## Contents

1 Introduction 5  
1.1 Interplay between logic and computation 5  
1.2 First-order logic formulas 6  
1.2.1 The syntax of logical formulas 6  
1.2.2 First-order and Higher-order logics 7  
1.2.3 Terms, atoms, formulas 7  
1.2.4 Quantifiers and variable bindings 8  
1.2.5 Formulas: with and without interpretation 8  
1.3 Different logics for different purposes 9  
1.3.1 Expressiveness of logic: Necessary and Contingent Truths 9  
1.3.2 Expressiveness of logic: Sensitivity to Resources 10  
1.4 Two ways to characterize theorems 10  
1.4.1 Provability and Truth: Proof 10  
1.4.2 Provability and Truth: Truth 11  
1.4.3 Soundness and Completeness: ⊢ ⇔ |= 11  
1.4.4 Incompleteness 11  
1.5 Applications of logic to computer science 12  
1.6 Exercises 13  

2 Proof systems for first-order logic 17  
2.1 Hilbert-style proofs 17  
2.2 Natural deduction proofs 18  
2.3 Sequent calculus 22  
2.4 Classical, intuitionistic, and minimal logics 24  
2.5 Mapping sequent calculus proofs to text 24  
2.6 Permutations of inference rules 25  
2.7 Cut-elimination and its consequences 26  
2.8 A one-sided sequent system for classical logic 27  
2.9 Additional readings 27  
2.10 Exercises 28
3 Proof and Truth 31
3.1 Herbrand Models 31
3.2 Soundness of sequent calculus proofs 32
3.3 Completeness of sequent calculus proofs 32
3.3.1 Derivation trees 33
3.3.2 Systematic derivation trees 34
3.4 Compactness 37
3.5 Additional readings 37
3.6 Exercises 38

4 Cut-elimination 39
4.1 Permutations of inference rules 39
4.2 Cut-elimination 39
4.3 The Mid-Sequent Theorem 42
4.4 Herbrand’s Theorem 43
4.5 Interpolation 43
4.6 Implicit and explicit definability 44
4.7 Exercises 45
Chapter 1

Introduction

1.1 Interplay between logic and computation

Logic and computation have influenced each other in many ways over the past few decades. Our interest here in logic programming relates to a small corner of that mutual influence. In particular, logics are generally used to specify computational systems in one of two broad 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 directly uses pieces of logical 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). This view of computation can be used to provide a theoretical basis to the functional programming paradigm. In the proof normalization approach, one focuses on showing that a given program (proof) yields, for example, at most one normalized value; that is, that computation is deterministic. If types are uses, then generally denote an “abstract domain” of values, such as the integers or function spaces. The fact that an expression has a does not mean that the term inhabits that type but rather that if the reduction of that expression terminates, the normal form expression is in that domain.

The proof search approach views the state of a computation as a sequent
(a structured collection of formulas encoding a current set of assumptions and the formula to be established). The process of computing in then the search for a proof of a sequent: the changes that take place in sequents capture the dynamics of computation. This view of computation can be used to provide the theoretical basis for the logic programming paradigm. In the logic programming approach, there might well be many different proofs of a statement, so computation is fundamentally non-deterministic. To deal with such non-determinism, backtracking search is generally implemented. Since terms do not reduce, they generally denote themselves only. As a result, unification becomes a key programming technique in order to discover when two terms (possibly open) can denote the same object.

Proof search is, however, a rather general activity. Mathematicians can be said to be searching for proofs when they are trying to determine the validity of some proposition. Of course, their activity has no resemblance to those low-level steps that we intend to propel a logic program’s computation. In particular, a key inference step employed by in mathematical reasoning is the discovery and use of lemmas. That is, the attempt to prove one proposition, say, \( B \), results in first finding some different formula \( C \) that is possible to prove first and whose truth, then allows the proof of \( B \). Of course, this process may need to be repeated (some new formula \( D \) is discovered to help prove \( C \), etc) and the result could be a great deal of new lemmas whose proofs inter-relate and support finally the proof of \( B \). In the sequent calculus presentation of logical deduction that we occasionally use in this book, the use of such lemmas is a use of the cut inference rule. If one attempted to model genuine mathematical proofs in a formal system, the cut rule (sometimes called modus ponens) is a frequent and critical rule. Gentzen’s famous cut-elimination theorem (for classical and intuitionistic logic) says that if a formula can be proved with cut, it has a proof without cut. This can be proved by showing that proofs of lemmas can be in-lined or redone each time the lemma is needed. Of course, the resulting cut-free proof is generally very, very large.

A difference between the proof normalization and proof search approaches to computation can be seen by reflecting on the cut rule. In the proof normalization process, it is exactly the in-lining of proofs of lemmas that can be seen as the dynamics of computing. In the proof search approach, proof search will always be restricted to mean cut-free proof search (that is, the search for proofs not containing lemmas). In this setting, thus the cut is not used during computation but can be used to reason about computations.

1.2 First-order logic formulas

1.2.1 The syntax of logical formulas

- Logical constants
  - truth values: \( \top \) (true), \( \bot \) (false)
  - connectives: \( \land \) (and, conjunction), \( \lor \) (or, disjunction), \( \Rightarrow \) (implication),
• quantifiers: $\forall$ (universal quantifier, for all), $\exists$ (existential quantifier, there exists)

- Non-logical constants
  - individual constants: $a, b, c, \ldots$
  - function constants: $f, g, h, \ldots$ (of fixed arity)
  - predicate constants: $p, q, r, \ldots$ (of fixed arity)

Warning: The literature contains many ways to write logical connectives and to present the syntax of formulas. Generally, these differences are shallow. Do not be confused.

Example 1 Propositional formulas do not have quantifiers nor function and individual constants.

$$(\neg r \supset p) \land (r \supset (q \lor r)) \land (\text{raining} \land \text{outside}) \supset \text{wet}$$

Predicate formulas (also called quantificational formulas) have quantifiers as well as function and individual constants.

$$\forall y \forall x (p(x) \land q(x, y) \supset \exists z q(z, a)) \land P(P a \supset P b)$$
$$\forall [\text{integer}(n) \supset (\text{even}(n) \lor \text{odd}(n))]$$
$$\forall X \forall L \forall K \forall M (\text{append}(L, K, M) \supset \text{append}(X :: L, K, X :: M))$$

1.2.2 First-order and Higher-order logics

There are two kinds of predicate logics typically studied.

First-order logic: the only variables allowed are individual variables.

Higher-order logic: variables can individual, function, and predicate.

In this course, we shall restrict ourselves to first-order logic. The study of higher-order logics has gained a great deal in recent years. This popularity arises from its greater expressiveness and since many problems are naturally thought of in terms of higher-order logic. We limit ourselves here not only to make our study of meta-theory for logic easier but also because first-order logic is a natural logic to consider and has a number of important computational applications.

1.2.3 Terms, atoms, formulas

In first-order logic, we have:

Terms: these are generated from individual constants, individual variables, and function constants. They can be represented as trees where the non-terminals are function constants and the terminal nodes are individual constants or variables.

Atomic formulas are of the form $P(t_1, \ldots, t_n)$, where $P$ is a predicate of arity $n$ ($n \geq 0$) and $t_1, \ldots, t_n$ are terms.

Formulas are built then from atomic formulas and logical connectives and quantifiers.
1.2.4 Quantifiers and variable bindings

Quantifiers introduce into the syntax of formulas the familiar notions of
1. bound and free variable occurrences,
2. variable scopes, and
3. alphabetic changes in bound variables.

For example, the following formula has three quantifiers (binders) for variables. What are their scopes?
\[ \forall x \forall y [P(x, y, z) \supset \exists x [Q(x, y)]] \]

An important operation on formulas (as in functional programming languages) is the notion of substitution, which can replace a variable with a term. For example, the following two formulas are “instantiations” of this formula.
\[ \forall y [P(a, y, z) \supset \exists x [Q(x, y)]], \quad \text{where } x \mapsto a \]
\[ \forall y' [P(f(a, y), y', z) \supset \exists x [Q(x, y)]]], \quad \text{where } x \mapsto f(a, y) \]

1.2.5 Formulas: with and without interpretation

Thus:
\[ P(a) \quad \forall x [P(x) \supset Q(x)] \]
Thus: \[ Q(a) \]

This step can be taken always, no matter what the meaning for \( P, Q, \) and \( a \) is. It depends only on the meaning of \( \forall \) and \( \supset \).

Consider the following meaning for these non-logical symbols.
\[ P(y) = y \text{ is a prime number} \]
\[ Q(y) = y \text{ is a positive number} \]
\[ a = \text{is the integer 17} \]

This is an (informal) example of a model: The domain of discourse is (apparently) the integers; the predicates \( P \) and \( Q \) are explained; and the symbol \( a \) is denoted.

Of course, there are many other such interpretations.

Example 2 (from the “blocks world”) Consider the following formula:
\[ \text{red}(a) \land \text{blue}(c) \land \text{on}(a, b) \land \text{on}(b, c) \land \forall x [\text{blue}(x) \supset \neg \text{red}(x)] \]

Is the following a consequence of this? (Is there a red block on a non-red block?)
\[ \exists u \exists v [\text{red}(u) \land \neg \text{red}(v) \land \text{on}(u, v)] \]

If so, how do you prove it?

Is \( a \) on \( c \)? Can a block be both red and blue?
Example 3 (A logic program) *The following is an example of Prolog code.*

```prolog
adj(a,b).
adj(b,c).
adj(b,d).
adj(d,e).

path(X,Y) :- adj(X,Y).
path(X,Z) :- adj(X,Y), path(Y,Z).
```

*Here, :- denotes the reverse of implications and comma denotes conjunction. Tokens starting with an uppercase letter are variables and tokens starting with a lower case letters are constants. Quantifiers are not written but are implicit.*

Is there a path from a to e? Can we see computing this answer as finding a proof?

1.3 Different logics for different purposes

There is not just one logic, although, historically, *classical logic* has dominated.

- Classical logic: Captures the notion of “truth.” Used by mathematicians and philosophers. Truth-tables are part of the analysis of (propositional) classical logic.
- Intuitionistic logic (1930s): Captures “provability” and “constructive truths.” If you prove that an object should exist, you are actually required to construct that object.
- Linear logic (1987): Captures “resources” and makes the specification of computation more immediate. It also can explain intuitionistic and classical logics.

There are also a number of other logics that have been designed: modal logics, temporal logics, relevant logics, etc.

One must be careful in designing new logics. See, for example, broccoli logic.

1.3.1 Expressiveness of logic: Necessary and Contingent Truths

Consider the truth of the following two statements.

- All USA presidents are native born USA citizens.
- All USA presidents are at least 40 years old when elected.

Both statements are true of the 43 current presidents.

Only the first is *necessarily* true (if the US Constitution is not amended). There could be a president elected in the future that is between the age of 35 and 40 years old.

Such distinctions are important in data bases.

Classical logic does not address these issues directly. Intuitionistic logic and some modal logics do address these distinctions.
1.3.2 Expressiveness of logic: Sensitivity to Resources

Consider the following two sentences.

○ If I have a dollar then I can buy a coke.
  \[ \text{dollar} \supset \text{coke} \]

○ If I have a dollar then I can buy a pepsi.
  \[ \text{dollar} \supset \text{pepsi} \]

In classical and intuitionistic logics, we would be able to infer
  \[ \text{dollar} \supset (\text{coke} \land \text{pepsi}) \],
which is, in a certain sense, not a sensible conclusion. More appropriate would be
  \[ (\text{dollar} \land \text{dollar}) \supset (\text{coke} \land \text{pepsi}) \].

Clearly, \( \text{dollar} \land \text{dollar} \) and \( \text{dollar} \) should not be equivalent, although they are if we are only modeling truth.

Linear logic will provide a setting were this distinct is directly supported.

1.4 Two ways to characterize theorems

Theorems are the statements that are to be valid consequences of a logic. There are generally speaking, two rather different approaches to characterizing a formulas as a theorem. One mention argues that no matter how the domain is organized, the statement must be true in that domain: this approach is the model-theoretic notion of truth. The other approach uses proof or arguments: this approach is the proof-theoretic approach.

1.4.1 Provability and Truth: Proof

Provability always refers to a finite demonstration (proof, argument) that a statement holds. Proofs are syntactic and can be encoded in and checked by machine (in fact, such checks should be “simple” polynomial-time computations). The following rewriting can serve as a proof that \( (p \supset q) \lor p \) holds (in classical logic).

\[
(p \supset q) \lor p \equiv (\neg p \lor q) \lor p \\
(q \lor \neg p) \lor p \equiv q \lor (\neg p \lor p) \equiv q \lor \top \equiv \top
\]

This proof, of course, assumes some simple “proof” principles (what are they?).
1.4.2 Provability and Truth: Truth

Truth is generally defined by use of mathematical structures, called models, which are often infinite. Computational tractability is not generally a concern.

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>(p ⊃ q) ∨ p</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>⊤</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td>⊥</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊤</td>
</tr>
</tbody>
</table>

Checking truth-tables for propositional formulas requires exponential algorithms. In first-order logic (with quantifiers), checking truth might, in principle, require checking an infinite number of models.

1.4.3 Soundness and Completeness: ⊢ ≡ |=

A very important property of a logic is that the following two theorems hold:

- **Soundness**: If B is provable then B is true. (If ⊢ B then |= B.)
- **Completeness**: If B is true then B is provable. (If |= B then ⊢ B.)

Soundness proofs are generally easier to prove, often by structural induction on proofs.
Completeness proofs are generally hard to prove and require deep understanding of both models and proofs.

The issue of soundness and completeness is echoed whenever we build machines/software for performing tasks. Consider a combination lock, for example.

- **Soundness**: If I enter 15-6-29 then the lock opens.
- **Completeness**: If the lock opens, then I entered 15-6-29.

1.4.4 Incompleteness

Sometimes, incompleteness means you should go back and either

1. add proof principles so you can prove more formulas (careful not to be inconsistent and prove too many formulas!), or
2. increase the number of structures that you consider models (of course, you don’t want to lose the structures that you are interested in).

There is one famous incompleteness result, however. Consider the “standard model” of arithmetic. (Recall Kronecker’s dictum: “Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.”)

Gödel (1931) proved that there is no reasonable way to add proof principles so as to have completeness for this model.
1.5 Applications of logic to computer science

Logic assertions in code for debugging and correctness proofs (Hoare logic).

Theorem provers for supporting artificial intelligence, expert systems, deductive databases.

Automation and assistance in doing formal mathematics. Computer algebra.

Model checking.

Logic programming languages: e.g., Prolog, λProlog.

Functional programming.

Proof carrying code.
1.6 Exercises

For the purposes of this exercise, we fix the following aspects of propositional logic. \textit{Prop} is a fixed set of propositional symbols (possibly infinite). The logical connectives are: \(\top, \bot, \neg, \land, \lor, \supset\). The size of a formula is the number of occurrences of either propositional symbols or \(\top\) or \(\bot\) in a formula.

Exercise 1: Strings denoting propositional formulas

Consider the following context-free grammar. Let the set of non-terminal symbols, \(\Sigma\), be \(\text{Prop} \cup \{ (,), \top, \bot, \neg, \land, \lor, \supset\}\). The only non-terminal is the start symbol \(S\) and the inference rules are the following:

\[
S \rightarrow p, \text{ for all } p \in \text{Prop}
\]

\[
S \rightarrow \top | \bot
\]

\[
S \rightarrow (\neg S) | (S \land S) | (S \lor S) | (S \supset S)
\]

Let \(\text{Form}\) be the set of formulas generated by this grammar. Generally, it is more customary to pick a language for propositional formulas that does not have so many parentheses. That number is generally reduced by using the following conventions: the two binary operators \(\land\) and \(\lor\) associate to the left, the binary operator \(\supset\) associate to the right, and the priority of operators is such that \(\neg\) binds tighter than \(\land\) which binds tighter than \(\lor\) which binds tighter than \(\supset\). For this exercise, assume the simpler description using lots of parentheses.

Define the function \(K : \Sigma \rightarrow \mathbb{Z}\) as follows:

- \(K(\ ) = -2, K(\ ) = 2, K(\neg) = 0, K(\land) = K(\lor) = K(\supset) = -1\)
- \(K(p) = 1, \text{ for all } p \in \text{Prop} \cup \{\bot, \top\}\).

Similarly define the function \(K^* : \Sigma^* \rightarrow \mathbb{Z}\) as \(K^*(u_1 \ldots u_n) = K(u_1) + \cdots + K(u_n)\) (where \(\{u_1, \ldots, u_n\} \subseteq \Sigma\)).

1. Prove that for any \(F \in \text{Form}\), \(K^*(F) = 1\).

2. Show that no proper prefix of a string denoting a propositional formula is itself a string denoting a propositional formula.

Exercise 2: Validity between sets of formulas

We extend the use of the binary relation \(\models\) to relate sets of formulas in the following way: Let \(\Delta\) and \(\Gamma\) be sets of propositional formulas. We say that \(\Delta \models \Gamma\) if for every model \(M\) such that for all \(D \in \Delta, M \models D\) then for some \(G \in \Gamma, M \models G\). Prove that if \(\Delta_1 \models \Gamma_1 \cup \{B\}\) and \(\Delta_2 \cup \{B\} \models \Gamma_2\) then \(\Delta_1, \Delta_2 \models \Gamma_1, \Gamma_2\). Conclude also that if \(B\) is a tautology and \(\Delta \cup \{B\} \models \Gamma\) then \(\Delta \models \Gamma\)
Exercise 3: Propositional Horn clauses

A propositional Horn clause is a formula of the form

$$(p_1 \land \cdots \land p_n) \supset p_0$$

where $n \geq 0$ and $\{p_0, \ldots, p_n\} \subseteq \text{PROP}$. If $n = 0$, then we chose to write this clause simply as $p_0$. Horn clauses have the form of rules ($n \geq 1$) or facts ($n = 0$). Let $\mathcal{H}$ be a finite set of propositional Horn clauses. Define the operator $T_{\mathcal{H}} : 2^{\text{PROP}} \rightarrow 2^{\text{PROP}}$ such that

$$T_{\mathcal{H}}(I) = \{p_0 \mid \text{there is a } (p_1 \land \cdots \land p_n) \{\supset} p_0 \text{ in } \mathcal{H} \text{ and } \{p_1, \ldots, p_n\} \subseteq I\}.$$

1. Show that $I$ is a model of $\mathcal{H}$ if and only if $T_{\mathcal{H}}(I) \subseteq I$.

2. Show that $T_{\mathcal{H}}$ is monotone: that is, if $I \subseteq J$ then $T_{\mathcal{H}}(I) \subseteq T_{\mathcal{H}}(J)$.

3. Show that the largest model of $\mathcal{H}$ is $\text{PROP}$ (that is, the model where all propositional symbols are true) and that the smallest model for $\mathcal{H}$ is

$$\bigcup_{N \geq 0} T_{\mathcal{H}}^N(\emptyset) = \emptyset \cup T_{\mathcal{H}}(\emptyset) \cup T_{\mathcal{H}}(T_{\mathcal{H}}(\emptyset)) \cup \ldots$$

In fact, the value above is the least fixed point for $T_{\mathcal{H}}$.

Exercise 4: Negation normal form

A formula is negation normal form (nnf) if it contains no occurrences of $\supset$ and if negations have only atomic scope. Any formula can be rewritten to a logically equivalent formula that is in negation normal form by rewriting it by a series of simple equivalences.

1. List all equivalences that are needed to make this transformation. For example, one such equivalence is probably $\neg(A \land B) \equiv (\neg A \lor \neg B)$.

2. How does the size of a formula change when it is rewritten in this fashion?

Exercise 5: Complete set of connectives

A set of connectives $\mathcal{S}$ is complete if, for all $n \geq 0$, they can be used to describe all functions from $\{\top, \bot\}^n \rightarrow \{\top, \bot\}$ by interpreting propositional formulas containing $n$-propositional letters and the connectives in $\mathcal{S}$. The set $\{\top, \bot, \neg, \land, \lor, \supset\}$ is a complete set of connectives.

1. Let $\uparrow$ be the NAND logical operator defined as $A \uparrow B \equiv (\neg(A \land B))$. Show that the set $\{\uparrow\}$ is a complete set of propositional connectives.

2. Enumerate all minimal subsets of the set $\{\top, \bot, \neg, \land, \lor, \supset\}$ that are complete sets of connectives.
Exercise 6: Conjunctive normal form and validity checking

A literal is either a propositional symbol or its negation. A clause is a disjunction of literals (for the empty disjunction, we take $\bot$). A formula is in conjunctive normal form (cnf) if it is a conjunction of clauses (for the empty conjunction, we take $\top$).

1. Prove that a clause is valid if and only if it contains a propositional symbol and its negation.

2. Describe an algorithm that determines if a given cnf formula is valid. How complex is this algorithm?

3. By using nnf and the distribution of $\lor$ over $\land$, namely,

\[(A \lor (B \land C)) \equiv ((A \lor B) \land (A \lor C)),\]

describe an algorithm for determining if an arbitrary formula is valid. Do not use truth tables here; instead, use the algorithm in the preceding problem as a subroutine.

4. Can modify your algorithm to produce a proof: in this case, a proof can be a sequence of logically equivalent formulas where the first formula is easily recognized as a tautology, the last formula is the formula to be proved, and the remaining formulas are all logically equivalent to their preceding formula by a simple rewriting.

5. How would you rewrite this exercise if you were interested in proving that a formula is satisfiable instead of valid.
Chapter 2

Proof systems for first-order logic

While this course will mostly focus on first-order classical logic, we will also discuss intuitionistic logic, at least in contrast it with classical logic.

There are a number of systems for expressing proofs. We shall first present two, namely, *Hilbert style* proofs and *natural deduction* proofs, and illustrate them briefly. We will then present the *sequent calculus* in much more detail and use this style of proof to explore the soundness and completeness of first-order classical logic.

2.1 Hilbert-style proofs

Hilbert-style proofs are the simplest one to define. These have been successfully used in a number of settings when one is interested in *provability* [And86, End72]. They are of little use if one is interested in the *structure of proofs*. One first identifies a collection of *axioms*. For this, take instances of the following formulas.

1. All tautological formulas.
2. $(\forall x.A) \supset A[t/x]$ for terms $t$ and $A$ formulas. The expression $A[t/x]$ denotes the result of substituting the term $t$ for all free occurrences of $x$ in $A$ (renaming bound variables to avoid variable capture).
3. $(\forall x(A \supset B) \supset ((\forall x.A) \supset (\forall x.B))$ for formulas $A$ and $B$.
4. $A \supset (\forall x.A)$ for formula $A$ and when $x$ is not free in $A$.
5. $(\exists x.A) \supset (\neg \forall x.\neg A)$ and $(\neg \forall x.\neg A) \supset (\exists x.A)$, for formula $A$.

Then one identifies the inference rule, which we will take as the following:

*Modus ponens*: from $A$ and $A \supset B$ infer $B$. 17
Generalization: from $A$, infer $\forall x.A$.

A proof of $B$ from the assumptions $\Delta$ (a possibly infinite set of formulas) in Hilbert-style is a sequence of formulas $B_1, \ldots, B_n$ where $n > 0$, $B$ is $B_n$, and for all $i \in \{1, \ldots, n\}$, $B_i$ is either an axiom or $B_i \in \Delta$ or $B_i$ is the consequence of an inference rule in which the premises are in the list $B_1, \ldots, B_{i-1}$. We shall write $\Delta \vdash B$ to denote that such a proof exists.

The compactness property for provability is immediate and trivial: if $\Delta \vdash B$ then there exists a finite set $\Delta' \subseteq \Delta$ such that $\Delta' \vdash B$. (Replacing the proof theoretic notion $\vdash$ with the semantic notion of $|=\$ yields a much more interesting and non-trivial result.)

The deduction theorem, namely, if $\Delta \cup \{ A \} \vdash B$ then $\Delta \vdash A \supset B$, must be proved about Hilbert-style proofs and requires a non-trivial proof. This natural form of reasoning will actually be built into the next two styles of proof that we shall see.

2.2 Natural deduction proofs

Natural deduction proofs were first introduced by Gentzen in [Gen69]. Prawitz [Pra65] was the first to study them in detail. We will not present a detailed definition of these proof objects here since we will not use these proofs in subsequent lectures. Instead, we simply point out that since you are familiar with typed lambda-terms and typed functional programs, then you are already familiar with natural deduction proofs for fragments of intuitionistic logic.

Consider the following closed expressions in the subset of ML (or CAML) that uses only application and function-abstraction.

- val I = fn x => x
  = and K = fn x => fn y => x
  = and S = fn f => fn g => fn x => (g (f x))
  = and C = fn f => fn x => fn y => (f x y)
  = and S = fn x => fn y => fn z => ((x z) (y z));

val I = fn : 'a -> 'a
val K = fn : 'a -> 'b -> 'a
val S = fn : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
val C = fn : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c

By identifying $\to$ with implication, all of these types correspond to tautologies. It is a simple matter to add pairing to the functional programs and thus to add conjunctions to the formulas corresponding to types.

- abstype ('a,'b) conj = c of 'a * 'b
  = with
  = fun pair x y = c(x,y)
  = fun left (c(x,y)) = x
fun right(c(x,y)) = y
end;

type ('a,'b) conj
val pair = fn : 'a -> 'b -> ('a,'b) conj
val left = fn : ('a,'b) conj -> 'a
val right = fn : ('a,'b) conj -> 'b

(* Prove: (a -> b) -> (a and c) -> b *)
fun p1 x y = (x (left y));
val p1 = fn : ('a -> 'b) -> ('a,'c) conj -> 'b

(* Prove: ((a -> b) and (a -> c)) -> a -> (b and c) *)
fun p2 x y = pair ((left x) y) ((right x) y);
val p2 = fn : ('a -> 'b,'a -> 'c) conj -> 'a -> ('b,'c) conj

(* Prove: (a and b) -> (b and a) *)
fun p3 x = pair (right x) (left x);
val p3 = fn : ('a,'b) conj -> ('b,'a) conj

Similarly, disjunctive types can also be easily added.

- abstype ('a,'b) or = left of 'a | right of 'b

val putl = fn : 'a -> ('a,'b) or
val putr = fn : 'a -> ('b,'a) or
val cases = fn : ('a,'b) or -> (left x) f g = f x

| cases (right x) f g = g x

end;

type ('a,'b) or
val putl = fn : 'a -> ('a,'b) or
val putr = fn : 'a -> ('b,'a) or
val cases = fn : ('a,'b) or -> (left x) f g = f x

| cases (right x) f g = g x

end;

If we consider the fragment of ML we have seen so far, we have the typing rules of Figure 2.1, written as inference rules. Another way to see these rules are as the inference rules in Figure 2.2, for a propositional logic, involving only implication, conjunction, and disjunction. These rules are examples of natural deduction inference rules.

To find a program of the type

('a -> 'b,'a -> 'c) or -> 'a -> ('b,'c) or
\[(x : a)\]

\[
\begin{array}{c}
t : a \rightarrow b \quad s : a \\
\hline
\end{array}
\]

\[
\begin{array}{c}
t : b \\
\hline
\end{array}
\]

\[
\begin{array}{c}
(t \ s) : b \\
\hline
(fn \ x \mapsto t) : a \rightarrow b
\end{array}
\]

\[
\begin{array}{c}
t : a \\
\hline
s : b \\
\hline
(t, s) : (a, b) \text{ conj}
\end{array}
\]

\[
\begin{array}{c}
t : (a, b) \text{ conj} \\
\hline
(\text{left } t) : a \\
\hline
(\text{right } t) : b \\
\hline
\end{array}
\]

\[
\begin{array}{c}
t : a \\
\hline
\end{array}
\]

\[
\begin{array}{c}
t : b \\
\hline
\end{array}
\]

\[
\begin{array}{c}
(\text{putl } t) : (a, b) \text{ or} \\
\hline
(\text{putr } t) : (a, b) \text{ or}
\end{array}
\]

\[
\begin{array}{c}
t : (a, b) \text{ or} \\
\hline
f : a \rightarrow c \\
\hline
\end{array}
\]

\[
\begin{array}{c}
g : b \rightarrow c \\
\hline
\end{array}
\]

\[
\begin{array}{c}
(\text{cases } t \ f \ g) : c
\end{array}
\]

Figure 2.1: Typing rules for a subset of ML.

(a)

\[
\begin{array}{cccccccc}
a \rightarrow b & a & b & a & b & a \text{ and } b & a \text{ and } b & \\
\hline
b & a \rightarrow b & a \text{ and } b & a & b & \\
\hline
a & b & a \text{ or } b & a \rightarrow c & b \rightarrow c & \\
\hline
\end{array}
\]

\[
\begin{array}{cccc}
a \text{ or } b & a \text{ or } b & c & \\
\hline
\end{array}
\]

Figure 2.2: Corresponding natural deduction proof rules.
consider how you might prove it as a formula. This leads naturally to the program

```ml
-fn x =>(fn y => cases x (fn z=> putl(z y)) (fn z=> putr(z y)));
```

For another example, consider finding a proof of

\((a, b) \text{ or } (a \rightarrow c) \rightarrow (b \ast d \rightarrow c) \rightarrow d \rightarrow c\)

This can be done with the following tree, which can be encoded using the following typed functional program.

```
  b   d
  ____
 b * d -> c  b * d
 ________________
       c
  ______
 (a, b) or  a -> c  b -> c
 _____________________________
       c
  ______
 d -> c
 ______________
 ((b * d) -> c) -> d -> c
 _____________________________
 (a -> c) -> ((b * d) -> c) -> d -> c
 _________________________________
(a,b) or -> (a -> c) -> ((b * d) -> c) -> d -> c
```

```ml
-fn x => fn y => fn z => fn w => (cases x y (fn m => (z (m,w))));
```

Clearly, programs correspond to proofs, and types correspond to propositional logic formulas. This correspondence is called by various names: \textit{formulas-as-type}, \textit{proofs-as-programs}, and the \textit{Curry-Howard Isomorphism}.

The logic for which the types corresponds most closely is intuitionistic propositional logic, which is weaker than classical logic. To illustrate this difference between classical and intuitionistic logic, take a proof of propositional formula \(B\) mean that there is a “pure” ML program of \(B\). Take truth (informally) to mean that the formula satisfies all truth assignments (via truth tables). The following is not hard to prove.

\textbf{Soundness:} If \(B\) is is provable then \(B\) is true.

The converse statement is called \textit{completeness} and it does not hold. That is, there are classically true formulas that are not the type of any ML program. For example, consider the following formulas/types.

\((\textsf{\texttt{('a -> 'b) -> 'a)} \rightarrow 'a}) \rightarrow 'a\)
\((\textsf{\texttt{('a -> 'b -> 'c)} \rightarrow ('a -> 'c, 'b -> 'c) or ('a -> 'c, 'b -> 'c) or})\)
2.3 Sequent calculus

The sequent calculus provides a setting to reason about logical truth by considering the form of deduction. In this approach to proof, there are no axioms, only inference rules. In fact, for most logical connectives there are two introduction rules, one describing the form of an argument in which one proves a formula with the given connective and one describing the form of an argument in which one reasons from such a formula. There are also the structural rules of contraction and weakening, as well as the two special rules called initial and cut.

Note that the rule called initial should not really be called an axiom. The sequent calculus does not work like Hilbert-style proofs. In sequent calculus proofs, proofs are more about the structure of an argument: the initial rule is better thought of as a primitive argument and not a formula which is accepted as a primitive truth.

Let \( F \) denote a formulation of first-order logic. We shall assume that set of non-logical constants are fixed and that formulas of \( F \) can contain constants only from this set. A first-order signature \( \Sigma \) is a set of first-order typed variables. A \( \Sigma \)-formula is a formula all of whose free variables are contain in the set \( \Sigma \). In the treatment of sequent calculus elsewhere, it is also common to omit mentioning signatures entirely and, instead, have notions of formulas with and without free variables. The more explicit approach we take here will have some advantages. Notice that although we allow variables to be typed (often called “multi-sorted first-order logic”), types will play be of little importance for us here: we shall usually assume that variables and constants involve only a single, primitive type, say \( i \).

To define sequent calculus proofs for \( F \) we follow Gentzen [Gen69]. A sequent of \( F \) is a triple \( \Sigma : \Gamma \rightarrow \Delta \), where \( \Sigma \) is a first-order signature over \( S \) and \( \Gamma \) and \( \Delta \) are finite (possibly empty) multisets of \( \Sigma \)-formulas. (A multiset is a set in which the multiplicity of elements is maintained.) The multiset \( \Gamma \) is this sequent’s antecedent and \( \Delta \) is its succedent. The expressions \( \Gamma, B \) and \( B, \Gamma \) denote the multiset union \( \Gamma \cup \{B\} \). The rules for introducing the logical connectives are presented in Figure 2.3, the initial and cut rules are given in Figure 2.4, and the structural rules are given in Figure 2.5. The following provisos are also attached to the four inference rules for quantifier introduction: in \( \forall R \) and \( \exists L \), the constant \( c \) is not in \( \Sigma \), and, in \( \forall L \) and \( \exists R \), \( t \) is a \( \Sigma \)-term of type \( \tau \).

The rules for the binary logical connectives that have two premises can be classified as either additive or multiplicative. A rule is additive if the context surrounding the logical formula introduced in the conclusion is the same as the context surrounding its immediate subformulas in the premise sequents. The \( \forall L \) and \( \wedge R \) rules are additive. A rule is multiplicative if the context surrounding the logical formula introduced in the conclusion is the accumulation of as the contexts surrounding its immediate subformulas in the premise sequents. The \( \supset L \) rule is multiplicative. As we show in an exercise below, in the presence of weakening and contraction, it is possible to move between these two styles.
\[ \Sigma : B, \Delta \rightarrow \Gamma \]  
\[ \Sigma : B \land C, \Delta \rightarrow \Gamma \]  
\[ \Sigma : C, \Delta \rightarrow \Gamma \]  
\[ \Sigma : B \land C, \Delta \rightarrow \Gamma \]  
\[ \Sigma : \Delta \rightarrow \Gamma, B \]  
\[ \Sigma : \Delta \rightarrow \Gamma, C \]  
\[ \Sigma : \Delta \rightarrow \Gamma, B \land C \]  
\[ \Sigma : B, \Delta \rightarrow \Gamma \land \Delta \]  
\[ \Sigma : B, \Delta \rightarrow \Gamma \]  
\[ \Sigma : \Delta \rightarrow \Gamma, B \land C \]  
\[ \Sigma : B \lor C, \Delta \rightarrow \Gamma \]  
\[ \Sigma : \Delta \rightarrow \Gamma, B \lor C \lor \Delta \]  
\[ \Sigma : B, \Delta \rightarrow \Gamma \lor \Delta \]  
\[ \Sigma : C, \Delta \rightarrow \Gamma \lor \Delta \]  
\[ \Sigma : \Delta \rightarrow \Gamma \lor \Delta \]  
\[ \Sigma : \Delta \rightarrow \Gamma \lor \Delta \lor \Delta \]  
\[ \Sigma : \Delta \rightarrow \Gamma \lor \Delta \lor \Delta \]  

Figure 2.3: Introduction rules.

\[ \Sigma : B \rightarrow B \text{ initial} \]
\[ \Sigma : \Delta_1 \rightarrow \Gamma_1, B \]
\[ \Sigma : \Delta_2 \rightarrow \Gamma_2 \]
\[ \Sigma : \Delta_1, \Delta_2 \rightarrow \Gamma_2 \]
\[ \Sigma : \Delta_1, \Delta_2 \rightarrow \Gamma_1, \Gamma_2 \]
\[ \Sigma : \Delta_1, \Delta_2 \rightarrow \Gamma_1, \Gamma_2 \]

Figure 2.4: Initial and cut rules.

\[ \Sigma : \Delta \rightarrow \Gamma \text{ weakL} \]
\[ \Sigma : \Delta, B \rightarrow \Gamma \text{ weakL} \]
\[ \Sigma : \Delta, B \rightarrow \Gamma \text{ weakR} \]
\[ \Sigma : \Delta \rightarrow \Gamma, B \]
\[ \Sigma : \Delta \rightarrow \Gamma, B \]
\[ \Sigma : \Delta \rightarrow \Gamma, B \]
\[ \Sigma : \Delta \rightarrow \Gamma, B \]

Figure 2.5: Structural rules.
of inference rules without changing the logic. If these structure rules are not present (as is the case in linear logic), then these two different ways to present inference rules result in two different logical connectives: for example, in linear logic there is an additive conjunction and a multiplicative conjunction (there is also an exponential that relates these two following the usual algebraic law: the exponential of an addition is a multiplication of exponentials).

A proof of the sequent $\Sigma : \Gamma \rightarrow \Theta$ is a finite tree constructed using these inference rules such that the root is labeled with $\Sigma : \Gamma \rightarrow \Theta$.

2.4 Classical, intuitionistic, and minimal logics

Any proof is also called a C-proof. Any C-proof in which the succedent of every sequent in it has at most one formula is also called an I-proof. Furthermore, an I-proof in which the succedent of every sequent in it has exactly one formula is also called an M-proof. Sequent proofs in classical, intuitionistic, and minimal logics are represented by, respectively, C-proofs, I-proofs, and M-proofs. Finally, let $\Sigma$ be a given first-order signature over $S$, let $\Gamma$ be a finite set of $\Sigma$-formulas, and let $B$ be a $\Sigma$-formula. We write $\Sigma; \Gamma \vdash_C B$, $\Sigma; \Gamma \vdash_I B$, and $\Sigma; \Gamma \vdash_M B$ if the sequent $\Sigma : \Gamma \rightarrow B$ has, respectively, a C-proof, an I-proof, or an M-proof. It follows immediately that $\Sigma; \Gamma \vdash_M B$ implies $\Sigma; \Gamma \vdash_I B$, and this in turn implies $\Sigma; \Gamma \vdash_C B$.

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

The notion of provability defined here is not equivalent to the more usual presentations of classical, intuitionistic, and minimal logic [Fit69, Gen69, Pra65, Tro73] in which signatures are not made explicit and substitution terms (the terms used in $\forall L$ and $\exists R$) are not constrained to be taken from such signatures. The main reason they are not equivalent is illustrated by the following example. Consider a formulation of $F$ such that there are no constants of type $i$ and consider the sequent

$$\{\} : \forall_i x p(x) \rightarrow \exists_i x p(x).$$

This sequent has no proof even though $\exists_i x p(x)$ follows from $\forall_i x p(x)$ in the traditional presentations of classical logics. The reason for this difference is that there are no $\{\}$-terms of type $i$: that is, the type $i$ is empty. Thus, there is no way to prove the above sequent. The usual assumption made when studying classical logic is that every primitive type is inhabited: that is, there is a non-logical constant of primitive type. Under this assumption, the notions of provability defined above coincide with the more traditional presentations.

2.5 Mapping sequent calculus proofs to text

In class, we describe how an I-proof can be mapped recursively to a collection of English language sentences.
2.6 Permutations of inference rules

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

\[
\begin{align*}
\Sigma : \Delta, p, r \rightarrow s, \Gamma & \quad \Sigma : \Delta, q, r \rightarrow s, \Gamma & \quad \triangleright L \\
\Sigma : \Delta, p \lor q, r \rightarrow s, \Gamma & \quad \triangleright R
\end{align*}
\]

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.

\[
\begin{align*}
\Sigma : \Delta, p, r \rightarrow s, \Gamma & \quad \Sigma : \Delta, q, r \rightarrow s, \Gamma & \quad \triangleright R \\
\Sigma : \Delta, p \lor q, r \rightarrow s, \Gamma & \quad \triangleright L
\end{align*}
\]

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

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

\[
\begin{align*}
\Sigma : \Delta, p, r \rightarrow s, \Gamma & \quad \Sigma : \Delta, q, r \rightarrow s, \Gamma & \quad \triangleright R \\
\Sigma : \Delta, p \lor q, r \rightarrow s, \Gamma & \quad \triangleright L
\end{align*}
\]

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

\[
\begin{align*}
\Sigma : \Delta, p, r \rightarrow s, \Gamma & \quad \Sigma : \Delta, q, r \rightarrow s, \Gamma & \quad \triangleright R \\
\Sigma : \Delta, p \lor q, r \rightarrow s, \Gamma & \quad \triangleright L
\end{align*}
\]

Notice that if the first collection of inference rules was from an I-proof, then \(\Gamma_1\) and \(\Gamma_2\) must be empty. However, the result of permuting these inference rule would necessarily be a C-proof since we are required to have a sequent with two copies of \(r \supset s\) on the right. In general, an \(\triangleright R\) below an \(\triangleright L\) in an I-proof cannot always be permuted.

One important use of permutations of inference rules is to show when an inference rule is invertible. For example, given the \(\triangleright R\) inference rule, we know that if \(\Sigma : \Delta, p, r \rightarrow s, \Gamma\) has a proof then \(\Sigma : \Delta, q, r \rightarrow s, \Gamma\) has a proof. The converse is also true. Why?
2.7 Cut-elimination and its consequences

The main theorem concerning sequent calculus proofs is called the cut-elimination theorem. A proof is cut-free if it contains no occurrences of the cut inference rule.

**Theorem 4** If a sequent has a C-proof (respectively, I-proof and M-proof) then it has a cut-free C-proof (respectively, I-proof and M-proof).

We shall prove this theorem in a later lecture. For now, I note that proofs of this theorem can be found in various places. Gentzen’s original proof [Gen69] is still quite readable. See also [Gal86, GTL89]. Semantic-based proofs are quite easy to give once one has a completeness theorem. Direct and constructive proofs can also be given and these result in procedures that can take a proof and systematically remove cut rules.

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

First, it is easy to show that logics satisfying the cut-elimination theorem are consistent. That is, it is easy to see that there can be no proof of \( \bot \): the only inference rules that could be used to prove the sequent \( \Sigma : \rightarrow \bot \) would be contrR and weakR, and these do not lead to a proof.

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

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

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

2.8 A one-sided sequent system for classical logic

It is possible to simplify the presentation of sequent calculus for classical logic. First, recall the following definition.

A formula is in negation normal form (nnf) if it contains no occurrences of $\supset$ and if negations have only atomic scope. Any formula can be rewritten to a classically equivalent formula that is in negation normal form by rewriting it by a series of simple equivalences. For example, $\neg\neg B \equiv B$ and $\neg (A \land B) \equiv (\neg A \lor \neg B)$.

Consider sequents of the form $\Sigma : \rightarrow \Gamma$ where $\Gamma$ is a set of formulas in nnf. Sequent calculus for this system is displayed in Figure 2.6. Here, we have organized things so that structural rules are not needed. Also, when we write $\neg B$ is a sequent formula, we mean the negation normal form of $\neg B$.

Notice that all inference rules are now invertible except for the $\exists R$ rule. To retain invertibility, one must ensure that the quantified formula $\exists x B$ occurs in the upper premise as well as the lower premise (via an implicit use of the contraction rule).

2.9 Additional readings

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

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

1. Provide proofs for each of the following sequents. Provide a \( I \)-proof only if there is no \( M \)-proof, and supply a \( C \)-proof only if there is no \( I \)-proof. Assume that the set of non-logical constants is \( \Sigma = \{ p : o, q : o, r : i \rightarrow o, a : i, b : i \} \).

   (a) \( p \land (p \supset q) \land (p \land q \supset s) \supset s \)
   (b) \( (p \supset q) \supset (\neg q \supset \neg p) \)
   (c) \( (\neg q \supset \neg p) \supset (p \supset q) \)
   (d) \( p \lor (p \supset q) \)
   (e) \( (r(a) \land r(b) \supset q) \supset \exists x(r(x) \supset q) \)
   (f) \( ((p \supset q) \supset p) \supset p \)
   (g) \( \exists y\forall x(r(x) \supset r(y)) \)

2. The multiplicative version of \( \land R \) would be the inference rule

\[
\begin{array}{c}
\Sigma : \Delta_1, \rightarrow B, \Gamma_1 \quad \Sigma : \Delta_2, \rightarrow C, \Gamma_2 \\
\hline
\Sigma : \Delta_1, \Delta_2, \rightarrow B \land C, \Gamma_1, \Gamma_2
\end{array}
\]

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

\[
\begin{array}{c}
\Sigma : B, \Delta_1, \rightarrow \Gamma_1 \quad \Sigma : C, \Delta_2, \rightarrow \Gamma_2 \\
\hline
\Sigma : B \lor C, \Delta_1, \Delta_2, \rightarrow \Gamma_1, \Gamma_2
\end{array}
\]

[Notice that the multiplicative form of this rule cannot be used with \( M \)-proofs. Is it possible to fix this for \( M \)-proofs?]

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

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

\[
\begin{align*}
clausal(A) &= 0 \quad \text{provided } A \text{ is atomic, } \top, \text{ or } \bot \\
clausal(B_1 \land B_2) &= \max(\clausal(B_1), \clausal(B_2)) \\
clausal(B_1 \lor B_2) &= \max(\clausal(B_1), \clausal(B_2)) \\
clausal(B_1 \supset B_2) &= \max(\clausal(B_1) + 1, \clausal(B_2)) \\
clausal(\forall x.B) &= \clausal(B) \\
clausal(\exists x.B) &= \clausal(B)
\end{align*}
\]
Let $n \geq 1$ and let $\Sigma : \Delta \rightarrow \Gamma$ be a sequent such that every formula in $\Delta$ is of order $n$ or less and every formula in $\Gamma$ is order $n-1$ or less. Prove that every sequent in a cut-free proof of $\Sigma : \Delta \rightarrow \Gamma$ has this same property.

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


7. List all inference rules that are invertible. Some might be invertible only with appropriate restrictions on the form of the proof. Why is observing invertibility important for, say, proof search?

8. Prove that the empty sequent, namely $\cdot : \cdot \rightarrow \cdot$, is provable if and only if all sequents are provable. Here, $\cdot$ denotes the empty signature and the empty multiset of formulas.

9. Prove that if the cut-elimination theorem holds, then classical, intuitionistic, and minimal logic are consistent (that is, they do not prove all sequents).

10. Change the proof system in Figures 2.3, 2.4, and 2.5 so that the structural rules are not needed: that is, they are somehow incorporated into the other rules of inference.

11. For the sake of this exercise, a Horn clause is a formula of the form

$$\forall x_1 \ldots \forall x_n. (A_1 \land \ldots \land A_m) \supset A_0$$

where $n$ and $m$ are non-negative and where $A_0, \ldots, A_m$ are atomic formulas. If $n = 0$ then we write no universal quantifiers and if $m = 0$, we drop the implication. We define also a Horn goal to be a formula that is the conjunction of atoms (the empty conjunction is denoted as $\top$). Write a specialized proof system for sequents of the form $\Sigma : \Delta \rightarrow \Gamma$ where $\Delta$ is a finite multiset of Horn clauses and $\Gamma$ is a finite multiset of Horn goals.

12. Prove the following: If $\Delta$ is a finite multiset of Horn clauses and $\Gamma$ is a finite multiset of Horn goals, then $\Sigma : \Delta \rightarrow \Gamma$ has a C-proof if and only if there is a $B \in \Gamma$ such that $\Sigma : \Delta \rightarrow B$ has a M-proof.

13. Take the equivalence $B \equiv B'$ to be an abbreviation for $(B \supset B') \land (B' \supset B)$. Prove that if $C[B]$ denotes some formula that contain an occurrence of $B$ as a subformula and $C[B']$ denotes the replacement of that subformula with $B'$, then $C[B] \equiv C[B']$. 

29
Chapter 3

Proof and Truth

In this chapter, we focus exclusively on classical first-order logic. We will present the soundness and completeness of the one-sided, sequent calculus proof system $\mathcal{G}$ given in Figure 2.6.

3.1 Herbrand Models

A Herbrand model is a particular kind of first-order model. The more general notion of first-order model will be presented later. For this Chapter, when we mention truth and validity, we shall think only of truth and validity within Herbrand models. As we shall see in later notes, if we restrict ourselves to first-order, classical logic without equality, a formula is true in all model if and only if it is true in all Herbrand models.

Herbrand models are somethings called term models since the domain to individuals is assumed to be terms instead of more general and more abstract objects. For example, in a Herbrand model, the three terms $1 + 2$, $2 + 1$, and $3$ are three different terms (i.e., syntactic expressions). In the more general notion of model it is possible to have the meaning of these terms be the same natural number $3$.

Definition 1 Assume that the set of non-logical constants (predicate, function, and individual constants) is fixed that it contains at least one individual constant. The Herbrand universe $\mathcal{H}$ is the set of all closed terms. A Herbrand model is a set of atomic formula in which the predicates are one of the fixed predicate constants and the terms are in $\mathcal{H}$ (these are the atomic formulas assumed to be truth in the model). Given a Herbrand model $M$, we define the binary relation $M \models F$ by induction on the structure of the first-order formula $F$ as follows.

- $F$ is atomic and $F \in \mathcal{H}$ then $M \models F$;
- $F$ is $F' \land F''$ then $F \in \mathcal{H}$ if $M \models F'$ and $M \models F''$;
- $F$ is $F' \lor F''$ then $F \in \mathcal{H}$ if $M \models F'$ or $M \models F''$;
• \( F \) is \( \neg F' \) then \( F \in H \) if \( M \models F \) does not hold;
• \( F \) is \( F' \supset F'' \) then \( F \in H \) if not \( M \models F' \) or \( M \models F'' \);
• \( F \) is \( \forall x. F' \) then \( F \in H \) if for all \( t \in H \), \( M \models F'[t/x] \);
• \( F \) is \( \exists x. F' \) then \( F \in H \) if for some \( t \in H \), \( M \models F'[t/x] \);

We shall find it convenient to reuse the symbol \( \models \) in the following notions: We write \( \models F \) (\( F \) is valid) if for all Herbrand models \( M \), we have \( M \models F \). If \( F \) and \( G \) are formulas, we write \( F \models G \) if for all Herbrand models \( M \), whenever \( M \models F \) then \( M \models G \).

Notice that if we are only interested in models for propositional formulas, the we have no need for a Herbrand universe: a model is the a set of propositional symbols (predicates of arity 0): in other words, a model is simply a column of a truth table in which the atoms in the model are exactly the propositional symbols attributed true in the truth-table.

3.2 Soundness of sequent calculus proofs

We extend the definitions of satisfiability and validity from formulas to sequents. We no longer write the arrow \( \rightarrow \) for one-sided sequents.

**Definition 2** Let \( \mathcal{M} \) be the model and let \( \Sigma : \Gamma \) be a sequent, where \( \Sigma = \{ x_1, \ldots, x_n \} \). We say that \( \Sigma : \Gamma \) is satisfiable in \( \mathcal{M} \) if for every list of terms \( t_1, \ldots, t_n \) there is an \( F \in \Gamma \) such that \( M \models F[t_1/x_1, \ldots, t_n/x_n] \). We write \( \mathcal{M} \models \Sigma : \Gamma \) if \( \Sigma : \Gamma \) is satisfiable in \( \mathcal{M} \). A sequent is unsatisfiable in \( \mathcal{M} \) if it is not satisfiable. The sequent \( \Sigma : \Gamma \) is valid, denoted as \( \models \Sigma : \Gamma \) if for every model \( \mathcal{M} \) we have \( \mathcal{M} \models \Sigma : \Gamma \).

What would an appropriate definitions of satisfiability and validity be for two-sided sequents?

We now state the soundness of the sequent system \( \mathcal{G} \).

**Theorem 5** If the sequent \( \Sigma : \Gamma \) is provable in \( \mathcal{G} \) then it is valid.

**Proof** This is a fairly straightforward proof by induction on the structure of proofs in system \( \mathcal{G} \).

3.3 Completeness of sequent calculus proofs

Proving the converse to soundness, namely, completeness is much more involved. First we state the main theorem of this chapter.

**Theorem 6** (Completeness) If the sequent \( \Sigma : \Gamma \) is valid then it is provable in \( \mathcal{G} \).
In fact, we will eventually prove the contrapositive of this statement, namely, that if a sequent does not have a proof then it is not valid. How can we know that a sequent does not have a proof? We shall present a systematic proof procedure that will search for a proof. If any \( G \) proof exists for our sequent, then our systematic proof mechanism will find a proof. If our systematic proof mechanism does not find a proof, then it will leave behind evidence of this failure, as either a finite or infinite tree. This failed tree will then contain the information we need to construct an interpretation that falsifies the sequent; meaning, of course, that the sequent is not valid.

### 3.3.1 Derivation trees

Since we shall be describing the “search” for proofs and since this search may result in successful proofs or in failures, we need to introduce a data structure that is more general than sequent calculus proofs. In particular, it will be “failed proofs” that are of particular interest in this chapter. Since the proof system \( G \) (Figure 2.6) only represents successful proofs, we need to define sometime more rich.

**Definition 3** A derivation tree is a rooted, labeled tree. It may be infinite but will, necessarily, be finite branching. Since such trees are rooted, we can talk about the in-arcs and out-arcs of node. Arcs of such a tree will be labeled by sequents of the following form:

\[
\Sigma : \mathcal{L}; \Gamma; \Phi
\]

where \( \mathcal{L} \) is a set of literals, \( \Gamma \) is a list of formula, and \( \Phi \) is a list of existentially quantified formulas. We say that the sequent \( \Sigma : \mathcal{L}; \Gamma; \Phi \) is satisfiable if and only if the sequent \( \Sigma : \mathcal{L}, \Gamma, \Phi \) is satisfiable (of course, we must coerce the list \( \Gamma \) and the set \( \mathcal{L} \) into multisets in the obvious way).
The nodes of a derivation tree are labeled with rules. There are three kinds of rules: improper, classification, and logical rules.

The three improper rules are: open, in which case, the node has one out-arc and no in-arcs; root, in which case, the node has one in-arc and no out-arc; and failure, in which case, the node has no in-arc, one out-arc, and where the out-arc is labeled with a sequent of the form \( \Sigma : \mathcal{L}; \; \vdash \; \cdot \) and \( \mathcal{L} \) does not contain complementary literals.

The classification rules correspond to the first three rules in Figure 3.1: these rules are responsible for moving literals and existential out of the middle context in order to place them into the respective contexts that are designed to store them. Obviously, if a node is labeled with one of these rules, then that node has one in-arc and one out-arc and those arcs are labeled with sequents that match those displayed in the inference rule.

The logical rules are the rules in Figure 3.2 and these correspond closely to the inference rules for proof system \( \mathcal{G} \) (Figure 2.6). If a node is labeled with one of these inference rules, then they all have one out-arc: they all have one in-arc, except for the \( \land R \) rule, which has two in-arcs, and the \( \top R \) rule which has zero in-arcs.

Note the following two particularities of the \( \exists R \) rule: the middle formula context must be empty in the sequent in the conclusion, and that the formula \( \exists_x \mathcal{B} \) is rotated from the front of the list of existential formulas in the lower sequent to the rear of that list in the upper sequent.

All derivation trees must have a node labeled root and this node must be, in fact, the root of the tree. We say that a given derivation tree, say, \( \Xi \), is a derivation for a sequent \( \Sigma : \mathcal{L}; \; \mathcal{G}; \; \Phi \) if this sequent is the label on the in-arc of the root node.

The smallest derivation tree is the two node tree where the root node is labeled with the open rule and the other arc is labeled with open rule. This tree denotes the initial derivation for the sequent labeling its sole edge and is called the start derivation tree.

We classify derivation trees in the following way. Let \( \Xi \) be a derivation tree. An open derivation is a tree that has a leaf labeled with the improper rule open. Informally, such trees can be extended. Assume now that \( \Xi \) is not an open (derivation) tree. If \( \Xi \) is finite, then it is either a proof tree if all its leaves are labeled with either the initial rule or the \( \top R \) rule; otherwise, a finite tree is called a finite failure tree. If \( \Xi \) is an infinite tree, then we say that it is an infinite failure tree.

### 3.3.2 Systematic derivation trees

We would like to be able to conclude that if we had a finite or infinite failure tree for a sequent then there is no proof of that sequent. This is clearly wrong: for example, there might be a proof for a sequent which involves choosing the correct terms to use the in \( \exists R \) rule. If instead, we always use other terms, we can build an infinite failure tree for a sequent with a proof. To fix this
problem, we shall introduce the notion of a particular kind of derivation tree, called a \textit{systematic derivation tree}, which is such that if we build a systematic derivation tree that is an infinite failure, then there is not proof.

We first note that all terms built from constants and variables form a denumerable set. We will fix an enumeration of all such terms. Thus, we will be able to speak of “the first term not present in a finite set of terms,” etc.

**Definition 4** A systematic derivation tree for the sequent $\Sigma : \mathcal{L}; \Gamma; \Phi$ is a derivation tree of this sequent defined inductively as follows.

1. The start derivation for this sequent is a systematic derivation tree.

2. If $\Xi$ is an open systematic derivation tree, then the replacement of an open node with either a classification or logical rule (Figures 3.1 and 3.2) or by the failure rule (in this latter case, the sequent is reclassified from something that is left to be attempted to one that is known not to be provable). This extension is governed by the following restrictions.

   (a) If we can chose the initial rule for making this extension, we do so. No other inference rule is permitted in this case.

   (b) If the rule used is $\exists R$, then the $\exists x \tau . B$ formula is instantiated with the first term $t$ that has not been used to instantiate $\exists x \tau . B$ before (that is, on the path from the this node to the root).

**Lemma 7** Given a sequent, if there is a systematic derivation tree that is a finite or infinite failure, then there is a model that makes the sequent unsatisfiable.

**Proof** Let $\Sigma : \mathcal{L}; \Gamma; \Phi$ be a sequent and let $\Xi$ be either a finite or infinite failure tree for $\Sigma : \mathcal{L}; \Gamma; \Phi$.

A path in a tree to a given node is a sequence of sequents $\theta = \theta_0, \theta_1, \ldots$ that label arcs from the root to the particular node. Let $\theta$ be a path in $\Xi$ that either ends in a failure node or is infinite. Notice that the first and third components increase monotonically as we move from the root higher up a tree.

We shall now show how to explicitly construct a model from $\theta$ in which the sequent $\Sigma : \mathcal{L}; \Gamma; \Phi$ is unsatisfiable. This model will be a Herbrand model. As such, to describe the model, we must only describe for all atomic formulas whether or not it is true or false.

Let $A$ be an atomic formula. If $A$ appears in the first component of some sequent in $\theta$, then we assign to $A$ the truth value $T$. If $\neg A$ appears in the first component of some sequent in $\theta$, then we assign to $A$ the truth value $F$. If $A$ does not appear in the first component of any sequent in $\theta$, then we can assign $A$ either $T$ or $F$. Of course, we need to show that this assignment is well defined, that is, that we are not assigning to $A$ both truth values. This can only happened if $A$ appears in some sequent, say, $\theta_i$, and $\neg A$ appears in some sequent, say, $\theta_j$. But if we set $k$ to be the maximum of $i$ and $j$, then $A$ and $\neg A$ both appear in $\theta_k$. By the definition of systematic derivation trees, $\theta_k$ must be
the consequence of the initial rule, which contradicts the choice of \( \theta \). Thus, the assignment of truth values to atoms is well defined.

Finally, we must prove the following by induction on the structure of formulas: If formula \( B \) occurs in some component of some sequent in \( \theta \), then \( B \) is false in the Herbrand model we have defined. Consider the cases for the structure of \( B \).

**B is a literal.** Then \( B \) is false by construction.

**B is \( C \land D \).** Assume that \( B \) appears in some sequent in \( \theta \). Eventually, this conjunction will become the first formula in the second component of some sequent, say, \( \theta_i \). Now, this sequent is the conclusion of two sequents, one of which is \( \theta_{i+1} \), which thus contains either \( C \) or \( D \). By the inductive assumption, either \( C \) or \( D \) is false in our model, so \( C \land D \) is false.

**B is \( C \lor D \).** Assume that \( B \) appears in some sequent in \( \theta \). Eventually, this disjunction will become the first formula in the second component of some sequent, say, \( \theta_i \). Now, this sequent is the conclusion of one sequent, namely, \( \theta_{i+1} \), which thus contains both \( C \) and \( D \). By the inductive assumption, both \( C \) and \( D \) are false in our model, so \( C \lor D \) is false.

**B is \( \forall x.C \).** Assume that \( B \) appears in some sequent in \( \theta \). Eventually, this formula will become the first formula in the second component of some sequent, say, \( \theta_i \). Now, this sequent is the conclusion of one sequent, namely, \( \theta_{i+1} \). By the inductive assumption, \( C[y/x] \) is false for some variable \( y \). Thus, \( \forall x.C \) is false.

**B is \( \exists x.C \).** Assume that \( B \) appears in some sequent in \( \theta \). Eventually, this formula will be a member of the third component of some sequent \( \theta_i \) (and in all following ones). To show that \( \exists x.C \) is false, we need to show that \( C[t/x] \) is false for all terms \( t \). Given the way the \( \exists R \) and the \( \exists \) rules are organized, every instantiation of \( \exists x.C \) will eventually be visited and entered into the path \( \theta \). Thus, by inductive hypothesis, all these instances will be smaller and, hence, false. Thus, \( \exists x.C \) is false.

This now completes the proof of this lemma.

We can now complete the proof of the Completeness Theorem (Theorem 6).

**Proof** Assume that the \( \mathcal{G} \)-sequent \( \Sigma : \Gamma \) is not provable. Consider a systematic derivation tree for \( \Sigma : \Gamma \). If this derivation tree is a proof, we must show that that systematic derivation tree can be translated to a \( \mathcal{G} \)-proof for the sequent \( \Sigma : \Gamma \). This is easily done. Thus, we would have a contradiction, and hence, a non-open systematic derivation tree is not a proof. The only other choices are that it is either a finite or infinite failure tree. Using Lemma 7, we conclude that \( \Sigma : \cdot ; \Gamma ; \cdot \) and, thus, \( \Sigma : \Gamma \) are unsatisfiable. We have thus proved the contrapositive of the completeness theorem.

**Theorem 8 (Löwenheim Theorem)** If a formula is satisfiable, it is satisfiable in a model with a denumerable domain.
The proof is left as an exercise.

### 3.4 Compactness

To make more use of our systematic derivation trees, we need to generalize them from working on one formula to working on denumerably many formulas. Fix a set $S$ to be a denumerable set of (closed) formulas that we may take as assumptions for attempting proofs. We shall now extend the notion of derivation tree to be $S$-derivation trees by allowing such trees to make use of the following “logical rule”,

$$\frac{\Sigma : \mathcal{L}; \Gamma, \neg B; \Phi}{\Sigma : \mathcal{L}; \Gamma; \Phi} S,$$

provided that $B \in S$. (Of course, we don’t mean literally $\neg B$ but the negation normal form of $\neg B$.) Notice that $B$ is placed at the end of the middle context (reading the rule bottom up). Similarly, we extend the notion of systematic derivation tree to systematic $S$-derivation trees by also doing the following. First, we order the formulas of $S$ (since it is denumerable). Second, we define the start derivation tree labeled with $\Sigma : \mathcal{L}; S_0; \mathcal{L}$ to be a systematic $S$-derivation tree. Finally, we allow trees to grow as before, except that after each extension of the tree by the replacement of an open node with something else, we require that we use the $S$ rule above to insert the next formula in $S$ into every open branch of the proof.

**Lemma 9** If the systematic $S$-derivation tree is a proof, then there is a finite subset of $S$ that is unsatisfiable. In other words, if all finite subsets of $S$ are satisfiable, then a non-open systematic $S$-derivation tree must be a failure.

The following theorem is also called the **Compactness Theorem** for first-order logic.

**Theorem 10 (Skolem-Löwenheim Theorem)** If all finite subsets of $S$ are satisfiable, then the entire set $S$ is satisfiable in a denumerable domain.

The proof is left as an exercise.

Comment on how this theorem forces non-standard models of arithmetic.

### 3.5 Additional readings

For a similar completeness theorem, but using tableaux proofs instead of sequent calculus proofs, see [Smu68]. See also Gallier’s text book [Gal86], particularly Sections 5.4 and 5.5.
3.6 Exercises

1. Consider the following three formulas.

\[ \forall x \exists y. R(x, y) \quad \neg \exists x. R(x, x) \quad \forall x \forall y \forall z [ R(x, y) \land R(y, z) \supset R(x, z) ] \]

Prove that the conjunction of these three formulas is not satisfiable in any model with a finite domain.

2. Prove the Löwenheim Theorem (Theorem 8).

3. Prove the Skolem-Löwenheim Theorem (Theorem 10).

4. Herbrand’s Theorem can be stated as follows: Let \( B \) be a quantifier free formula and let \( x_1, \ldots, x_n \) be a list of variables that contains the free variables of \( B \). The formula \( \exists x_1 \ldots \exists x_n. B \) is provable if and only if there are \( m \geq 1 \) substitutions \( \theta_1, \ldots, \theta_m \) such that \( B \theta_1 \lor \cdots \lor B \theta_m \) is tautologous. Prove this version of theorem.

5. The four color theorem for planar graphs states that any finite planar graph can be colored so that no two adjacent words have the same color. Given that the four color theorem holds, show that it must also hold for infinite graphs.
Chapter 4

Cut-elimination

In this chapter, we will continue our focus on classical first-order logic and particularly on the one-sided proof system $\mathcal{G}$ presented in Figure 2.6. We shall present the Cut-Elimination Theorem and several direct consequences of it: namely, the mid-sequent theorem, the Herbrand theorem, the interpolation theorem, and the equivalence of implicit and explicit definability.

4.1 Permutations of inference rules

Consider the two fragments of proofs in Figure 4.1. These illustrate how inference rules can be permuted. We shall see many more examples of such permutations in the cut-elimination theorem below.

$$
\begin{align*}
\Xi_1 & \quad \Sigma : \Gamma_1, B \\
\Xi_2 & \quad \Sigma : \Gamma_2, \neg B, C \\
\Xi_3 & \quad \Sigma : \Gamma_2, \neg B, D \\
\Xi_4 & \quad \Sigma : \Gamma_1, \Gamma_2, C \land D
\end{align*}
$$

- $\Sigma : \Gamma_1, \Gamma_2, C \land D$ (cut)
- $\Sigma : \Gamma_1, \Gamma_2, \neg B, C$ (cut)
- $\Sigma : \Gamma_1, \Gamma_2, \neg B, D$ (cut)

Figure 4.1: Examples of permutations of inference rules: permuting cut up over $\land R$.

4.2 Cut-elimination

We write $\Sigma \vdash_{\mathcal{G}} \Gamma$ to mean that the sequent $\Sigma : \Gamma$ is provable in the system $\mathcal{G}$ (Figure 2.6). We write $\Sigma \vdash_{\mathcal{G}}^{cf} \Gamma$ to mean that the sequent $\Sigma : \Gamma$ is provable in
the system $\mathcal{G}$ but not using its cut rule. The following is the cut-elimination theorem for $\mathcal{G}$.

**Theorem 11** Let $\Sigma$ be a signature and $\Gamma$ a set of formulas. The $\Sigma \vdash_{\mathcal{G}} \Gamma$ if and only if $\Sigma \vdash^{cf}_{\mathcal{G}} \Gamma$.

**Proof** [Semantic Proof] We have essentially already proved this result using the Completeness Theorem for $\mathcal{G}$ (Theorem 6). Derivation tree that are classified as proof trees actually encode cut-free proofs in $\mathcal{G}$. Assume that $\Sigma : \Gamma$ does not have a cut-free proof in $\mathcal{G}$. If we then consider the systematic derivation tree built starting with the sequent $\Sigma : : \Gamma$. As we can see, this process must result in a failure (finite or infinite) and, as a consequence of Lemma 7, the set $\Gamma$ is unsatisfiable. Thus, $\Sigma : \Gamma$ cannot have a (cut) proof in $\mathcal{G}$. We have thus proved the contrapositive form of the non-trivial part of this theorem.

We shall provide a second, more constructive and more general approach to cut-elimination. The following proof follows Gentzen’s original proof [Gen69] at least in the sense that we provide a systematic way to eliminating instances of the cut-rule by permuting them higher up the proof tree in such a way that cuts can eventually be removed entirely.

**Proof** [Proof-theoretic proof] Consider a proof $\Xi$ in $\mathcal{G}$ of the sequent $\Sigma : \Gamma$. For simplicity, we shall assume that $\Xi$ is atomically closed (see Exercise 3 in Chapter 2, page 28): that is, all instances of the initial rule in Figure 2.6 are such that the formula $B$ is atomic. Assume also that $\Xi$ has an instance of the cut rule, say,

\[
\begin{array}{c}
\Sigma \vdash_{\mathcal{G}} \Xi_1, B \\
\Sigma \vdash_{\mathcal{G}} \Xi_2, \neg B \\
\Sigma \vdash_{\mathcal{G}} \Xi_1 \cup \Xi_2, \neg B
\end{array}
\]

where $\Xi_1$ and $\Xi_2$ are proofs of the left and right premise, respectively. Recall that by $\neg B$ we refer to the negation normal form of the negation of $B$. We shall also say that such an occurrence of the cut inference rule is atomic if $B$ is atomic, and is non-atomic otherwise.

Consider the possibilities for the last inference rule for both $\Xi_1$ and $\Xi_2$. If the last inference rule for $\Xi_1$ is a rule in which $B$ is a side-formula (that is, appears in the “$\Gamma$” variable in the instance of that rule from Figure 2.6), then inference rule ending $\Xi_1$ can be permuted below the cut-rule (equivalently, the cut-rule can be permuted up over the cut-rule). Similarly, we can show that the if $\neg B$ is a side-formula of the last inference rule in $\Xi_2$, then cut can be permuted up similarly.

Thus, we consider the cases where $B$ is not the side formula in the last inference of $\Xi_1$ and $\neg B$ is not the side formula in the last inference of $\Xi_2$. First assume that the cut-rule is non-atomic, that is, $B$ is not an atomic formula. We consider the three cases for non-atomic $B$ and $\neg B$.

Case: $B$ is $B_1 \land B_2$ and $\neg B$ is $\neg B_1 \lor \neg B_2$ (the symmetric case where $B$ is a disjunction and $\neg B$ is a conjunction is essentially identical). Consider the
following cut inference rule for this conjunction/disjunction pair.

\[
\frac{\Sigma : \Gamma_1, B_1 \quad \Sigma : \Gamma_1, B_2 \quad \Sigma : \Gamma_2, \neg B_1, \neg B_2}{\Sigma : \Gamma_1, B_1 \land B_2 \quad \Sigma : \Gamma_2, B_1 \land \neg B_2 \quad \Sigma : \Gamma_1, \Gamma_2 \quad \text{cut}}
\]

This cut can be replace by two cuts, each with a smaller cut formulas, as follows.

\[
\frac{\Sigma : \Gamma_1, B_1 \quad \Sigma : \Gamma_1, B_2 \quad \Sigma : \Gamma_2, \neg B_1, \neg B_2}{\Sigma : \Gamma_1, \Gamma_2 \quad \text{cut}}
\]

Case: \( B \) is \( \forall x.B' \) and \( \neg B \) is \( \exists x.\neg B' \) (the symmetric case where \( B \) is existentially quantified and \( \neg B \) is universally quantified is essentially identical).

Consider the following cut inference rule for this quantifier pair.

\[
\frac{\Sigma : \Gamma_1, B'[y/x]}{\Sigma : \forall x.B' \quad \text{cut}}
\]

This cut can be replace by two cuts, each with a smaller cut formulas, as follows. Here, the expression \( \Xi_1[t/y] \) is the proof that results from replacing all occurrences of \( y \) in the sequents of that proof with \( t \). [One needs to prove that what results is, indeed, a proper \( G \) proof. This can be done using induction on the structure of proofs.]

\[
\frac{\Sigma : \Gamma_1, B'[y/x]}{\Sigma : \Gamma_1, \forall x.B' \quad \text{cut}}
\]

The only remaining non-atomic case is when \( B \) is \( \top \) and \( \neg B \) is \( \bot \). In that case, our instance of the cut rule is simply

\[
\frac{\Sigma : \Gamma_1, \top \quad \Sigma : \Gamma_2, \bot}{\Sigma : \Gamma_1, \Gamma_2 \quad \text{cut}}
\]

This cut can be completely eliminated by using the proof \( \Xi + \Gamma_1 \) which results from adding the set of formulas \( \Gamma_1 \) to all sequents in \( \Xi \): this structure is then a proof for the sequent \( \Sigma : \Gamma_1, \Gamma_2 \).

Using these observations, it is then possible to show that all non-atomic formulas can be removed for atomic cuts only (using induction of the structure of formulas). What induction measures can be used to formally show that this process of replacing cuts with (possibly) other cuts terminates? Notice that sometimes, one cut is replaced by two.
To remove atomic cuts, consider again the cut rule.

\[
\begin{align*}
\Sigma : \Gamma_1, B & \quad \Xi_1 \\
\Xi_2 & \quad \Sigma : \Gamma_2, \neg B \\
\hline
\Sigma : \Gamma_1, \Gamma_2 & \text{ cut}
\end{align*}
\]

Here, we are assuming that one of the premises is proved directly using the initial rule (otherwise, we would have permuted the cut rule higher on that side). If the last inference rule of \(\Xi_1\) is initial, then \(\neg B \in \Gamma_1\). Thus, this cut can be eliminated by using the proof \(\Xi_2 + (\Gamma_1 - \{\neg B\})\). Similarly, if the last inference rule of \(\Xi_2\) is initial, then \(B \in \Gamma_2\) and this cut can be eliminated by using the proof \(\Xi_1 + (\Gamma_2 - \{B\})\). Thus, by an induction of the structure of proofs, we can finally eliminate atomic cuts (leaving, of course, no cut rules).

Gentzen showed cut-elimination for both classical and intuitionistic logic using two-sided sequent systems. See [Gen69] for the original proof (quite readable). See also [Gal86] for versions of this theorem for a version of first-order classical logic also including equality. See [GTL89] for still other examples of this theorem. It is now common place that before one claims that a new inference system describes a logic, one must first prove that it satisfies a cut-elimination theorem. Of course, one might want many other results (completeness, compactness, etc), but this seems like an appropriate minimum property to establish.

### 4.3 The Mid-Sequent Theorem

In classical logic, it is formally possible to separate quantificational reasoning from propositional reasoning. Using the sequent calculus, it is possible to describe such a separation in a rather vivid fashion.

A formula is in prenex-normal form if no propositional connective in it (negation, conjunction, disjunction, implication) contain a quantifier in their scope. In other words, if the formula contains quantifiers, they have outermost scope only. Using simple equivalences of classical logic, for example,

\[
(\forall x.Bx) \lor C \equiv \forall x(Bx \lor C) \quad (\forall x.Bx) \land (\forall x.Cx) \equiv \forall x(Bx \land Cx) \quad \ldots
\]

any formula of first-order classical logic can be rewritten to an equivalent formula in prenex-normal form.

The following theorem is also sometimes called Gentzen’s Sharpened Hauptsatz Theorem.

**Theorem 12 (The mid-sequent theorem)** Given a sequent \(\Sigma : \Gamma\) containing only prenex-normal formulas, if \(\Sigma : \Gamma\) has a proof in \(\mathcal{G}\) then it has a cut-free proof in \(\mathcal{G}\) that contains a sequent \(\Sigma' : \Gamma'\) called the mid-sequent that has the following properties:

1. every formula in \(\Gamma'\) is quantifier-free;
2. every inference rule above \(\Sigma' : \Gamma'\) is a logical (not a quantifier rule); and
3. every inference rule below $\Sigma' : \Gamma'$ is a quantifier rule.

**Proof** Follows from rather simple observations of cut-free proofs and permutations of inference rules.

Thus, the mid-sequent separates a proof into a top part, which is propositional, and the lower part, which is quantificational.

### 4.4 Herbrand’s Theorem

The following was stated as an exercise in Chapter 3.

**Theorem 13 (Herbrand’s Theorem)** Let $B$ be a quantifier free formula and let $x_1, \ldots, x_n$ be a list of variables that contains the free variables of $B$. The formula $\exists x_1 \ldots \exists x_n B$ is provable if and only if there are $m \geq 1$ substitutions $\theta_1, \ldots, \theta_m$ such that $B \theta_1 \lor \cdots \lor B \theta_m$ is tautologous.

**Proof** This is a rather direct consequence of the mid-sequent theorem.

### 4.5 Interpolation

Given two formulas $B$ and $C$ such that $\vdash B \supset C$, an **interpolant** is a formula $R$ such that $\vdash B \supset R$ and $\vdash R \supset C$ and all non-logical constants occurring in $R$ appear in both $B$ and $C$. That is, interpolants can only contain constants that appear in both $B$ and $C$. (Interpolation theorems go back to Craig [Cra57].)

**Theorem 14** Given any two formulas $B$ and $C$ such that $\Sigma \vdash B \supset C$, there is an interpolant $R$ for this implication.

**Proof** The proof of this will be described in class. Basically, it makes use of the modified sequent system in Figure 4.2 in which the boxed formulas are the interpolants. Note that this inference system is essentially a cut-free proof system. The one item not explained in Figure 4.2 is the meaning of the syntax $\forall \bar{x}$ and $\exists \bar{x}$ in the last two inference rules. Consider the second last of these rules: when one moves from $B[[t/x]]$ to $\exists x B$, there might be constants, say, $a_1, \ldots, a_n$ that occurs in $B[[t/x]]$ and in $\Delta$ but do not occur in $\Gamma \cup \{\exists x B\}$. In that case, they must be hidden in the interpolant in the conclusion of the inference rule. This is done by setting the interpolant to be $\forall x_1 \ldots \forall x_n R[x_1/a_1, \ldots, x_n/a_n]$ (the meaning of $\forall \bar{x}$). For the other inference rule, a dual quantification is needed for hiding, and it uses a series of existential quantifiers instead.

### 4.6 Implicit and explicit definability

Let $\Gamma$ be a finite set of formulas, let $P$ be a predicate of one argument, and let $P'$ be a predicate of one argument that is different from $P$ and does not occur in $\Gamma$. Let $\Gamma'$ be a copy of $\Gamma$ where all occurrences of $P$ are replaced by $P'$. We say that $\Gamma$ **implicitly defines** $P$ if it is the case that $\Gamma, \Gamma' \vdash \forall x [P(x) \equiv P'(x)]$. 

43
Figure 4.2: Modification of the proof system $\mathcal{G}$ to help compute interpolants. The notion $\forall$ and $\exists$ are explained in the proof of Theorem 14.

**Theorem 15** If $\Gamma$ implicitly defines $P$ then there is an explicit definition of $P$ from $\Gamma$: that is, there is a formula $\phi(x)$ of one free variable $x$ which does not contain $P$ and is such that $\Gamma \vdash \forall x [P(x) \equiv \phi(x)]$.

**Proof** The proof involves a simple use of the interpolation theorem. Assume that $\Gamma$ implicitly defines $P$. Thus, $\vdash (\Gamma \land \Gamma') \supset \forall x [P(x) \equiv P'(x)]$ and $\vdash \Gamma \land P(c) \supset (\Gamma' \supset P'(c))$. Here, $c$ is some new constant symbol. Let $R$ be an interpolant for the above implication. Thus, $\vdash (\Gamma \land P(c)) \supset R$ and $\vdash R \supset (\Gamma' \supset P'(c))$ and the latter implication entails $\vdash \Gamma' \supset (R \supset P'(c))$. The proof of the latter sequent also yields a proof of $\vdash \Gamma \supset (R \supset P(c))$ by replacing all occurrences of $P'$ in that proof with $P$. Thus, we have $\vdash \Gamma \supset (R \equiv P(c))$. If we now set $\phi(x)$ to be the formula $R[x/c]$, then we finally have $\vdash \Gamma \supset \forall x (\phi(x) \equiv P(x))$, using the $\forall R$ inference rule. 

*Note:* The notion $P(x)$ for an atomic formula and $\phi(c)$ for an instance of a formula are confusing.
4.7 Exercises

1. We say that a formula $B$ is a subformula of $C$ is the usual way with respect to propositional connectives: $B$ is a subformula of $C_1 \land C_2$ if $B$ is either equal to this formula or it is a subformula of either $C_1$ or $C_2$; $B$ is a subformula of $C_1 \lor C_2$ if $B$ is either equal to this formula or it is a subformula of either $C_1$ or $C_2$; $B$ is a subformula of $\neg C_1$ if $B$ is either equal to this formula or it is a subformula of either $C_1$; $B$ is a subformula of $\forall x.C_1$ if $B$ is either equal to this formula or it is a subformula of $C_1[t/x]$ for some term $t$; and finally $B$ is a subformula of $\exists x.C_1$ if $B$ is either equal to this formula or it is a subformula of $C_1[t/x]$ for some term $t$. Prove the following subformula property for cut-free proofs: Let $\Xi$ be a cut-free proof of the sequent $\Sigma : \Gamma$ and let $\Sigma : \Gamma'$ be some sequent appearing in $\Xi$. If $B$ is a formula in $\Gamma'$ then $B$ is a subformula of some formula in $\Gamma$.

2. Prove the following equivalence: The empty sequent is provable if and only if there is a formula $B$ such that $\Sigma \vdash G B$ and $\Sigma \vdash G \neg B$. If neither of these conditions hold, we say that the proof system is consistent. Prove that $G$ is consistent (give a semantic proof and a proof theoretic proof of this statement.)

3. Compute an interpolate for the following pairs of formulas.
   $$(q\, a) \land \forall x(q\, x \supset p\, x) \quad \text{and} \quad (q\, b) \supset (p\, b)$$
   $$\forall x(p\, x \supset r\, x) \land \forall x(r\, x \supset q\, x) \quad \text{and} \quad (p\, b) \supset (q\, b)$$

4. Consider adding a definition mechanism to the propositional fragment of $G$. That is, allow a definition to be a finite list of pairs, written
   $$\{p_1 \triangleq B_1, \ldots, p_n \triangleq B_n\}$$
   where $n \geq 0$, the symbols $p_1, \ldots, p_n$ are distinct propositional constants, and $B_1, \ldots, B_n$ are propositional formulas. Extend the inference rules for $G$ with the following two rules (we drop the signature used in sequents since we are only dealing with the propositional fragment here).
   $$(\Gamma, B_i, \Gamma, p_i \triangleright \Gamma, \neg p_i)$$
   Although these rules seem quite natural, the resulting system does not, in general, have the cut-elimination theorem.
   (a) Show that cut-elimination fails in general by considering the definition $\{p \triangleq \neg p\}$.
   (b) Show that using the definition $\{p \triangleq \neg p\}$, we have an inconsistent proof system.
   (c) Prove that if the formulas $B_1, \ldots, B_n$ do not contain negations, then cut-elimination holds.
Bibliography


