sigma\colon \Gamma_1, r \vdash \Delta_1, p \qquad \Sigma\colon \Gamma_2, q \vdash \Delta_2, s}{\Sigma\colon \Gamma_1, \Gamma_2, p \supset q, r \vdash \Delta_1, \Delta_2, s}\ \supset\mathrm{L}}{\Sigma\colon \Gamma_1, \Gamma_2, p \supset q \vdash \Delta_1, \Delta_2, r \supset s}\ \supset\mathrm{R}$$

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

$$\dfrac{\dfrac{\dfrac{\Sigma : \Gamma_1, r \vdash \Delta_1, p}{\Sigma : \Gamma_1, r \vdash \Delta_1, p, s}\, wR}{\Sigma : \Gamma_1 \vdash \Delta_1, p, r \supset s}\, \supset\mathrm{R} \qquad \dfrac{\dfrac{\Sigma : \Gamma_2, q \vdash \Delta_2, s}{\Sigma : \Gamma_2, q, r \vdash \Delta_2, s}\, wL}{\Sigma : \Gamma_2, q \vdash \Delta_2, r \supset s}\, \supset\mathrm{R}}{\dfrac{\Sigma : \Gamma_1, \Gamma_2, p \supset q \vdash \Delta_1, \Delta_2, r \supset s, r \supset s}{\Sigma : \Gamma_1, \Gamma_2, p \supset q \vdash \Delta_1, \Delta_2, r \supset s}\, cR}\, \supset\mathrm{L}$$

If these additional structural rules are not available, then the original two inference rules cannot be permuted.

As we encounter inference rules for specific logics later, we will first observe what pairs of inference rules are permutable. Such information is central to the proof of cut-elimination as well as to the establishment of normal forms of proofs used to understand the nature of proof search.

2.9 Cut-elimination and its consequences

For most of the sequent proof systems we consider, the cut-elimination theorem holds: that is, a sequent has a proof if and only if it has a cut-free proof (a proof with no occurrences of the cut rule). This central theorem of sequent calculus proof systems has a number of consequences, some of which we list here.

The consistency of a logic is a simple consequence of cut-elimination. In particular, assume that the sequent $\cdot \vdash \bot$ has a proof. Since it has a cut-free proof, that proof must end in a structural rule (since there is usually no introduction rule for $\bot$ on the right). But the structural rules do not yield a provable sequent, so $\bot$ has no proof. In this case, what is explicitly ruled out by not having the cut rule is the possibility that there is some formula B such that B and $\neg B$ are provable: that is, that the sequents $\cdot \vdash B$ and $B \vdash \bot$ are provable.

The success of proving the cut-elimination theorem also signals that certain aspects of the logic's proof system were well designed in the sense that what the left-hand rule says about a logical connective is complementary to what the right hand rule says about the same logical connective.

When formulas involve only first-order quantification, a formula occurring in a sequent in a cut-free proof is always a subformula of some formula of the endsequent. This is the so-called *subformula property* of cut-free proofs. When searching for a proof, one needs only to choose and rearrange subformulas (which also means choosing instantiations of quantified expressions) in building a proof. In the higher-order setting, instantiating a predicate variable can result in larger formulas: thus, there is not a simple and meaningful notion of subformula property. Even in the higher-order setting, however, the cut elimination theorem can offer many useful structural properties about provability.

If one is attempting to prove theorems that might be mathematically interesting, one discovers that cut (also called *modus ponens*) actually serves as the main inference rule: it is a common activity when doing mathematically motivated proofs to state lemmas and invariants that are not simply subformulas of one's intended theorem and then to link them together by a chain of *modus ponens*. Eliminating cut in such a proof would necessarily yield a huge and low-level proof where all lemmas are "in-lined" and reproved at every instance of their use.

As we have seen, the fact that cuts can be eliminated from proofs is an important property of a proof system since it can imply that logic's consistency, for example.

Cut-free proofs can be huge objects. For example, if one uses the number of nodes in a proof as a measure of its size, there are cases where cut-free proofs are hyper-exponentially bigger than proofs allowing cut. Thus, sequents with proof of rather

small size can have cut-free proofs that require more inference rules than atomic particles in the universe. It is almost certainly the case that if a cut-free proof is actually computed and stored in some computer memory, the thing that that proof proves is almost certainly not *mathematically interesting*. This observation does not disturb us here since we are not interested in cut-free proofs as ways of describing computation traces only. For us, cut-free proofs are more akin to Turing machines configurations: that is, they provide a low-level and detailed history of a computation.

Recording a computation as a cut-free proof can be superior to, say, recording Turing machine configurations since proofs can be reasoned with in rather deep ways. For example, assume that we have a cut-free proof of the two-sided sequent $\mathcal{P} \vdash G$ for some logic, say, $\mathcal{X}$. As we shall see, in many approaches to proof search, it is natural to identify the left-hand context $\mathcal{P}$ to specification of a (logic) program and G as the goal or query to be established. A cut-free proof of such a sequent is then a trace that this goal can be established from this program. Now assume that we can prove $\mathcal{P}' \vdash^+ \mathcal{P}$ where $\mathcal{P}'$ is some other logic program and $\vdash^+$ is provability in $\mathcal{X}^+$ which is some strengthening of $\mathcal{X}$ in which, say, induction and/or co-induction principles are added (as well as cut). If the stronger logic satisfies cut-elimination, then we know that $\mathcal{P}' \vdash G$ has a cut-free proof in the stronger logic $\mathcal{X}^+$. If things have been organized well, it can then become a simple matter to see that cut-free proofs of such sequents do not, in fact, make use of the stronger proof principles, and, hence, $\mathcal{P} \vdash G$ has a cut-free proof in $\mathcal{X}$. Thus, using cut-elimination, we have been able to move from a *mathematical* proof about programs $\mathcal{P}$ and $\mathcal{P}'$ immediately to the conclusion that whatever goals can be established for $\mathcal{P}$ can be established for $\mathcal{P}'$. Clearly, the ability to do this kind of direct, logically principled reason about programs and their computations should be a central strength of the proof search paradigm for computing.

2.10 Additional readings

Texts for the lambda-calculus: Barendregt, Krivine, ...

The use of untyped λ-expressions is similar to the so-called "Curry-style" of typed λ-terms: bound variables are not assumed globally to have types but are provided a type when they are initially bound. This is in contrast to the Church-style approach where variables have types independently of whether or not they are bound.

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

Proofs of cut-elimination theorem can be found in various places. The original proof due to Gentzen [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.

The duality of the initial and cut rules is easily seen when one considers their presentation in the Calculus of Structures [Gug07] or when using linear logic as a meta-logic for sequent calculus [MP02, MP04]. In both of these cases, the formal dual of one of these inference rules is the other.

3

Classical and Intuitionistic Logics

3.1 First-order formulas

Formulas in both classical and intuitionistic first-order logic make use of the same set of logical connectives, namely, $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\top$ (truth), $\bot$ (absurdity), $\forall_\tau$ (universal quantification over type τ), and $\exists_\tau$ (existential quantification over type τ). The negation of B, written $\neg B$, is an abbreviation for the formula $B \supset \bot$. The logical constants have type o, the binary constants have type $o \to o \to o$, and the quantifiers $\forall_\tau$ and $\exists_\tau$ have type $(\tau \to o) \to o$.

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

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

This measure counts the number of times implications are nested to the left of implications. In particular, $\text{clausal}(\neg B) = \text{clausal}(B) + 1$. The clausal order of a finite set or multiset of formulas is the maximum clausal order of any formula in that set or multiset.

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 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 $B \supset B'$.

Signatures are used to introduce both non-logical constants and variables: the difference between a constant and variable is determined by their use: variables are tokens that can vary (by being instantiated by terms) while constants are tokens that do not vary.

3.2 Inference rules

To provide a modular presentation of provability in classical and intuitionistic logics, we shall use sequents of the form $\Sigma\colon \Delta \vdash \Gamma$, where Δ is a *set* of formulas and Γ is a multiset of formulas.

The rules for introducing the logical connectives are presented in Figure 3.1, the identity rules are given in Figure 3.2, and the structural rules are given in Figure 3.3. Since the left-hand side of sequents are sets, no left-hand side structural rules need to be presented.

$$\frac{\Sigma\colon B, \Gamma \vdash \Delta}{\Sigma\colon B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L} \qquad \frac{\Sigma\colon C, \Gamma \vdash \Delta}{\Sigma\colon B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L}$$

$$\frac{\Sigma\colon \Gamma \vdash \Delta, B \qquad \Sigma\colon \Gamma \vdash \Delta, C}{\Sigma\colon \Gamma \vdash \Delta, B \wedge C}\ \wedge\mathrm{R}$$

$$\frac{\Sigma\colon B, \Gamma \vdash \Delta \qquad \Sigma\colon C, \Gamma \vdash \Delta}{\Sigma\colon B \vee C, \Gamma \vdash \Delta}\ \vee\mathrm{L}$$

$$\frac{\Sigma\colon \Gamma \vdash \Delta, B}{\Sigma\colon \Gamma \vdash \Delta, B \vee C}\ \vee\mathrm{R} \qquad \frac{\Sigma\colon \Gamma \vdash \Delta, C}{\Sigma\colon \Gamma \vdash \Delta, B \vee C}\ \vee\mathrm{R}$$

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

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

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

$$\frac{}{\Sigma\colon \Gamma, \bot \vdash}\ \bot\mathrm{L} \qquad \frac{}{\Sigma\colon \Gamma \vdash \top}\ \top\mathrm{R}$$

Fig. 3.1. Introduction rules.

$$\frac{}{\Sigma\colon \Gamma, B \vdash B}\ \mathit{init} \qquad \frac{\Sigma\colon \Gamma \vdash \Delta_1, B \qquad \Sigma\colon B, \Gamma \vdash \Delta_2}{\Sigma\colon \Gamma \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Fig. 3.2. Identity rules.

$$\frac{\Sigma\colon \Gamma \vdash \Delta}{\Sigma\colon \Gamma \vdash \Delta, B}\ \mathit{wR} \qquad \frac{\Sigma\colon \Gamma \vdash \Delta, B, B}{\Sigma\colon \Gamma \vdash \Delta, B}\ \mathit{cR}$$

Fig. 3.3. Structural rules for the right-hand side only.

Of the four inference rules with two premises, $\supset$L and *cut* are multiplicative rules while $\wedge$R and $\vee$L are additive.

Provability in *classical logic* is given using the notion of a **C**-proof, which is any proof using inference rules in Figures 3.1, 3.2, and 3.3. Since both right structural rules are admitted in **C**-proofs, it is possible to give another presentation of classical logic in which both the left and right of the sequent are sets. Provability in *intuitionistic logic* is given using the notion of a **I**-proof, which is any **C**-proof in which the right-hand side of all sequents contain either 0 or 1 formula.

Let Σ be a given first-order signature over S, let Δ be a finite set of Σ-formulas, and let B be a Σ-formula. We write $\Sigma; \Delta \vdash_C B$ and $\Sigma; \Delta \vdash_I B$ if the sequent $\Sigma\colon \Delta \vdash B$ has, respectively, a **C**-proof or an **I**-proof.

Proposition 3.1 *If $\Sigma\colon \Delta \vdash \Gamma$ has an* **I***-proof then that proof does not contain occurrences of* cR *and only occurrences of* wR *of the form*

$$\frac{\Sigma\colon \Gamma \vdash}{\Sigma\colon \Gamma \vdash B}$$

The notion of provability defined here is not equivalent to the more usual presentations of classical and intuitionistic logic [Fit69, Gen69, Pra65, Tro73] in which signatures are not made explicit and substitution terms (the terms used in ∀L and ∃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

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

This sequent has no proof even though $\exists_i x\,(px)$ follows from $\forall_i x\,(px)$ in the traditional presentations of classical and intuitionistic logics. The reason for this difference is that there are no $\{p\colon i \to o\}$-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.

Exercise 8. Provide proofs for each of the following sequents. Provide a **C**-proof only if there is no **I**-proof. Assume that the signature for non-logical constants is $\{p\colon o,\ q\colon o,\ r\colon i \to o,\ s\colon i \to i \to o,\ a\colon i,\ b\colon i\}$.

1. $p \wedge (p \supset q) \wedge (p \wedge q \supset r) \supset r$
2. $(p \supset q) \supset (\neg q \supset \neg p)$
3. $(\neg q \supset \neg p) \supset (p \supset q)$
4. $p \vee (p \supset q)$
5. $(r\ a \wedge r\ b \supset q) \supset \exists x(r\ x \supset q)$
6. $((p \supset q) \supset p) \supset p$
7. $\exists y \forall x(r\ x \supset r\ y)$
8. $\forall x \forall y(s\ x\ y) \supset \forall z(s\ z\ z)$

Exercise 9. A formula of the form $B \vee \neg B$ is an example of an *excluded middle*: B is either true or false, and any third possibility is excluded. Clearly there is a simple **C**-proof for any formula of this kind. Take the formulas in Exercise 8 which have **C**-proofs but no **I**-proof and reorganized them into **I**-proofs in which appropriate instances of an excluded middle formula are added to the left-hand context. For example, show that the sequent

$$\Sigma\ :\ r\ a \vee \neg r\ a\ \vdash (r\ a \wedge r\ b \supset q) \supset \exists x(r\ x \supset q)$$

has an **I**-proof. Of course, to remove this additional assumption, cut with a **C**-proof is needed. (Here, Σ is given in Exercise 8.)

Exercise 10. Assume that the set of sorts S contains the two tokens i and j and that the only non-logical constant is $f\colon i \to j$. In particular, assume that there are no constants of type i declared in the non-logical signature. Is there an **I**-proof of

$$(\exists_j x \top) \vee (\forall_i y \exists_j x \top).$$

Under the same assumption, does the formula

$$(\exists_j x \top) \vee (\forall_i x\ \bot)$$

have a **C**-proof? An **I**-proof? Compare the issue of provability for this formula with the one in Exercise 8(4).

Exercise 11. The multiplicative version of ∧R would be the inference rule

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

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

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

Exercise 12. Consider adding the following rule

$$\frac{\Sigma\colon \Gamma \vdash B}{\Sigma\colon \Gamma \vdash C}\ \mathit{Restart}$$

to **I**-proofs along with the following proviso on how it is used in a proof: on the path from an occurrence of this rule to the root of the proof, there is a sequent that contains B in the succedent. Prove that a formula has a **C**-proof if and only if it has an **I**-proof with the Restart rule.

Exercise 13. Show that if we consider **C**-proofs, then all pairs of inference rules for propositional connectives (i.e., excluding the quantifiers) permute.

Exercise 14. Not all pairs of quantification introduction rules permute. Present those pairs of inference rules that do not permute.

Exercise 15. Let A be an atomic formula. Describe all pairs of formulas $\langle B, C\rangle$ where B and C are different members of the set

$$\{A, \neg A, \neg\neg A, \neg\neg\neg A\}$$

such that $B \vdash C$ has a **C**-proof. Make the same list such that $B \vdash C$ has an **I**-proof.

Exercise 16. Let Ξ be a proof of $\Sigma\colon \Gamma \vdash \Delta$ and let Γ' be a set of Σ-formulas and Δ' be a multiset of Σ-formulas. Show that if Ξ is a **C**-proof, then the result of adding Γ' to all antecedents of every sequent in Ξ and adding Δ' to all succedents of every sequent in Ξ yields a **C**-proof of $\Sigma\colon \Gamma, \Gamma' \vdash \Delta, \Delta'$. Furthermore, if Ξ is also an **I**-proof and Δ' is empty, then the resulting proof is an **I**-proof.

Exercise 17. Let Ξ be a **C**-proof (resp., **I**-proof) of $\Sigma, x\colon \Gamma \vdash \Delta$ and let t be a Σ-term. The result of substituting t for the bound variable x in this sequent and all the bound variables corresponding to x is all other sequents in Ξ yields a **C**-proof (resp., **I**-proof) Ξ' of the sequent $\Sigma\colon \Gamma[t/x] \vdash \Delta[t/x]$. The arrangement of inference rules in Ξ and in Ξ' are the same.

3.3 The initial rule

An occurrence of the initial rule of the form $\Sigma\colon \Gamma, B \vdash B$ is an *atomic initial* if B is an atomic formula. In classical and intuitionistic logic, we can restrict the initial rule to be atomic initial rules only.

Proposition 3.2 *If a sequent has a* **C***-proof (resp, an* **I***-proof) then it has a* **C***-proof (resp, an* **I***-proof) in which all occurrence of the* init *rule are atomic initial rules.*

Proof. A simple induction on the structure of B shows that the sequent $\Sigma\colon \Gamma, B \vdash B$ can be proved by a cut-free proof involving only atomic initial rules.

The fact that the initial rules involving non-atomic formulas can be replaced by introduction rules and initial rules on subformulas is an important and desirable property of a proof system. In general, however, atomic initial rules cannot be removed from proofs. Atoms are built from non-logical constants, such as predicates and function systems, and their meaning comes from outside logic. In particular, it is via non-logical symbols and atomic formulas that we shall eventually specify *logic programs* for the purpose of sorting list, representing transition systems, etc. Atoms provide the plugs for the programmer to provide their own meaning.

Consider defining a third logic, usually called *minimal logic*, as follows: an **M**-proof is any **I**-proof in which the right-hand side of all sequents contains exactly one formula. We shall write $\Sigma; \Delta \vdash_M B$ if the sequent $\Sigma\colon \Delta \vdash B$ has an **M**-proof.

Exercise 18. Show that Proposition 3.2 does not hold for minimal logic (consider the sequent $\bot \vdash \bot$).

The reason for $\bot$ to lose this important proof theoretic properties is that weakening for $\bot$ on the right *is* the proper treatment for $\bot$ on the right. As the following exercise shows, $\bot$ is not a real logical connective in minimal logic.

Exercise 19. Let q be a non-logical symbol of type o, let B be a formula, and let B' be the result of replacing all occurrences of q in B with $\bot$. Show that B is provable in intuitionistic logic if and only if B' is provable in minimal logic.

3.4 The cut rule

The cut rule can also be restricted to atomic formulas in a manner similar to that for restricting the initial rule to atomic formulas. For example, consider a proof which contains the following cut with a conjunctive formula in which the two occurrences of that conjunction are immediately introduced in the two subproofs to cut.

$$\dfrac{\dfrac{\overset{\Xi_1}{\Sigma\colon \Gamma_1 \vdash A_1, \Delta_1} \quad \overset{\Xi_2}{\Sigma\colon \Gamma_1 \vdash A_2, \Delta_1}}{\Sigma\colon \Gamma_1 \vdash A_1 \wedge A_2, \Delta_1}\,\wedge\mathrm{R} \quad \dfrac{\overset{\Xi_3}{\Sigma\colon \Gamma_2, A_i \vdash \Delta_2}}{\Sigma\colon \Gamma_2, A_1 \wedge A_2 \vdash \Delta_2}\,\wedge\mathrm{L}}{\Sigma\colon \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\,cut$$

Here, i is either 1 or 2. This part of the proof can be changed locally to

$$\dfrac{\overset{\Xi_i}{\Sigma\colon \Gamma_1 \vdash A_i, \Delta_1} \quad \overset{\Xi_3}{\Sigma\colon \Gamma_2, A_i \vdash \Delta_2}}{\Sigma\colon \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\,cut$$

In the process of reorganizing the proof in this manner, one of the subproofs Ξ_1 and Ξ_2 is discarded and the new occurrence of cut is on a subformula of $A_1 \wedge A_2$.

Consider a proof which contains the following cut with an implication in which the two occurrences of that implication are immediately introduced in the two subproofs to cut.

$$\dfrac{\dfrac{\overset{\Xi_1}{\Sigma\colon \Gamma_1, A_1 \vdash A_2, \Delta_1}}{\Sigma\colon \Gamma_1 \vdash A_1 \supset A_2, \Delta_1}\,\supset\mathrm{R} \quad \dfrac{\overset{\Xi_2}{\Sigma\colon \Gamma_2 \vdash A_1, \Delta_2} \quad \overset{\Xi_3}{\Sigma\colon \Gamma_3, A_2 \vdash \Delta_3}}{\Sigma\colon \Gamma_2, \Gamma_3, A_1 \supset A_2 \vdash \Delta_2, \Delta_3}\,\supset\mathrm{L}}{\Sigma\colon \Gamma_1, \Gamma_2, \Gamma_3 \vdash \Delta_1, \Delta_2, \Delta_3}\,cut$$

This part of the proof can be changed locally to

$$\dfrac{\dfrac{\overset{\Xi_2}{\Sigma\colon \Gamma_2 \vdash A_1, \Delta_2} \quad \overset{\Xi_1}{\Sigma\colon \Gamma_1, A_1 \vdash A_2, \Delta_1}}{\Sigma\colon \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A_2}\,cut \quad \overset{\Xi_3}{\Sigma\colon \Gamma_3, A_2 \vdash \Delta_3}}{\Sigma\colon \Gamma_1, \Gamma_2, \Gamma_3 \vdash \Delta_1, \Delta_2, \Delta_3}\,cut$$

In the process of reorganizing the proof in this manner, the cut rule occurrence for $A_1 \supset A_2$ is replaced by two instances of cut, where each cut is on the subformula of A_1 and A_2.

Consider a proof that contains the following cut with $\top$ in which the premise where $\top$ is on the right-hand side is proved with the $\top$R.

$$\dfrac{\dfrac{}{\Sigma\colon \Gamma_1 \vdash \top, \Delta_1}\ \top\mathrm{R} \qquad \Sigma\colon \Gamma_2, \overset{\Xi}{\top} \vdash \Delta_2}{\Sigma\colon \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

This proof can be changed to remove this occurrence of cut entirely as follows. First, the proof Ξ of $\Sigma\colon \Gamma_2, \top \vdash \Delta_2$ can be transformed to a proof Ξ' of $\Sigma\colon \Gamma_2 \vdash \Delta_2$ by removing the occurrence of $\top$ in the endsequent and, hence, all the other occurrences of $\top$ that can be traced to that occurrence. As a result of Exercise 16, Ξ' can be transformed to a proof Ξ'' of $\Sigma\colon \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2$. The proof Ξ'' contains one fewer instances of the cut-rule than the original displayed proof above.

Consider a proof that contains the following cut with $\forall$ in which the two occurrences of that quantifier are immediately introduced in the two subproofs to cut.

$$\dfrac{\dfrac{\overset{\Xi_1}{\Sigma, x\colon \Gamma_1 \vdash Bx, \Delta_1}}{\Sigma\colon \Gamma_1 \vdash \forall x.Bx, \Delta_1}\ \forall\mathrm{R} \qquad \dfrac{\overset{\Xi_2}{\Sigma\colon \Gamma_2, Bt \vdash \Delta_2}}{\Sigma\colon \Gamma_2, \forall x.Bx \vdash \Delta_2}\ \forall\mathrm{L}}{\Sigma\colon \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Here, t is a Σ-term. By Exercise 17, the proof Ξ_1 of $\Sigma, x\colon \Gamma_1 \vdash Bx, \Delta_1$ can be transformed into a proof Ξ_1' of $\Sigma\colon \Gamma_1 \vdash Bt, \Delta_1$ (notice that x is not free in any formula of Γ_1 and Δ_1 nor in the abstraction B). The above instance of cut can now be rewritten as

$$\dfrac{\overset{\Xi_1'}{\Sigma\colon \Gamma_1 \vdash Bt, \Delta_1} \qquad \overset{\Xi_2}{\Sigma\colon \Gamma_2, Bt \vdash \Delta_2}}{\Sigma\colon \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Exercise 20. Repeat the above rewriting of cut inference rules when the cut formula is $\bot$, a disjunction, or an existential quantifier.

The above rewriting suggests that each of the logical connectives, in isolation, have been designed well. Each logical connective is given two senses: introduction on the right provides the means to prove a logical connective; introduction on the left provides the means to argue from a logical connective as an assumption. The rewritings above provides a partial justification that these two means are describing the same connective. Of course, we are interested to see if all cuts can be removed.

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

For the details of the proof, see, for example, [Gen69], [GTL89, Chapter 13], [Gal86, Chapter 6].

Exercise 21. Define a new binary logical connective, written $\diamond$, giving it the left introduction rules for $\wedge$ but the right introduction rules for $\vee$. Can cut be eliminated from proofs involving $\diamond$? Can *init* be restricted to only atomic formulas? This connective is the "tonk" connective of Prior [Pri60].

3.5 Choices when doing proof search

Since we will be considering the use of proof search to support computation, we should look carefully at the many choices that are available in building a proof (in a bottom-up fashion) and look for possible means to reduce those choices for automation even if

those choices make logic less amendable for mathematical (i.e., not automated) proof. We characterize the many choices in how one searches for proofs as follows.

- It is always possible to use the cut rule to prove any sequent. In that case, we need to produce a cut-formula (lemma) to be proved on one branch and to be used on the other.
- The structural rules of contractions and weakening can always be applied to make additional copies of a formula or to remove formulas.
- There may be many non-atomic formulas in a sequent and we can generally apply an introduction rule for every one of these formulas.
- One can also see if a given sequent is initial.

Some of these choices produce sub-choices. For example, choosing the cut rule requires finding a cut-formula; choosing $\vee$R requires selecting a disjunct; choosing $\wedge$L requires selecting a conjunct; 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 set Γ and multiset Δ into pairs (for which there are exponentially many splits).

All this freedom in searching for proofs is not, however, needed, and greatly reducing the sets of choices can still result in complete proof procedures. Many of these choices can be dealt with 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 arguments. But since our goal here is the specification of computation, we shall generally live inside this choice.
- Often, structural rules can be built into inference rules. For example, weakening on the left is built into the *init* rule. Also, instead of attempting to split 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. Equivalently, we can try to understand when the additive version of this rule can replace the multiplicative version, in which case, contexts are copied and not split.
- The problem of determining appropriate substitution terms in the $\forall$L and $\exists$R rules is a serious problem whose solution falls outside our investigations here. When systems based on proof search are implemented, they generally make use of various techniques, such as employing the so-called "logic variable" and unification to determine instantiation terms in a lazy fashion. Although such techniques are completely standard, we shall not discuss them here.
- The choices between which introduction rule to select can be structured by first noticing that some introduction rules are *invertible*: that is, their premises are provable if and only if their conclusion is provable. Thus, applying such introduction rules does not lose completeness. While non-invertible introduction rules represent genuine choices in the search for proofs, some structure to how these rules are applied can also be described (see, for example, backchaining in Section 4.4).

3.6 Dynamics and change during of proof search

Within the proof search paradigm, changes to sequents during search represents the dynamics of search. Thus it is important to understand what kinds of dynamics are supported by a given logic.

The following exercises illustrate to what extent sequents can change within classical and intuitionistic logics.

Exercise 22. Show that a cut-free **C**-proof Ξ of $\Sigma: \Gamma \vdash \Delta$ can be transformed into a proof Ξ' of the same sequent such that for every sequent $\Sigma': \Gamma' \vdash \Delta'$ in Ξ', we

have that $\Sigma \subseteq \Sigma'$, $\Gamma \subseteq \Gamma'$, and $\Delta \subseteq \Delta'$. (For this exercise, assume that the initial sequents allowed for **C**-proofs are of the form $\Sigma: \Gamma \vdash \Delta$ where $\Gamma \cap \Delta$ is non-empty.) Furthermore, let $n \geq 0$ and assume that every formula in Γ is of clausal order n or less and every formula in Δ is of clausal order $n-1$ or less. Then we can also assume that the clausal order of formulas in Δ' are $n-1$ or less and that Γ' is the set union of Γ and a set Γ'' and that the formulas in Γ'' have clausal orders of order $n-2$ or less.

Exercise 23. Show that a cut-free **I**-proof Ξ of $\Sigma: \Gamma \vdash \Delta$ can be transformed into a proof Ξ' of the same sequent such that for every sequent $\Sigma': \Gamma' \vdash \Delta'$ in Ξ', we have that $\Sigma \subseteq \Sigma'$ and $\Gamma \subseteq \Gamma'$. Furthermore, let $n \geq 0$ and assume that every formula in Γ is of clausal order n or less and every formula in Δ is of clausal order $n-1$ or less. Then we can also assume that the clausal order of formulas in Δ' are $n-1$ or less and that Γ' is the set union of Γ and a set Γ'' and that the formulas in Γ'' have clausal orders of order $n-2$ or less.

The dynamics of classical logic seems quite weak in the sense that contexts only grow during proof search. No formulas are required to be forgotten or dropped. Since the semantics of classical logic are based on notions of "static" truth, this proof search characterization of classical logic seems appropriate.

Intuitionistic logic has a bit richer dynamics since the antecedents of sequents can change significantly since only one formula is keep in the succecedent: contrary to classical logic, antecedent formulas cannot be copied (using cR) and restarted (Exercise 12). Thus, intuitionistic logic allows for growing antecedents and more complicated varying succedents. As we shall see in the next chapter, if we are in a setting where goal-directed proof search is complete, the dynamics of the succedent is reduced to the changing of one atomic formula with another. Thus, most of this dynamics occurs within *non-logical* context by changes to the terms within atomic formulas. Constraining such dynamics to non-logical contexts means that logical reasoning will provide little immediate help in reasoning about computational dynamics.

4

Horn and hereditary Harrop formulas

4.1 Goal-directed search

One approach to modeling logic programming with sequent calculus involves seeing *logic programs* as theories from which deductions are attempted and *goals* (also called *queries*) are formulas whose entailment is attempted from logic programs. The state of an idealized interpreter can be represented as the two-sided sequent $\Sigma\colon \mathcal{P} \vdash G$, where Σ is the signature that declares current set of eigenvariables, $\mathcal{P}$ is a set of Σ-formulas denoting a program, and G is a Σ-formula denoting the goal we wish to prove from $\mathcal{P}$.

It also seems natural to also impose that computation should proceed in the following fashion: when given the program $\mathcal{P}$ and a non-atomic goal G, then the proof should proceed in a fixed fashion to decompose the goal formula G first and without regard to the program. Thus, the "search semantics" for a logical connective at the head of a goal is fixed by the logic and is independent of the program. It is only when attempting a proof of an atomic formula that the program is consulted so as to provide meaning for the non-logical predicate constant at the head of that atom. In particular, the following are completely natural reductions to attempts to prove a goal.

- Reduce an attempt to prove $\Sigma\colon \mathcal{P} \vdash B_1 \wedge B_2$ to the attempts to prove the two sequents $\Sigma\colon \mathcal{P} \vdash B_1$ and $\Sigma\colon \mathcal{P} \vdash B_2$.
- Reduce an attempt to prove $\Sigma\colon \mathcal{P} \vdash B_1 \vee B_2$ to an attempt to prove either $\Sigma\colon \mathcal{P} \vdash B_1$ or $\Sigma\colon \mathcal{P} \vdash B_2$.
- Reduce an attempt to prove $\Sigma\colon \mathcal{P} \vdash \exists_\tau x.B$ to an attempt to prove $\Sigma\colon \mathcal{P} \vdash B[t/x]$, for some Σ-term t of type τ.
- Reduce an attempt to prove $\Sigma\colon \mathcal{P} \vdash B_1 \supset B_2$ to an attempt to prove $\Sigma\colon \mathcal{P}, B_1 \vdash B_2$.
- Reduce an attempt to prove $\Sigma\colon \mathcal{P} \vdash \forall_\tau x.B$ to an attempt to prove $\Sigma, c\colon \tau\colon \mathcal{P} \vdash B[c/x]$, where c is token not in Σ.
- Attempting to prove $\Sigma\colon \mathcal{P} \vdash \top$ yields an immediate success.

Clearly, these reduction steps are the bottom-up readings of the right-introduction rules found in Figure 3.1. This suggests the following definition to formalize the notion of *goal-directed search*: a cut-free **I**-proof is *uniform* if every occurrence of a sequent whose succedent contains a non-atomic formula is the conclusion of an inference figure that introduces its top-level connective. Searching for uniform proofs is now greatly restricted since building a uniform proof means that one applies right-rules when the succedent has logical constants. We are not allowed to interleave choosing right and left introductions rules. The definition of uniform proof provides no guidance or possible restrictions for applying left-introduction rules, although such guidance will soon appear.

Exercise 24. Show that in a uniform proof, *init* inference rules are always atomic and that a sequent is the conclusion of a left rule only if that sequent has an atomic succedent.

There are provable sequents for which no uniform proof exists. For example, let the non-logical constants be $\{p : o, q : o, r : i \to o, a : i, b : i\}$ and let Σ be an signature. The sequents

$$\Sigma\colon (r\ a \wedge r\ b) \supset q \vdash \exists_i x(r\ x \supset q) \quad \text{and} \quad \Sigma\colon \vdash p \vee (p \supset q)$$

have **C**-proofs but no **I**-proofs, so clearly they have no uniform proofs. The two sequents

$$\Sigma\colon p \vee q \vdash q \vee p \quad \text{and} \quad \Sigma\colon \exists_i x.\ r\ x \vdash \exists_i x.\ r\ x$$

have **I**-proofs but no uniform proofs.

One way to define logic programming, at least from the point-of-view of logical connectives and quantifiers, is to consider those collections of programs and goals for which uniform proofs are, in fact, complete. In particular, an *abstract logic programming language* is a triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ such that for all signatures Σ, for all finite sets $\mathcal{P}$ of Σ-formulas from $\mathcal{D}$, and all Σ-formulas G of $\mathcal{G}$, we have $\vdash \Sigma\colon \mathcal{P} \vdash G$ if and only if $\Sigma\colon \mathcal{P} \vdash G$ has a uniform proof.

Both the definition of uniform proof and abstract logic programming language are restricted to **I**-proofs. We shall refer to this as the *single-conclusion* version of these notions. Later we present a generalization of them to the multiple conclusion setting.

A theory Δ is said to hold the *disjunction property* if the provability of $\Sigma\colon \Delta \vdash B \vee C$ implies the provability of either $\Sigma\colon \Delta \vdash B$ or $\Sigma\colon \Delta \vdash C$. A theory Δ is said to hold the *existence property* if the provability of $\Sigma\colon \Delta \vdash \exists_\tau x.\ B$ implies the existence of a Σ-term t of type τ such that $\Sigma\colon \Delta \vdash B[t/x]$ is provable. Clearly, if uniform proofs are complete for a given theory and notion of provability, that theory has both the disjunctive and existential properties. In a sense, when uniform proofs are complete, these properties are satisfied at all point in building a cut-free proof.

4.2 Horn clauses

The first approaches to describing the structure of proofs using Horn clauses were done using resolution refutations. In that setting, Horn clauses were generally defined as the universal closure of disjunctions of literals (atomic formulas or their negation) that contain at most one positive literal (an atomic formula). That is, a clause is a closed formula for the form

$$\forall x_1 \ldots \forall x_n[\neg A_1 \vee \cdots \vee \neg A_m \vee B_1 \vee \cdots \vee B_p],$$

where $n, m, p \geq 0$ and $p \leq 1$. If $n = 0$ then the quantifier prefix is not written and if $m = p = 0$ then the body of the clause is considered to be $\bot$. If the clause contained exactly one positive literal ($p = 1$), it is a *positive* Horn clause. If it contained no positive literal ($p = 0$), it is a *negative* Horn clause.

When we shift from the search for refutations to the search for sequent calculus proofs, it is natural to shift the presentation of Horn clauses to one of the following. Let τ be some member of S (primitive type) and let A be a syntactic variable ranging over atomic formulas. Consider the following three, separate and recursive definitions of the two syntactic categories of *program clauses* (*definite clause*) given by the syntactic variable D and *goals* given by the syntactic variable G.

$$\begin{aligned} G &::= \quad A \mid G \wedge G \\ D &::= A \mid G \supset A \mid \forall_\tau x\ D. \end{aligned} \tag{4.1}$$

Program clauses in this style presentation are formulas of the form

$$\forall x_1 \ldots \forall x_n (A_1 \wedge \cdots \wedge A_m \supset A_0),$$

where we adopt the convention that if $m = 0$ then the implication is not written. A second, richer definition of these syntactic classes is the following.

$$\begin{aligned} G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_\tau x\, G \\ D &::= \ \ A \mid G \supset D \mid D \wedge D \mid \forall_\tau x\, D. \end{aligned} \tag{4.2}$$

Finally, a compact presentation of Horn clauses and goals is possible using only implication and universal quantification.

$$\begin{aligned} G &::= \qquad\quad A \\ D &::= A \mid A \supset D \mid \forall_\tau x\, D. \end{aligned} \tag{4.3}$$

This last definition describes a Horn clause as a formula built from implications and universals such that to the left of an implication there are no occurrences of logical connectives.

Definition (4.1) above corresponds closely to the definition of Horn clauses given using disjunction of literals. In this case, positive clauses correspond to the D-formulas. Classical equivalences are needed (not intuitionistic equivalences). Negative clauses are not exactly negations of G formulas since such G formulas are not allowed to have existential quantifiers (although allowing such existential quantifiers are allowable, as is done in (4.2).

Let $\mathcal{D}_1$ be the set of D-formulas and $\mathcal{G}_1$ be the set of G-formulas satisfying the recursion (4.2).

Exercise 25. Given any of the three presentations of Horn clauses and goals above, show that the clausal order (see Section 3.1) of a Horn goal is always 0 and of a Horn clause is 0 or 1.

Exercise 26. Let D be a Horn clause using (4.2). Show that there is a set Δ of Horn clauses using description (4.1) or (4.3) such that D is equivalent to the conjunction of formulas in $\mathcal{D}$. Show that this rewriting might make the resulting conjunction exponentially larger than the original clause.

Exercise 27. Let Σ be a signature, let $\mathcal{P}$ be a set of Σ-formulas in $\mathcal{D}_1$, and let G be a Σ-formula in $\mathcal{G}_1$. Let Ξ be a cut-free **C**-proof of $\Sigma: \mathcal{P} \vdash G$. Show that every sequent in Ξ is of the form $\Sigma: \Gamma \vdash \Delta$ such that Γ is a subset of $\mathcal{D}_1$ and Δ is a subset of $\mathcal{G}_1$. Show also that the only inference rules that can appear in Ξ are cR, wR, *init*, $\forall$L, $\wedge$L, $\supset$L, $\wedge$R, $\vee$R, $\exists$R, and $\top$R.

Exercise 28. Prove that Horn clause programs are always consistent by proving that for any signature Σ and any finite set of Horn clauses $\mathcal{P}$, the sequent $\Sigma: \mathcal{P} \vdash$ is not provable. Show that an **I**-proof of $\Sigma: \mathcal{P} \vdash G$ for a Horn goal G is also an **M**-proof.

We first show that in the Horn clause setting, classical provability is conservative over intuitionistic logic.

Proposition 4.1 *Let Σ be a signature, let $\mathcal{P}$ be a set of Σ-formulas in $\mathcal{D}_1$, and let G be a Σ-formula in $\mathcal{G}_1$. If $\Sigma: \mathcal{P} \vdash G$ has a* **C***-proof then it has an* **I***-proof.*

Proof. We actually show the following stronger result: If $\Sigma: \mathcal{P} \vdash \Gamma$ has a cut-free **C**-proof then there is a $G \in \Gamma$ such that $\Sigma: \mathcal{P} \vdash G$ has an **I**-proof. We prove this by induction on the structure of **C**-proofs.

The three base cases are easy: $\bot$L is not possible since $\bot$ is not a member of $\mathcal{P}$ and the two other cases of $\top$R and *init* are immediate. If the last rule is the structural rule wR or cR then by induction, the formula selected from the succedent of the premise is also present in the conclusion and, thus, can be selected for the conclusion as well.

Now consider all possible introduction rules that might be the last inference rule (these are enumerated in Exercise 27). If that last rule is $\supset$L, then the proof has the form

$$\frac{\Sigma\colon \mathcal{P}_1 \vdash \Delta_1, G \qquad \Sigma\colon D, \mathcal{P}_2 \vdash \Delta_2}{\Sigma\colon G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash \Delta_1, \Delta_2}\ \supset\text{L},$$

By the induction assumption, there is a formula $H_1 \in \Delta_1 \cup \{G\}$ for which $\Sigma\colon \mathcal{P}_1 \vdash H_1$ has an **I**-proof and a formula $H_2 \in \Delta_2$ for which $\Sigma\colon D, \mathcal{P}_2 \vdash H_2$ has an **I**-proof. In the case that $H_1 \in \Delta_1$, the sequent $\Sigma\colon \mathcal{P}_1 \vdash H_1$ can be weakened to $\Sigma\colon G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash \Delta_1, \Delta_2$ using the result in Exercise 16 to add formulas to the antecedent. On the other hand, if $H_1 = G$, then we select from the multiset $\Delta_1 \cup \Delta_2$ the formula H_2 and build an **I**-proof using the following instance of the inference rule

$$\frac{\Sigma\colon \mathcal{P}_1 \vdash G \qquad \Sigma\colon D, \mathcal{P}_2 \vdash H_2}{\Sigma\colon G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash H_2}\ \supset\text{L},$$

and the two promised **I**-proofs of the premises.

All the remaining cases of introduction rules can be treated in a similar fashion.

Notice that Exercise 28 is an immediate consequence of the proof of Proposition 4.1.

Proposition 4.2 *Let Σ be a signature, let $\mathcal{P}$ be a set of Σ-formulas in $\mathcal{D}_1$, and let G be a Σ-formula in $\mathcal{G}_1$. If $\Sigma\colon \mathcal{P} \vdash G$ has a* **C***-proof then it has a uniform proof.*

Proof. By Proposition 4.1, if $\Sigma\colon \mathcal{P} \vdash G$ has a **C**-proof, it has an **I**-proof. Let Ξ be such an **I**-proof. By Proposition 3.2, we can also assume that the initial rules in Ξ are all atomic initial rules. If Ξ is not already a uniform proof, then there must be a left-introduction rule applied to a sequent with a non-atomic succedent. In this case, consider an occurrence of a left introduction rule that has a non-atomic right-hand side and which has premises with minimal height. At least one of the premises must be a right-introduction rule (the case of wR is ruled out by Exercise 28). Given the pairs of left and right introduction rules that can appear in Ξ, it is easy to show that all pairs of right-introduction rules over left-introduction rules permute and that this process of permuting terminates. In this way, the proof Ξ can be converted into a uniform proof of $\Sigma\colon \mathcal{P} \vdash G$.

$$\frac{}{G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash G_1 \wedge G_2}$$

The preceding propositions shows that the triple $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash \rangle$ is an abstract logic programming if $\vdash$ is taken to be $\vdash_C$, $\vdash_I$, or $\vdash_M$. That is, when dealing with Horn clauses, there is no separation between these three logics. Anyone of these three abstract logic programming languages will be called *fohc* (for *first-order Horn clauses*).

Note that *fohc* is a weak logic programming language in the proof theoretic sense that there are few logical connectives for which the right and left behavior exist within uniform proofs. If we use the (4.2) presentation of Horn clauses, then it is only atoms or conjunctions of atoms that are both goals and program clauses. All the other connectives are either dismissed (such as $\bot$) or are restricted to just half their "meaning": when a disjunction and existential quantifier are encountered in proof search, only their right introduction rules are needed and when implication and universal quantification are encountered, only their left introduction rules are needed.

4.3 Hereditary Harrop formulas

An extension to Horn clauses that allow implications and universal quantifiers in goals (and, thus, in the body of program clauses) is called the *first-order hereditary Harrop formulas*. Proof search involving such formulas may involve left and right introduction rules for implications and universal quantifiers as well as conjunctions (as in the Horn clause case). Parallel to the three presentations of *fohc* in Section 4.2, there are the following three presentations of goals and program clauses for first-order hereditary Harrop formulas.

$$\begin{aligned} G &::= A \mid G \wedge G \mid D \supset G \mid \forall_\tau x.G \\ D &::= \quad A \mid G \supset A \mid \forall x.D \end{aligned} \tag{4.4}$$

The definitions of G- and D-formulas are mutually recursive and that a negative (resp, 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

$$\begin{aligned} G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G \mid D \supset G \mid \forall x.G \\ D &::= \quad A \mid G \supset D \mid D \wedge D \mid \forall x.D \end{aligned} \tag{4.5}$$

When referring to first-order hereditary Harrop formulas and goals we shall assume this definition of formulas. We use $\mathcal{D}_2$ to denote the set of all such D-formulas and $\mathcal{G}_2$ for the set of all G-formulas. A more compact presentation can be given as

$$\begin{aligned} G &::= A \mid D \supset G \mid G \wedge G \mid \forall x.G \\ D &::= A \mid G \supset D \mid D \wedge D \mid \forall x.D \end{aligned} \tag{4.6}$$

In this presentation, D and G formulas are the same set of formula and there is no need for a definition that allows for mutual recursion. Thus, hereditary Harrop formulas are simply the logic of conjunction, implication, and universal quantification. The propositions that we now present concerning proof search also tolerate the right-introduction rules for disjunction and existential quantification (hence, presentation (4.5) is taken as the larger and official presentation of first-order hereditary Harrop formulas).

The triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_C \rangle$ is not an abstract logic programming language. For example, the formulas numbered 4, 5, 6, and 7 in Exercise 8 are hereditary Harrop goals that have classical proofs but no uniform proof.

Exercise 29. Show that Peirce's formula $((p \supset q) \supset p) \supset p$ (Exercise 8(6)) is the smallest classical theorem (counting occurrences of logical connectives) that is composed only of implications and atomic formulas and which has no uniform proof.

Let *fohh* denote the triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ or $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_M \rangle$. The following proposition shows that *fohh* is an abstract logic programming language.

Proposition 4.3 *Let Σ be a signature, let $\mathcal{P}$ be a set of Σ-formulas in $\mathcal{D}_2$, and let G be a Σ-formula in $\mathcal{G}_2$. If $\Sigma: \mathcal{P} \vdash G$ has an $\mathbf{I}$-proof then it has a uniform proof.*

The proof of this proposition is essentially the same as the proof of Proposition 4.2. Consider following class of first-order formulas given by

$$D := A \mid B \supset D \mid \forall x\ D \mid D_1 \wedge D_2.$$

Here A ranges over atomic formulas and B over arbitrary first-order formulas. These D-formulas are known as *Harrop formulas*. Clearly hereditary Harrop formulas are Harrop formulas.

$$\frac{\Sigma\colon \mathcal{P} \overset{D}{\longrightarrow} A}{\Sigma\colon \mathcal{P} \vdash A}\ \textit{decide} \qquad \frac{}{\Sigma\colon \mathcal{P} \overset{A}{\longrightarrow} A}\ \textit{init}$$

$$\frac{\Sigma\colon \mathcal{P} \overset{D_1}{\longrightarrow} A}{\Sigma\colon \mathcal{P} \overset{D_1 \wedge D_2}{\longrightarrow} A}\ \wedge\mathrm{L} \qquad \frac{\Sigma\colon \mathcal{P} \overset{D_2}{\longrightarrow} A}{\Sigma\colon \mathcal{P} \overset{D_1 \wedge D_2}{\longrightarrow} A}\ \wedge\mathrm{L}$$

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

Fig. 4.1. 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.

Exercise 30. Consider the sequent $\Sigma\colon \mathcal{P} \vdash B$ where $\mathcal{P}$ is a set of Harrop formulas and B is an arbitrary formula. Show that Harrop formulas are "uniform at the root"; that is, if B is non-atomic, then this sequent is intuitionistically provable if and only if it has a **I**-proof that ends in a right-introduction rule. Are uniform proofs complete for such sequents?

4.4 Backchaining

The restriction to uniform proofs provides some structure on how to do right rules: in the bottom-up search for proofs, right rules are attempted whenever the antecedent is non-atomic and left-rules are attempted only when the succedent is atomic. We now present restrictions on the application of left-introduction rules which do not result in the loss of completeness.

Consider searching for a proof by applying the following instance of the $\supset$L inference rule

$$\frac{\Sigma\colon \mathcal{P} \vdash G \qquad \Sigma\colon D, \mathcal{P} \vdash A}{\Sigma\colon G \supset D, \mathcal{P} \vdash A}\ \supset\mathrm{L},$$

where A is an atomic formula. Applying this rule reduces an attempt to prove the atomic formula A from program $\mathcal{P}$ to attempting to prove two things, one of which is still an attempt to prove A but this time from the (possibly) larger program $\mathcal{P} \cup \{D\}$. It would seem natural to expect this inference rule was used because this new instance of D is "directly" useful in helping to solve A. For example, D could itself be A or some sequence of additional left-rules applied to D might reduce it to an occurrence of A.

We can formalize a proof system where left introduction rules are used in such a fashion via the inference rules present in Figure 4.1. To do so, we introduce the new sequent arrow $\Sigma\colon \mathcal{P} \overset{D}{\longrightarrow} A$: the plan is that this sequent should be provable if and only if the sequent $\Sigma\colon \mathcal{P}, D \vdash A$ is provable. The formula over the sequent arrow is the only one on which left-introduction rules may be applied. The *decide* rule is used to turn the attempt to prove an atomic formula via the standard two-sided sequent into an attempt to prove this new three-place sequent.

The sequent $\Sigma\colon \mathcal{P} \vdash G$ or the sequent $\Sigma\colon \mathcal{P} \overset{D}{\longrightarrow} A$ has an **O**-*proof* if it has a proof using the right rules in Figure 3.1 and the rules in Figure 4.1. The notion $\Sigma\colon \mathcal{P} \vdash_O G$ denotes the proposition that the sequent $\Sigma\colon \mathcal{P} \vdash G$ has an **O**-proof. We shall view this proof system as capturing a high-level description of the *operational semantics* of logic programming in intuitionistic logic. Proof search for an **O**-proof has three phases. The first phase is the *goal-reduction* phase where right rules are used to find a proof of a non-atomic formula. The second phase is the decide rule in which some program clause D from the logic program is selected: alternatives to this choice of

selection may well need to be investigated. The third phase is called *backchaining* and is a focused application of left-rules and *init* in which alternatives to the choice of conjunction in the $\wedge$L rule and the choice of term in the $\forall$L rule may need to be considered.

Proposition 4.4 *Let $\mathcal{P}$ be an fohh logic program and G an fohh goal. Then $\Sigma\colon \mathcal{P} \vdash_O G$ if and only if $\Sigma\colon \mathcal{P} \vdash_I G$.*

Proof. This proof is done by permutation of inference rules. More details (meaning, the full inductive argument) should be added here. For now, see, for example [Mil89, Lemma 11], for a similar proof.

The following shows that the polarity of formulas and subformulas are maintained within cut-free **I**-proofs.

Proposition 4.5 *Let $\mathcal{P}$ be an fohh logic program and G an fohh goal and let Ξ be a cut-free* **I***-proof of $\Sigma\colon \mathcal{P} \vdash G$. If $\Sigma'\colon \Gamma \vdash B$ is a sequent in Ξ then Γ is a fohh logic program and B is an fohh goal formula.*

Let Δ be a finite set of formulas. The set of pairs $|\Delta|_\Sigma$ is defined to be the smallest set such that

- if $D \in \Delta$ then $\langle \emptyset, D\rangle \in |\Delta|_\Sigma$,
- if $\langle \Gamma, D_1 \wedge D_2\rangle \in |\Delta|_\Sigma$ then $\langle \Gamma, D_1\rangle \in |\Delta|_\Sigma$ and $\langle \Gamma, D_2\rangle \in |\Delta|_\Sigma$,
- if $\langle \Gamma, G \supset D\rangle \in |\Delta|_\Sigma$ then $\langle \Gamma \cup \{G\}, D\rangle \in |\Delta|_\Sigma$, and
- if $\langle \Gamma, \forall_\tau x\, D\rangle \in |\Delta|_\Sigma$ and t is a Σ-term of type τ then $\langle \Gamma, D[t/x]\rangle \in |\Delta|_\Sigma$.

Assuming that Δ is a set of Σ-formulas that are also *fohh* program clauses, then whenever $\langle \Gamma, D\rangle \in |\Delta|_\Sigma$ then D is a *fohh* program clause and Γ is a finite set of *fohh* goals.

By using this definition of $|\Delta|_\Sigma$, it is possible to describe backchaining as a single inference rule instead of left-introduction rules. In particular, consider the proof system $\mathcal{O}'$ that contains the right-introduction rules in Figure 3.1 and the following inference rule

$$\frac{\{\Sigma\colon \Delta \vdash G \;\mid\; G \in \Gamma\}}{\Sigma\colon \Delta \vdash A}\; BC$$

provided A is atomic and $\langle \Gamma, A\rangle \in |\Delta|_\Sigma$. If Δ is empty, then this rule has no premises.

The completeness of $\mathbf{O}'$-proofs for intuitionistic provability in the context of *fohh* is a simple consequence of the completeness of **O**-proofs (Proposition 4.4).

Proposition 4.6 *Let $\mathcal{P}$ be an fohh logic program and G an fohh goal. Then $\Sigma\colon \mathcal{P} \vdash G$ has an* $\mathbf{O}'$*-proof if and only if $\Sigma\colon \mathcal{P} \vdash_I G$.*

Exercise 31. Given the sequence $a_0, a_1, \ldots$ of atomic (propositional) formulas, define the following sequence of propositional Horn clauses

$$D_n = a_0 \supset \cdots \supset a_{n-1} \supset a_n. \quad (n \geq 0)$$

For example, D_0 is a_0, D_1 is $a_0 \supset a_1$, and D_2 is $a_0 \supset a_1 \supset a_2$. Let $n \geq 0$ be a specific number. In general, there are a great many uniform proofs of the sequent $D_0, \ldots, D_n \vdash a_n$. Among these, consider those in which the left premise of the $\supset$L rule is trivial (an initial rule). Those proofs correspond to forwardchaining proof. How do these differ in size to proofs based only on backchaining (that is, proofs in $\mathcal{O}'$)?

4.5 Dynamics of proof search for *fohc*

If $\mathcal{P}$ is an *fohc* program and G is an *fohc* goal, then there are no occurrences of $\supset$R or of $\forall$R in an **O**-proof of $\Sigma : \mathcal{P} \vdash G$. Thus, every sequent occurring in such an **O**-proof has Σ as its signature and $\mathcal{P}$ as its left-hand side. Since signatures and programs (the left-hand of sequents) remain constant during the search for proofs in *fohc*, the logic program is global. During computation, if a program clause is every needed (via the decide rule), it must be present at the beginning along with all other clauses that might be needed during the computation (proof search). Thus, the logic of *fohc* does not directly support hierarchical programming in which certain programs are designed to be local to others or in which code is assembled in modules and certain modules are "visible" or not to other modules.

The only changeable part of a sequent during proof search is the right-hand side. Since goal reduction in *fohc* is invertible (when using definitions (4.1) or (4.3)), the computational significance of the goal is given by the atoms to which it decomposes. Thus, as computation progresses, the only essential change in proof search is with atoms appearing on the right of the sequent arrow. Given that we allow first-order term and these can encode rich structures (such as natural numbers, lists, trees, Turing machine tapes, etc), it is easy to see that proof search in *fohc* has sufficient dynamics to encode general computation. Unfortunately, *all* of that dynamics takes place within *non-logical* contents, namely, within atomic formulas. As a result, logical techniques for analyzing computation via proof search have little direct impact on what can be said directly about non-logical contexts. Thus, reasoning about properties of Horn clause programs will benefit little from logical and proof theoretic analysis.

During a computation, all data structures that are built and represented using first-order terms are built from the non-logical, fixed signature, and any items that appear in signature declaration for a given sequent. In the first-order Horn clause case, neither of these signatures change and as a result, all data structures that need to be built during proof search must be available and equally "visible". Thus, *fohc* does not directly support a hierarchical notion of data structures such as is provided in many programming languages via abstract data types.

Thus computation using *fohc* is flat and supports no direct support for program-level abstractions: all the program clauses and every data type constructor must be present in the initial, endsequent in order to be used during computation. No abstractions or hiding mechanisms are available.

4.6 Examples of *fohc* logic programs

```
kind nat               type.
type z                 nat.
type s                 nat -> nat.
type sum               nat -> nat -> nat -> o.
type leq, greater      nat -> nat -> o.

sum z N N.
sum (s N) M (s P) :- sum N M P.
leq z N.
leq (s N) (s M)   :- leq N M.
greater N M       :- leq (s M) N.
```

Fig. 4.2. *fohc* programs specifying relations over natural numbers.

Figure 4.2 presents some examples of Horn clauses, along with two kinds of declarations. The syntax here is quite natural and follows the λProlog conventions. To declare members of the set of sorts S, the `kind` declaration is used: the expression

```
kind tok    type.
```

declares that `tok` is a token that is to be used as a primitive type. The expressions

```
type tok    <type expression>.
```

declares that the non-logical signature should contain the declaration of `tok` for the associated type expression. Logic program clauses are the remaining entries. In such expressions, the infix symbol `:-` denotes the reverse of $\supset$, a semicolon denotes a disjunctions, a comma (which binds tighter than `:-` and the semicolon) denotes a conjunction of goal formulas while `&` denotes conjunction for Horn clauses (in this setting, both symbols denote the same logical connective $\wedge$). Tokens with initial capital letters are universally quantified with scope around an individual clause (which is terminated by a period).

In Figure 4.2, the symbol `nat` is declared to be a primitive type and `z` and `s` are used to construct natural numbers via zero and successor. The symbol `sum` is declared to be relation of three natural numbers while the two symbols symbols `leq` and `greater` are declared to be binary relations on natural numbers. The following lines describe the meaning for these three predicates. For example, if the `sum` predicate holds for the triple M, N, and P then $N + M = P$: this relation is described recursively using the facts that $0 + N = N$ and if $N + M = P$ then $(N + 1) + M = (P + 1)$. Similarly, relations describing $N \leq M$ and $N > M$ are also specified.

Similarly, Figure 4.3 introduces a primitive type for lists (of natural numbers) and two constructors for lists, namely, the empty list constructor `nil` and the non-empty list constructor, the infix symbol `::`. The binary predicate `sumup` relates a list of natural numbers with the sum of those numbers. The binary predicate `max` relates a list of numbers with the largest number in that list. The predicate `maxx` is an auxiliary predicate used to help compute the `max` relation.

```
kind list              type.
type nil               list.
type ::                nat -> list -> list.
infixr ::              5.
type sumup, max        list -> nat -> o.
type maxx              list -> nat -> nat -> o.

sumup nil z.
sumup (N::L) S  :- sumup L T, sum N T S.
max L M         :- maxx L z M.
maxx nil A A.
maxx (X::L) A M :- leq X A,     maxx L A M.
maxx (X::L) A M :- greater X A, maxx L X M.
```

Fig. 4.3. Specifications of some relation between natural numbers and lists.

Exercise 32. Informally describe the predicates specified by Horn clauses in Figure 4.5.

Exercise 33. Take a standard definition of Turing machine and show how to define an interpreter for a Turing machine in *fohc*. The specification should be able to encode the fact that a given machine accepts a given word if and only if some atomic formula is provable.

```
kind node              type.
type a, b, c, d, e, f  node.
type adj, path         node -> node -> o.

adj a b  &  adj b c  &  adj c d  &  adj a c  &  adj e f.
path X X.
path X Z :- adj X Y, path Y Z.
```

Fig. 4.4. Encoding the adjacency and path relations for a directed graph.

```
type memb              nat  -> list -> o.
type append            list -> list -> list -> o.

memb X (X::L).
memb X (Y::L) :- memb X L.
append nil L L.
append (X::L) K (X::M) :- append L K M.

type sort              list -> list -> o.
type split             nat -> list -> list -> list -> o.

split X nil nil nil.
split X (A::L) (A::S) B :- leq A X,      split X L S B.
split X (A::L) S (A::B) :- greater A X,  split X L S B.
sort nil nil.
sort (X::L) S :- split X L Small Big, sort Small SmallS,
                 sort Big BigS, append SmallS (X::BigS) S.
```

Fig. 4.5. More examples of Horn clause programs.

4.7 Dynamics of proof search for *fohh*

Proof search using *fohh* programs and goals is slightly more dynamic. In particular, both logic programs and signatures can grow. In this setting, every sequent in an **O**-proof of the sequent $\Sigma\colon \mathcal{P} \vdash G$ is either of the form

$$\Sigma, \Sigma'\colon \mathcal{P}, \mathcal{P}' \vdash G' \qquad \text{or} \qquad \Sigma, \Sigma'\colon \mathcal{P}, \mathcal{P}' \stackrel{D}{\vdash\!\!\!-} A.$$

Thus, the signature can grow by the addition of Σ' and the logic program can grown by the addition of $\mathcal{P}'$ (a *fohh* program over $\Sigma \cup \Sigma'$). More generally, it follows from Exercise 22 that if the clausal order of $\mathcal{P}$ is $n \geq 0$ and the clausal order of G is at most $n-1$, then the clausal order of $\mathcal{P}'$ is at most $n-2$. Similarly, it is easy to see that Σ' declares items only of primitive types (excluding o).

Since the terms used to instantiate quantifiers in the concluding sequent of the $\exists$R and $\forall$L inference rules range over the signature of that sequent, more terms are available for instantiation as proof search progresses. These additional terms include the eigenvariables of the proof that are introduced by $\forall$R inference rules. Notice that once an eigenvariable is introduced, it is not instantiated by the proof search process. As a result, eigenvariable do not actually vary and, hence, act as locally scoped constants.

Modular programming: Scope extrusion via multiple conclusions.

Exercise 34. If we allow the addition of new non-logical connectives to a program, then all *fohh* programs can be reduced to programs of order 2 or less. [Hint: the inner implication of a formula of order, say 3, can be "defined" equivalent to a new atomic formula using two way implications.]

Exercise 35. Define the *core* of an abstract logic programming language $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ to be the intersection $\mathcal{D} \cap \mathcal{G}$. What is the core of *fohc*? Of *fohh*?

4.8 Examples of *fohh* logic programs

One might describe a jar as *sterile* if every germ in it is dead. Consider proving that if a given jar `j` is heated then that jar is sterile (given the fact that heating a jar kills all germs in that jar). A specification of this using *fohh* is given in Figure 4.6.

```
kind jar, germ          type.
type j                  jar.
type sterile, heated    jar -> o.
type dead               germ -> o.
type in                 germ -> jar -> o.

sterile Y :- pi x\ in x Y => dead x.
dead X    :- heated Y, in X Y.
heated j.
```

Fig. 4.6. Heating a jar makes it sterile.

The expression `pi x\` denotes universal quantification of the variable `x` with scope that extends as far to the right as consistent with parentheses or the end of the expression. The first of the clauses above could be written as

$$\forall y(\forall x(in\ x\ y \supset dead\ x) \supset sterile\ y)$$

Notice that no constructors for type `germ` are provided in Figure 4.6 and no explicit assumptions about the binary predicate `in` is given. Their role in this specification is hypothetical.

Exercise 36. Construct the **O**-proof of goal formula $(sterile\ j)$ from the logic program in Figure 4.6.

Notice that *fohh* allows for a simple notion of modular logic programming. For example, let *classify*, *scanner*, and *misc* name (possibly large) program clause have some role within a larger programming task (for example, *scanner* might contain code to convert a list of characters into a list of tokens prior to parsing, etc). The goal formula

$$misc \supset ((classify \supset G_1) \wedge (scanner \supset G_2) \wedge G_3)$$

Attempting a proof of this goal will cause attempts of the three goals G_1, G_2, and G_3 to be attempted with respect to different programs: *misc* and *classify* are used to prove G_1; *misc* and *scanner* are used to prove G_2; and *misc* is used to prove G_3. Thus, implicational goals can be used to structure the runtime environment of a program. For example, the code present in *classify* is not available during the proof attempt of G_2.

A specification for the binary predicate that relates a list with the reverse of that list can be given in *fohc* using the following program clauses:

```
reverse L K :- rev L nil K.
rev nil   L L.
rev (X::M) N L :- rev M (X::N) L.
```

Here, `reverse` is a binary relation on lists and the auxiliary predicate `rev` is a ternary relation on lists. By moving to *fohh*, it is possible to write the following specification instead.

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

Here, the auxiliary predicate `rv` is a binary predicate on lists. With this second specification, the use of non-logical context is slightly reduced in the sense that the atomic formula `(rev M K L)` in the first specification is encoded using the logical formula `(rv [] L => rv M K)` in the second specification. Notice that the definition of reverse above has clausal order 2. It is possible to specify reverse with a clause of order 3 as follows in which not only the base case for `rv` is assumed in the body of `reverse` but also the recursive case.

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

Exercise 37. Reversing a pile of papers L can informally be describing as: start by allocating an additional empty pile and then systematically move the top member of the original pile to the top of the newly allocated pile. When the original pile is empty, the other list is the reverse. Using the last specification of `reverse` above, show where in the construction of a proof of the reverse relation the informal computation actually takes place.

4.9 Limitation to *fohc* and *fohh* logic programs

The following two meta-theorems help illustrate some limitations of coding in both *fohc* and *fohh*. The following two propositions can be compared to the "Pumping Lemmas" for regular languages which help to circumscribe the expressive power of such languages. The following is similar to the Exercise 16 and implies that weakening is a property of **I**-proofs (even if weakening on the left is not an explicit inference rule).

Proposition 4.7 *Assume that $\Sigma\colon \Gamma \vdash_I G$. If Σ' is an extension of Σ and Γ' is a set of Σ'-formulas containing Γ, then $\Sigma'\colon \Gamma' \vdash_I G$.*

This proposition is proved by a simple induction on the structure of **I**-proofs. Use this Proposition to solve the following two exercises.

Exercise 38. Assume that the set or primitive types and the signature of non-logicals constants extend those in Figure 4.2. Also assume that a and *maxa* are predicates of one argument of sort *nat*. Show that there is no *fohh* program $\mathcal{P}$ that satisfies the following specification: for every set $k \geq 1$ and $\{n_1, \ldots, n_k\}$, we have $\mathcal{A}, \mathcal{P} \vdash_I maxa\ n$ if and only if n is the maximum of the set $\{n_1, \ldots, n_k\}$ and $\mathcal{A}$ is the set of atomic formulas $\{a\ n_1, \ldots, a\ n_k\}$.

As was illustrated in Figure 4.3, the maximum of a set of numbers can be computed in *fohc* if that set of numbers is stored in a list and not in the logical context as require by this exercise.

Exercise 39. Given the encoding of directed graphs as is illustrated in Figure 4.4, show that it is not possible to specify in *fohh* a predicate that is true of two nodes if and only if there is no path between them.

Another property of provable sequents is that one can substitute eigenvariables with terms and still have a provable sequent.

```
type subSome           j -> i -> i -> i -> o.

subSome X T (c X)   T.
subSome X T (c Y)   (c Y).
subSome X T (f U)   (f W)   :- subSome X T U W.
subSome X T (g U V) (g W Y) :- subSome X T U W, subSome X T V Y.
```

Fig. 4.7. Substitution of some occurrences.

Proposition 4.8 *Let τ be a primitive type and let t be a Σ-term of type τ. If $x{:}\tau, \Sigma{:}\Gamma \vdash_I G$ then $\Sigma{:}\Gamma'[t/x] \vdash_I G[t/x]$.*

Notice that this proposition can be applied to non-logical constants of primitive types in the following sense. Consider a non-logical signature, Σ_0, that contains the declaration that $c{:}\tau$. Let Σ_0' be the result of removing $c{:}\tau$ from Σ. Then the sequent $\Sigma{:}\mathcal{P} \vdash G$ is provable when the non-logical signature is Σ_0 if and only if the sequent $c{:}\tau, \Sigma{:}\mathcal{P} \vdash G$ is provable when the non-logical signature is Σ_0', which (by the above proposition) implies that $\Sigma{:}\mathcal{P}[t/c] \vdash G[t/c]$ holds for t a $\Sigma \cup \Sigma_0'$-term of type τ.

To illustrate an application of Proposition 4.8, consider the following type declarations, where i and j are primitive types.

$$c{:}\, j \to i,\ \ f{:}\, i \to i,\ \ g{:}\, i \to i \to i$$

Terms of type i exist only in contexts where constants or variables of type j are declared. Figure 4.7 contains a specification of predicate *subSome* such that $(subSome\ x\ s\ t\ r)$ is provable if and only if r is the result of substituting *some* occurrences of x (actually, of $(c\ x)$) in t with s.

Exercise 40. Prove that it is not possible in *fohh* to write a specification of *subAll* such that $(subAll\ x\ s\ t\ r)$ is provable if and only if r is the result of substituting *all* occurrences of x in t with s. Notice that this specification would need to work in any extension of the non-logical signature (in particular, for extensions that contain constants of type j that do not occur in the specification of *subAll*).

Exercise 41. Write a *fohh* specification of *subOne* such that the atom

$$(subOne\ x\ s\ t\ r)$$

is provable if and only if r is the result of substituting *exactly one* occurrences of x in t with s. One might thing that *subAll* can be specified using repeated calls to *subOne*. Given the previous exercise, this must not be possible. Explain why.

Solutions to Selected Exercises

Solution to Exercise 12 (page 18). We provide only a high-level outline of the proof: various details need to be filled in.

For one direction, we shall show how to transform a **C**-proof with restart to a **C**-proof without restart. Since **I**-proofs are **C**-proofs, this establishes the forward implication. Restarts can be removed one-by-one via the following transformation.

$$\dfrac{\dfrac{\dfrac{\overset{\Xi}{\vdots}}{\Sigma\colon \Gamma \vdash B, \Delta}}{\Sigma\colon \Gamma \vdash C, \Delta}\ \mathit{Restart}}{\dfrac{\vdots}{\Sigma'\colon \Gamma' \vdash B, \Delta'}} \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{\overset{\Xi}{\vdots}}{\Sigma\colon \Gamma \vdash B, \Delta}}{\Sigma\colon \Gamma \vdash C, B, \Delta}\ wR}{\dfrac{\vdots}{\Sigma'\colon \Gamma' \vdash B, B, \Delta'}\ cR}}{\Sigma'\colon \Gamma' \vdash B, \Delta'}$$

That is, the restart rule can be implemented using a contraction and a weakening on the right. Of course, one must check that the formula B can be added to all possible inference rules below this occurrence of the restart rule.

For a sketch of the converse direction, consider a **C**-proof. Using Exercise 22, we can assume that both the antecedent and succedent of sequents increase monotonically when moving from the bottom up. (Note that this form of proof uses a different form of the *init* rule and we are not allowed to use the wR rule.) Now mark a formula on the right-hand side of every sequent as follows. The single formula on the right of the endsequent is marked (assuming that we start proof search with a single formula to prove). If the last inference rule of the proof is a left-introduction rule, then the marked occurrence of the formula in the conclusion is also marked in all the premises. If the last inference rule is a right-introduction rule, the we have two cases: If the introduced formula is already marked, then mark its subformulas that appear in the right-hand side of any premise (for example, if the marked formula is $A \Rightarrow B$ then mark B in the premise; if the marked formula is $A \wedge B$ then mark A in one premise and B in the other; etc). Otherwise, the right-hand formula introduced is not marked, in which case, we have a *marking break*, and we mark in the premises of the inference rules the subformulas of the right-hand formula introduced and continue. The only other rules that might be applied are: cL, in which case the marked formula on the right persists from conclusion to premise; cL, in which case, if the marked formula is the one contracted then select one of its copies to mark in the premise, otherwise, the marked formula persists in the premise; and *init*, in which case, if the marked formula on the right is not the same as the formula on the left, then this occurrence of the *init* rule is also a marking break.

To illustrate this notion of marking formulas, consider the following **C**-proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p, q^*, p \supset q, p \vee (p \supset q)}\, init^*}{\vdash p, (p \supset q)^*, p \supset q, p \vee (p \supset q)}\, \supset\mathrm{R}}{\vdash p, (p \supset q)^*, p \vee (p \supset q)}\, cR}{\vdash p^*, p \vee (p \supset q), p \vee (p \supset q)}\, \vee\mathrm{R}^*}{\vdash p^*, p \vee (p \supset q)}\, cR}{\vdash p \vee (p \supset q)^*, p \vee (p \supset q)}\, \vee\mathrm{R}}{\vdash p \vee (p \supset q)^*}\, cR$$

Here, an asterisk is used to indicate marked formulas and to indicate which inference rules correspond to marking gaps.

Now the **I**-proof with Restart is built as follows. For sequents that are the conclusion of a rule that is not a marking break, delete all non-marked formula on the right. For sequents that are the conclusion of a rule that is a marking break, then this one inference rule become two: an instance of the Restart rule must be inserted and then the version of the inference rule corresponding to the marking break is put into the proof with the non-marked right-hand formulas deleted.

For example, performing this transformation on the **C**-proof yields the following structure.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\, init}{p \vdash q}\, Restart}{\vdash p \supset q}\, \supset\mathrm{R}}{\vdash p \supset q}\, cR}{\vdash p \vee (p \supset q)}\, \vee\mathrm{R}}{\vdash p}\, Restart}{\vdash p}\, cR}{\vdash p \vee (p \supset q)}\, \vee\mathrm{R}}{\vdash p \vee (p \supset q)}\, cR$$

This sequence of rules is not yet an **I**-proof: there are three occurrences of cR that are not allowed in **I**-proofs: these can either be deleted or reclassified as Restart rules.

Solution to Exercise 15 (page 18). The list of pairs for which entailment is provable in classical logic is

$$\{\langle A, \neg\neg A\rangle, \langle \neg\neg A, A\rangle, \langle \neg A, \neg\neg\neg A\rangle, \langle \neg\neg\neg A, \neg A\rangle, \}$$

The list of pairs for which entailment is provable in intuitionistic logic is the same list except that the pair $\langle \neg\neg A, A\rangle$ is removed.

Solution to Exercise 40 (page 35). Assume that there is a *fohh*-logic specifications S over the signature Σ_S. Also assume that this signature contains the constants $a : i$ and $f : i \to i \to i$. Also, assume that the constants $d : i$ and $e : i$ are not declared in Σ_S. By the specification of *subAll*, it is the case that

$$d : i, e : i, \Sigma_S \vdash_I subAll\ d\ a\ (f\ d\ e)\ (f\ a\ e).$$

By Proposition 4.8 and using the substitution of e for d, we know that

$$e : i, \Sigma_S \vdash_I subAll\ e\ a\ (f\ e\ e)\ (f\ a\ e).$$

But this contradicts the specification for *subAll*.

References

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

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

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

[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, Amsterdam, 1969. Translation of article that appeared in 1935.

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

[Gug07] Alessio Guglielmi. A system of interaction and structure. *ACM Trans. on Computational Logic*, 8(1):1–64, January 2007.

[Kle52] Stephen Cole Kleene. Permutabilities of inferences in Gentzen's calculi LK and LJ. *Memoirs of the American Mathematical Society*, 10:1–26, 1952.

[Mil89] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, January 1989.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.

[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.

[MP02] Dale Miller and Elaine Pimentel. Using linear logic to reason about sequent systems. In Uwe Egly and Christian G. Fermüller, editors, *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *LNCS*, pages 2–23. Springer, 2002.

[MP04] Dale Miller and Elaine Pimentel. Linear logic as a framework for specifying sequent calculus. In Jan van Eijck, Vincent van Oostrom, and Albert Visser, editors, *Logic Colloquium '99: Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic*, Lecture Notes in Logic, pages 111–135. A K Peters Ltd, 2004.

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

[Pri60] A. N. Prior. The runabout inference-ticket. *Analysis*, 21(2):38–39, December 1960.

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