Logic and Proof

Computer Science Tripos Part IB

Mateja Jamnik
Department of Computer Science and Technology
University of Cambridge

`mateja.jamnik@cl.cam.ac.uk`

Contents

1 Introduction and Learning Guide

This course is a brief introduction to logic, including the resolution method of theorem-proving and its relation to the language Prolog. Formal logic is used for specifying and verifying computer systems.

The course should help you to understand Prolog and is a prerequisite for more advanced verification courses. It describes many techniques used in automated theorem provers. Understanding the various deductive methods is a crucial part of the course, but you should also try to acquire some intuitions about logic.

Although the course notes are self-contained, some students will want to read alternative treatments of the material. A suitable course text is

> Michael Huth and Mark Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd edition (CUP, 2004)

It covers most topics save resolution theorem proving. It includes material (symbolic model checking) that could be useful later.

The following book may be a useful supplement to Huth and Ryan. It covers resolution and other relevant topics.

> Mordechai Ben-Ari, *Mathematical Logic for Computer Science*, 2nd edition (Springer, 2001)

The following book provides a different perspective on modal logic, and it develops propositional logic carefully.

> Sally Popkorn, *First Steps in Modal Logic* (CUP, 2008)

The following paper is a wonderful exposition of the workings and power of SAT solvers. It is available for download within the University.

> Marijn Heule and Oliver Kullmann. The Science of Brute Force. *CACM* **60** (2017), 70–79. `http://dl.acm.org/citation.cfm?id=3107239`

There are numerous exercises in these notes, and they are suitable for supervision purposes. Most old examination questions for *Foundations of Logic Programming* (the former name of this course) are still relevant. As of 2013/14, Herbrand's theorem has been somewhat deprecated, still mentioned but no longer in detail. Some unification theory has also been removed. These changes created space for a new lecture on Decision Procedures and SMT Solvers.

- 2017 Paper 6 Q5: modal logic and resolution
- 2017 Paper 6 Q6: BDDs and the sequent calculus
- 2016 Paper 6 Q5: clause methods and resolution
- 2016 Paper 6 Q6: SMT solving and BDDs
- 2015 Paper 6 Q5: resolution and factoring
- 2015 Paper 6 Q6: Boolean satisfiability; sequent / tableaux
- 2014 Paper 6 Q5: propositional proof: BDDs and DPLL
- 2014 Paper 6 Q6: decision procedures; variant of resolution
- 2013 Paper 6 Q5: DPLL, sequent or tableau calculus
- 2013 Paper 6 Q6: resolution problems
- 2012 Paper 6 Q6: sequents / tableaux, modal logic, BDDs
- 2011 Paper 6 Q5: resolution, linear resolution, BDDs
- 2011 Paper 6 Q6: unification, modal logic
- 2010 Paper 6 Q5: BDDs and models
- 2010 Paper 6 Q6: sequent or tableau calculus, DPLL. **Note**: the formula should be $(\exists x\, P(x) \to Q) \to \forall x\, (P(x) \to Q)$.
- 2008 Paper 3 Q6: BDDs, DPLL, sequent calculus
- 2008 Paper 4 Q5: (dis)proving first-order formulas, resolution
- 2009 Paper 6 Q7: modal logic
- 2009 Paper 6 Q8: resolution, tableau calculi
- 2007 Paper 5 Q9: propositional methods, resolution, modal logic
- 2007 Paper 6 Q9: proving or disproving first-order formulas
- 2006 Paper 5 Q9: proof and disproof in FOL and modal logic
- 2005 Paper 5 Q9: resolution
- 2005 Paper 6 Q9: DPLL, BDDs, tableaux
- 2004 Paper 5 Q9: semantics and proof in FOL
- 2003 Paper 5 Q9: BDDs; clause-based proof methods
- 2003 Paper 6 Q9: sequent calculus
- 2002 Paper 5 Q11: semantics of first-order logic
- 2002 Paper 6 Q11: resolution; proof systems
- 2001 Paper 5 Q11: satisfaction relation; logical equivalences
- 2001 Paper 6 Q11: clause methods; *ternary* decision diagrams
- 2000 Paper 5 Q11: tautology checking; sequent calculus
- 1999 Paper 5 Q10: Prolog versus general resolution
- 1998 Paper 5 Q10: BDDs, sequent calculus, etc.
- 1998 Paper 6 Q10: modal logic; resolution
- 1997 Paper 5 Q10: first-order logic
- 1997 Paper 6 Q10: sequent rules for quantifiers
- 1996 Paper 5 Q10: sequent calculus
- 1996 Paper 6 Q10: DPLL versus Resolution
- 1995 Paper 5 Q9: BDDs
- 1995 Paper 6 Q9: outline logics; sequent calculus

Acknowledgements. These notes were originally created by Lawrence C. Paulson. Chloë Brown, Jonathan Davies and Reuben Thomas pointed out numerous errors in these notes. David Richerby and Ross Younger made detailed suggestions. Thanks also to Nathanael Alcock, Julia Bibik, Darren Foong, Thomas Forster, Simon Frankau, Adam Hall, Ximin Luo, Roddy MacSween, Priyesh Patel, Steve Payne, Kuba Perlin, Tom Puverle, Max Spencer, Nik Sultana, Ben Thorner, Tjark Weber and John Wickerson.

2 Propositional Logic

Propositional logic deals with truth values and the logical connectives *and*, *or*, *not*, etc. Most of the concepts in propositional logic have counterparts in first-order logic. Here are the most fundamental concepts.

Syntax refers to the formal notation for writing assertions. It also refers to the data structures that represent assertions in a computer. At the level of syntax, 1 + 2 is a string of three symbols, or a tree with a node labelled + and having two children labelled 1 and 2.

Semantics expresses the meaning of a formula in terms of mathematical or real-world entities. While 1 + 2 and 2 + 1 are syntactically distinct, they have the same semantics, namely 3. The semantics of a logical statement will typically be true or false.

Proof theory concerns ways of proving statements, at least the true ones. Typically we begin with *axioms* and arrive at other true statements using *inference rules*. Formal proofs are typically finite and mechanical: their correctness can be checked without understanding anything about the subject matter.

Syntax can be represented in a computer. Proof methods are syntactic, so they can be performed by computer. On the other hand, as semantics is concerned with meaning, it exists only inside people's heads. This is analogous to the way computers handle digital photos: the computer has no conception of what your photos mean to you, and internally they are nothing but bits.

2.1 Syntax of propositional logic

Take a set of *propositional symbols* P, Q, R, ... A formula consisting of a propositional symbol is called *atomic*. We use **t** and **f** to denote true and false.

Formulas are constructed from atomic formulas using the logical connectives[1]

$\neg$	(not)
$\land$	(and)
$\lor$	(or)
$\rightarrow$	(implies)
$\leftrightarrow$	(if and only if)

These are listed in order of precedence; $\neg$ is highest. We shall suppress needless parentheses, writing, for example,

$$(((\neg P) \land Q) \lor R) \rightarrow ((\neg P) \lor Q) \text{ as } \neg P \land Q \lor R \rightarrow \neg P \lor Q.$$

In the *metalanguage* (these notes), the letters A, B, C, ... stand for arbitrary formulas. The letters P, Q, R, ... stand for atomic formulas.

2.2 Semantics

Propositional Logic is a formal language. Each formula has a meaning (or semantics) — either 1 or 0 — relative to the meaning of the propositional symbols it contains. The meaning can be calculated using the standard truth tables.

A	B	$\neg A$	$A \land B$	$A \lor B$	$A \rightarrow B$	$A \leftrightarrow B$
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

By inspecting the table, we can see that $A \rightarrow B$ is equivalent to $\neg A \lor B$ and that $A \leftrightarrow B$ is equivalent to $(A \rightarrow B) \land (B \rightarrow A)$. (The latter is also equivalent to $\neg(A \oplus B)$, where $\oplus$ is exclusive-or.)

Note that we are using **t** and **f** in the language as *symbols* to denote the truth values 1 and 0. The former belongs to syntax, the latter to semantics. When it comes to first-order logic, we shall spend some time on the distinction between symbols and their meanings.

We now make some definitions that will be needed throughout the course.

Definition 1 An *interpretation*, or *truth assignment*, for a set of formulas is a function from its set of propositional symbols to $\{1, 0\}$.

An interpretation *satisfies* a formula if the formula evaluates to 1 under the interpretation.

A set S of formulas is *valid* (or a *tautology*) if every interpretation for S satisfies every formula in S.

A set S of formulas is *satisfiable* if there is some interpretation for S that satisfies every formula in S.

A set S of formulas is *unsatisfiable* if it is not satisfiable.

A set S of formulas *entails* A if every interpretation that satisfies all elements of S, also satisfies A. Write $S \models A$.

Formulas A and B are *equivalent*, $A \simeq B$, provided $A \models B$ and $B \models A$.

Some relationships hold among these primitives. Note the following in particular:

- $S \models A$ if and only if $\{\neg A\} \cup S$ is unsatisfiable.
- If S is unsatisfiable, then $S \models A$ for any A. This is an instance of the phenomenon that we can deduce anything from a contradiction.
- $\models A$ if and only if A is valid, if and only if $\{\neg A\}$ is unsatisfiable.

It is usual to write $A \models B$ instead of $\{A\} \models B$. We may similarly identify a one-element set with a formula in the other definitions.

Note that $\models$ and $\simeq$ are not logical connectives but relations between formulas. They belong not to the logic but to the metalanguage: they are symbols we use to discuss the logic. They therefore have lower precedence than the logical connectives. No parentheses are needed in $A \land A \simeq A$ because the only possible reading is $(A \land A) \simeq A$. We may not write $A \land (A \simeq A)$ because $A \simeq A$ is not a formula.

In propositional logic, a valid formula is also called a *tautology*. Here are some examples of these definitions.

- The formulas $A \rightarrow A$ and $\neg(A \land \neg A)$ are valid for every formula A.

[1] Using $\supset$ for implies and $\equiv$ for if-and-only-if is archaic.

- The formulas P and $P \land (P \to Q)$ are satisfiable: they are both true under the interpretation that maps P and Q to 1. But they are not valid: they are both false under the interpretation that maps P and Q to 0.
- If A is a valid formula then $\neg A$ is unsatisfiable.
- This set of formulas is unsatisfiable: $\{P, Q, \neg P \lor \neg Q\}$.

2.3 Applications of propositional logic

In hardware design, propositional logic has long been used to minimize the number of gates in a circuit, and to show the equivalence of combinational circuits. There now exist highly efficient tautology checkers, such as BDDs (Binary Decision Diagrams), which can verify complex combinational circuits. This is an important branch of hardware verification. The advent of efficient SAT solvers has produced an explosion of applications involving approximating various phenomena as large propositional formulas, typically through some process of iterative refinement.

Chemical synthesis is a more offbeat example.[2] Under suitable conditions, the following chemical reactions are possible:

$$\begin{aligned} \mathrm{HCl} + \mathrm{NaOH} &\to \mathrm{NaCl} + \mathrm{H_2O} \\ \mathrm{C} + \mathrm{O_2} &\to \mathrm{CO_2} \\ \mathrm{CO_2} + \mathrm{H_2O} &\to \mathrm{H_2CO_3} \end{aligned}$$

Show we can make H_2CO_3 given supplies of HCl, NaOH, O_2, and C.

Chang and Lee formalize the supplies of chemicals as four axioms and prove that H_2CO_3 logically follows. The idea is to formalize each compound as a propositional symbol and express the reactions as implications:

$$\begin{aligned} \mathrm{HCl} \land \mathrm{NaOH} &\to \mathrm{NaCl} \land \mathrm{H_2O} \\ \mathrm{C} \land \mathrm{O_2} &\to \mathrm{CO_2} \\ \mathrm{CO_2} \land \mathrm{H_2O} &\to \mathrm{H_2CO_3} \end{aligned}$$

Note that this involves an ideal model of chemistry. What if the reactions can be inhibited by the presence of other chemicals? Proofs about the real world *always* depend upon general assumptions. It is essential to bear these in mind when relying on such a proof.

2.4 Equivalences

Note that $A \leftrightarrow B$ and $A \simeq B$ are different kinds of assertions. The formula $A \leftrightarrow B$ refers to some fixed interpretation, while the metalanguage statement $A \simeq B$ refers to all interpretations. On the other hand, $\models A \leftrightarrow B$ means the same thing as $A \simeq B$. Both are metalanguage statements, and $A \simeq B$ is equivalent to saying that the formula $A \leftrightarrow B$ is a tautology.

Similarly, $A \to B$ and $A \models B$ are different kinds of assertions, while $\models A \to B$ and $A \models B$ mean the same thing. The formula $A \to B$ is a tautology if and only if $A \models B$.

Here is a listing of some of the more basic equivalences of propositional logic. They provide one means of reasoning about propositions, namely by transforming one proposition into an equivalent one. They are also needed to convert propositions into various normal forms.

idempotency laws

$$A \land A \simeq A$$
$$A \lor A \simeq A$$

commutative laws

$$A \land B \simeq B \land A$$
$$A \lor B \simeq B \lor A$$

associative laws

$$(A \land B) \land C \simeq A \land (B \land C)$$
$$(A \lor B) \lor C \simeq A \lor (B \lor C)$$

distributive laws

$$A \lor (B \land C) \simeq (A \lor B) \land (A \lor C)$$
$$A \land (B \lor C) \simeq (A \land B) \lor (A \land C)$$

de Morgan laws

$$\neg(A \land B) \simeq \neg A \lor \neg B$$
$$\neg(A \lor B) \simeq \neg A \land \neg B$$

other negation laws

$$\neg(A \to B) \simeq A \land \neg B$$
$$\neg(A \leftrightarrow B) \simeq (\neg A) \leftrightarrow B \simeq A \leftrightarrow (\neg B)$$

laws for eliminating certain connectives

$$A \leftrightarrow B \simeq (A \to B) \land (B \to A)$$
$$\neg A \simeq A \to \mathbf{f}$$
$$A \to B \simeq \neg A \lor B$$

simplification laws

$$A \land \mathbf{f} \simeq \mathbf{f}$$
$$A \land \mathbf{t} \simeq A$$
$$A \lor \mathbf{f} \simeq A$$
$$A \lor \mathbf{t} \simeq \mathbf{t}$$
$$\neg\neg A \simeq A$$
$$A \lor \neg A \simeq \mathbf{t}$$
$$A \land \neg A \simeq \mathbf{f}$$

Propositional logic enjoys a principle of *duality*: for every equivalence $A \simeq B$ there is another equivalence $A' \simeq B'$, derived by exchanging $\land$ with $\lor$ and $\mathbf{t}$ with $\mathbf{f}$. Before applying this rule, remove all occurrences of $\to$ and $\leftrightarrow$, since they implicitly involve $\land$ and $\lor$.

[2] Chang and Lee, page 21, as amended by Ross Younger, who knew more about Chemistry!

2.5 Normal forms

The language of propositional logic has much redundancy: many of the connectives can be defined in terms of others. By repeatedly applying certain equivalences, we can transform a formula into a *normal form*. A typical normal form eliminates certain connectives and uses others in a restricted manner. The restricted structure makes the formula easy to process, although the normal form may be much larger than the original formula, and unreadable.

Definition 2 *(Normal Forms)*

- A *literal* is an atomic formula or its negation. Let K, L, L', ... stand for literals.
- A formula is in *Negation Normal Form* (NNF) if the only connectives in it are $\land$, $\lor$, and $\neg$, where $\neg$ is only applied to atomic formulas.
- A formula is in *Conjunctive Normal Form* (CNF) if it has the form $A_1 \land \cdots \land A_m$, where each A_i is a disjunction of one or more literals.
- A formula is in *Disjunctive Normal Form* (DNF) if it has the form $A_1 \lor \cdots \lor A_m$, where each A_i is a conjunction of one or more literals.

An atomic formula like P is in all the normal forms NNF, CNF, and DNF. The formula

$$(P \lor Q) \land (\neg P \lor S) \land (R \lor P)$$

is in CNF. Unlike in some hardware applications, the disjuncts in a CNF formula do not have to mention all the variables. On the contrary, they should be as simple as possible. Simplifying the formula

$$(P \lor Q) \land (\neg P \lor Q) \land (R \lor S)$$

to $Q \land (R \lor S)$ counts as an improvement, because it will make our proof procedures run faster. For examples of DNF formulas, exchange $\land$ and $\lor$ in the examples above. As with CNF, there is no need to mention all combinations of variables.

NNF can reveal the underlying nature of a formula. For example, converting $\neg(A \to B)$ to NNF yields $A \land \neg B$. This reveals that the original formula was effectively a conjunction. Every formula in CNF or DNF is also in NNF, but the NNF formula $((\neg P \land Q) \lor R) \land P$ is in neither CNF nor DNF.

2.6 Translation to normal form

Every formula can be translated into an equivalent formula in NNF, CNF, or DNF by means of the following steps.

Step 1. Eliminate $\leftrightarrow$ and $\to$ by repeatedly applying the following equivalences:

$$A \leftrightarrow B \simeq (A \to B) \land (B \to A)$$
$$A \to B \simeq \neg A \lor B$$

Step 2. Push negations in until they apply only to atoms, repeatedly replacing by the equivalences

$$\neg\neg A \simeq A$$
$$\neg(A \land B) \simeq \neg A \lor \neg B$$
$$\neg(A \lor B) \simeq \neg A \land \neg B$$

At this point, the formula is in Negation Normal Form.

Step 3. To obtain CNF, push disjunctions in until they apply only to literals. Repeatedly replace by the equivalences

$$A \lor (B \land C) \simeq (A \lor B) \land (A \lor C)$$
$$(B \land C) \lor A \simeq (B \lor A) \land (C \lor A)$$

These two equivalences obviously say the same thing, since disjunction is commutative. In fact, we have

$$(A \land B) \lor (C \land D) \simeq (A \lor C) \land (A \lor D) \land (B \lor C) \land (B \lor D).$$

Use this equivalence when you can, to save writing.

Step 4. Simplify the resulting CNF by deleting any disjunction that contains both P and $\neg P$, since it is equivalent to $\mathbf{t}$. Also delete any conjunct that includes another conjunct (meaning, every literal in the latter is also present in the former). This is correct because $(A \lor B) \land A \simeq A$. Finally, two disjunctions of the form $P \lor A$ and $\neg P \lor A$ can be replaced by A, thanks to the equivalence

$$(P \lor A) \land (\neg P \lor A) \simeq A.$$

This simplification is related to the resolution rule, which we shall study later.

Since $\lor$ is commutative, a conjunct of the form $A \lor B$ could denote any possible way of arranging the literals into two parts. This includes $A \lor \mathbf{f}$, since one of those parts may be empty and the empty disjunction is false. So in the last simplification above, two conjuncts of the form P and $\neg P$ can be replaced by $\mathbf{f}$.

Steps 3' and 4'. To obtain DNF, apply instead the other distributive law:

$$A \land (B \lor C) \simeq (A \land B) \lor (A \land C)$$
$$(B \lor C) \land A \simeq (B \land A) \lor (C \land A)$$

Exactly the same simplifications can be performed for DNF as for CNF, exchanging the roles of $\land$ and $\lor$.

2.7 Tautology checking using CNF

Here is a (totally impractical) method of proving theorems in propositional logic. To prove A, reduce it to CNF. If the simplified CNF formula is $\mathbf{t}$ then A is valid because each transformation preserves logical equivalence. And if the CNF formula is not $\mathbf{t}$, then A is not valid.

To see why, suppose the CNF formula is $A_1 \land \cdots \land A_m$. If A is valid then each A_i must also be valid. Write A_i as $L_1 \lor \cdots \lor L_n$, where the L_j are literals. We can make an interpretation I that falsifies every L_j, and therefore falsifies A_i. Define I such that, for every propositional letter P,

$$I(P) = \begin{cases} 0 & \text{if } L_j \text{ is } P \text{ for some } j \\ 1 & \text{if } L_j \text{ is } \neg P \text{ for some } j \end{cases}$$

This definition is legitimate because there cannot exist literals L_j and L_k such that L_j is $\neg L_k$; if there did, then simplification would have deleted the disjunction A_i.

The powerful BDD method is based on similar ideas, but uses an if-then-else data structure, an ordering on the propositional letters, and some standard algorithmic techniques (such as hashing) to gain efficiency.

Example 1 Start with

$$P \vee Q \to Q \vee R.$$

Step 1, eliminate →, gives

$$\neg(P \vee Q) \vee (Q \vee R).$$

Step 2, push negations in, gives

$$(\neg P \wedge \neg Q) \vee (Q \vee R).$$

Step 3, push disjunctions in, gives

$$(\neg P \vee Q \vee R) \wedge (\neg Q \vee Q \vee R).$$

Simplifying yields $(\neg P \vee Q \vee R) \wedge \mathbf{t}$ and then

$$\neg P \vee Q \vee R.$$

The interpretation $P \mapsto 1$, $Q \mapsto 0$, $R \mapsto 0$ falsifies this formula, which is equivalent to the original formula. So the original formula is not valid.

Example 2 Start with

$$P \wedge Q \to Q \wedge P$$

Step 1, eliminate →, gives

$$\neg(P \wedge Q) \vee Q \wedge P$$

Step 2, push negations in, gives

$$(\neg P \vee \neg Q) \vee (Q \wedge P)$$

Step 3, push disjunctions in, gives

$$(\neg P \vee \neg Q \vee Q) \wedge (\neg P \vee \neg Q \vee P)$$

Simplifying yields $\mathbf{t} \wedge \mathbf{t}$, which is $\mathbf{t}$. Both conjuncts are valid since they contain a formula and its negation. Thus $P \wedge Q \to Q \wedge P$ is valid.

Example 3 Peirce's law is another example. Start with

$$((P \to Q) \to P) \to P$$

Step 1, eliminate →, gives

$$\neg(\neg(\neg P \vee Q) \vee P) \vee P$$

Step 2, push negations in, gives

$$(\neg\neg(\neg P \vee Q) \wedge \neg P) \vee P$$

$$((\neg P \vee Q) \wedge \neg P) \vee P$$

Step 3, push disjunctions in, gives

$$(\neg P \vee Q \vee P) \wedge (\neg P \vee P)$$

Simplifying again yields $\mathbf{t}$. Thus Peirce's law is valid.

There is a dual method of refuting A (proving inconsistency). To refute A, reduce it to DNF, say $A_1 \vee \cdots \vee A_m$. If A is unsatisfiable then so is each A_i. Suppose A_i is $L_1 \wedge \cdots \wedge L_n$, where the L_j are literals. If there is some literal L' such that the L_j include both L' and $\neg L'$, then A_i is unsatisfiable. If not then there is an interpretation that verifies every L_j, and therefore A_i.

To prove A, we can use the DNF method to refute $\neg A$. The steps are exactly the same as the CNF method because the extra negation swaps every $\vee$ and $\wedge$. Gilmore implemented a theorem prover based upon this method in 1960.

Exercise 1 Is the formula $P \to \neg P$ satisfiable, or valid?

Exercise 2 Verify the de Morgan and distributive laws using truth tables.

Exercise 3 Each of the following formulas is satisfiable but not valid. Exhibit an interpretation that makes the formula true and another that makes the formula false.

$$P \to Q \qquad\qquad P \vee Q \to P \wedge Q$$
$$\neg(P \vee Q \vee R) \qquad \neg(P \wedge Q) \wedge \neg(Q \vee R) \wedge (P \vee R)$$

Exercise 4 Convert each of the following propositional formulas into Conjunctive Normal Form and also into Disjunctive Normal Form. For each formula, state whether it is valid, satisfiable, or unsatisfiable; justify each answer.

$$(P \to Q) \wedge (Q \to P)$$
$$((P \wedge Q) \vee R) \wedge \neg(P \vee R)$$
$$\neg(P \vee Q \vee R) \vee ((P \wedge Q) \vee R)$$

Exercise 5 Using ML, define datatypes for representing propositions and interpretations. Write a function to test whether or not a proposition holds under an interpretation (both supplied as arguments). Write a function to convert a proposition to Negation Normal Form.

3 Proof Systems for Propositional Logic

We can verify any tautology by checking all possible interpretations, using the truth tables. This is a *semantic* approach, since it appeals to the meanings of the connectives.

The *syntactic* approach is formal proof: generating theorems, or reducing a conjecture to a known theorem, by applying syntactic transformations of some sort. We have already seen a proof method based on CNF. Most proof methods are based on axioms and inference rules.

What about efficiency? Deciding whether a propositional formula is satisfiable is an NP-complete problem (Aho, Hopcroft and Ullman 1974, pages 377–383). Thus all approaches are likely to be exponential in the length of the formula. Technologies such as BDDs and SAT solvers, which can decide huge problems in propositional logic, are all the more stunning because their success was wholly unexpected. But even they require a "well-behaved" input formula and are exponential in the worst case.

3.1 A Hilbert-style proof system

Here is a simple proof system for propositional logic. There are countless similar systems. They are often called *Hilbert systems* after the logician David Hilbert, although they existed before him.

This proof system provides rules for implication only. The other logical connectives are not taken as primitive. They are instead *defined* in terms of implication:

$$\neg A \overset{\text{def}}{=} A \to \mathbf{f}$$
$$A \lor B \overset{\text{def}}{=} \neg A \to B$$
$$A \land B \overset{\text{def}}{=} \neg(\neg A \lor \neg B)$$

Obviously, these definitions apply *only* when we are discussing this proof system!

Note that $A \to (B \to A)$ is a tautology. Call it Axiom K. Also,

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

is a tautology. Call it Axiom S. The Double-Negation Law $\neg\neg A \to A$, is a tautology. Call it Axiom DN.

These axioms are more properly called *axiom schemes*, since we assume all instances of them that can be obtained by substituting formulas for A, B and C. For example, Axiom K is really an infinite set of formulas.

Whenever $A \to B$ and A are both valid, it follows that B is valid. We write this as the inference rule

$$\frac{A \to B \quad A}{B.}$$

This rule is traditionally called Modus Ponens. Together with Axioms K, S, and DN and the definitions, it suffices to prove all tautologies of classical propositional logic.However, this formalization of propositional logic is inconvenient to use. For example, try proving $A \to A$!

A variant of this proof system replaces the Double-Negation Law by the Contrapositive Law:

$$(\neg B \to \neg A) \to (A \to B)$$

Remark: If the Double-Negation Law is simply omitted, we get *intuitionistic logic*. This formal system is motivated by a philosophy of constructive mathematics. It has close connections with advanced topics including type theory and the combinators S and K in the λ-calculus. Many of the familiar identities of boolean algebra, even $A \lor \neg A$, do not hold in intuitionistic logic.

Another formalization of propositional logic consists of the Modus Ponens rule plus the following axioms:

$$A \lor A \to A$$
$$B \to A \lor B$$
$$A \lor B \to B \lor A$$
$$(B \to C) \to (A \lor B \to A \lor C)$$

Here $A \land B$ and $A \to B$ are defined in terms of $\neg$ and $\lor$.

Where do truth tables fit into all this? Truth tables define the *semantics*, while proof systems define what is sometimes called the *proof theory*. A proof system must respect the truth tables. Above all, we expect the proof system to be *sound*: every theorem it generates must be a tautology. For this to hold, every axiom must be a tautology and every inference rule must yield a tautology when it is applied to tautologies.

The converse property is *completeness*: the proof system can generate every tautology. Completeness is harder to achieve and show. There are complete proof systems even for first-order logic. (And Gödel's *incompleteness* theorem uses the word "completeness" with a different technical meaning.)

Soundness and completeness are all we need for the case of propositional logic. But if we generalise and consider formal systems without an obvious semantics, then soundness and completeness are not applicable. A more general property of a proof system is *consistency*: not generating contradictory theorems, such as A and $\neg A$ for some A. Typically, an inconsistent formal system makes A a theorem for **every** formula A. In this case the very concept of a theorem becomes vacuous. Any formal system that is sound for propositional logic must necessarily be consistent.

3.2 Gentzen's Natural Deduction Systems

Natural proof systems do exist. Natural deduction, devised by Gerhard Gentzen, is based upon three principles:

1. Proof takes place within a varying context of assumptions.

2. Each logical connective is defined independently of the others. (This is possible because item 1 eliminates the need for tricky uses of implication.)

3. Each connective is defined by *introduction* and *elimination* rules.

For example, the *introduction* rule for $\land$ describes how to deduce $A \land B$:

$$\frac{A \quad B}{A \land B}\ (\land i)$$

The *elimination* rules for $\land$ describe what to deduce *from* $A \land B$:

$$\frac{A \land B}{A}\ (\land e1) \qquad \frac{A \land B}{B}\ (\land e2)$$

The elimination rule for $\to$ says what to deduce from $A \to B$. It is just Modus Ponens:

$$\frac{A \to B \quad A}{B}\ (\to e)$$

The introduction rule for → says that $A \to B$ is proved by assuming A and deriving B:

$$\frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A \to B}\;(\to i)$$

For simple proofs, this notion of assumption is pretty intuitive. Here is a proof of the formula $A \land B \to A$:

$$\frac{\dfrac{[A \land B]}{A}\;(\land e1)}{A \land B \to A}\;(\to i)$$

The key point is that rule $(\to i)$ *discharges* its assumption: the assumption could be used to prove A from $A \land B$, but is no longer available once we conclude $A \land B \to A$.

The introduction rules for ∨ are straightforward:

$$\frac{A}{A \lor B}\;(\lor i1) \qquad \frac{B}{A \lor B}\;(\lor i2)$$

The elimination rule says that to show some C from $A \lor B$ there are two cases to consider, one assuming A and one assuming B:

$$\frac{A \lor B \quad \begin{array}{c}[A]\\ \vdots\\ C\end{array} \quad \begin{array}{c}[B]\\ \vdots\\ C\end{array}}{C}\;(\lor e)$$

The scope of assumptions can get confusing in complex proofs. Let us switch attention to the sequent calculus, which is similar in spirit but easier to use.

3.3 The sequent calculus

The *sequent calculus* resembles natural deduction, but it makes the set of assumptions explicit. Thus, it is more concrete.

A *sequent* has the form $\Gamma \Rightarrow \Delta$, where Γ and Δ are finite sets of formulas.[3] These sets may be empty. The sequent

$$A_1, \ldots, A_m \Rightarrow B_1, \ldots, B_n$$

is *true* (in a particular interpretation) if $A_1 \land \ldots \land A_m$ implies $B_1 \lor \ldots \lor B_n$. In other words, if each of $A_1, \ldots, A_m$ are true, then at least one of $B_1, \ldots, B_n$ must be true. The sequent is *valid* if it is true in *all* interpretations.

A *basic* sequent is one in which the same formula appears on both sides, as in $P, B \Rightarrow B, R$. This sequent is valid because, if all the formulas on the left side are true, then in particular B is; so, at least one right-side formula (B again) is true. Our calculus therefore regards all basic sequents as proved.

Every basic sequent might be written in the form $\{A\} \cup \Gamma \Rightarrow \{A\} \cup \Delta$, where A is the common formula and Γ and Δ are the other left- and right-side formulas, respectively. The sequent calculus identifies the one-element set $\{A\}$ with its element A and denotes union by a comma. Thus, the correct notation for the general form of a basic sequent is $A, \Gamma \Rightarrow A, \Delta$.

Sequent rules are almost always used backward. We start with the sequent that we would like to prove. We view the sequent as saying that $A_1, \ldots, A_m$ are true, and we try to show that one of $B_1, \ldots, B_n$ is true. Working backwards, we use sequent rules to reduce it to simpler sequents, stopping when those sequents become trivial. The forward direction would be to start with known facts and derive new facts, but this approach tends to generate random theorems rather than ones we want.

Sequent rules are classified as *right* or *left*, indicating which side of the $\Rightarrow$ symbol they operate on. Rules that operate on the right side are analogous to natural deduction's introduction rules, and left rules are analogous to elimination rules.

The sequent calculus analogue of $(\to i)$ is the rule

$$\frac{A, \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \to B}\;(\to r)$$

Working backwards, this rule breaks down some implication on the right side of a sequent; Γ and Δ stand for the sets of formulas that are unaffected by the inference. The analogue of the pair $(\lor i1)$ and $(\lor i2)$ is the single rule

$$\frac{\Gamma \Rightarrow \Delta, A, B}{\Gamma \Rightarrow \Delta, A \lor B}\;(\lor r)$$

This breaks down some disjunction on the right side, replacing it by both disjuncts. Thus, the sequent calculus is a kind of multiple-conclusion logic. Figure 1 summarises the rules.

Let us prove that the rule $(\lor l)$ is sound. We must show that if both premises are valid, then so is the conclusion. For contradiction, assume that the conclusion, $A \lor B, \Gamma \Rightarrow \Delta$, is *not* valid. Then there exists an interpretation I under which the left side is true while the right side is false; in particular, $A \lor B$ and Γ are true while Δ is false. Since $A \lor B$ is true under interpretation I, either A is true or B is. In the former case, $A, \Gamma \Rightarrow \Delta$ is false; in the latter case, $B, \Gamma \Rightarrow \Delta$ is false. Either case contradicts the assumption that the premises are valid.

3.4 Examples of Sequent Calculus Proofs

To illustrate the use of multiple formulas on the right, let us prove the classical theorem $(A \to B) \lor (B \to A)$. Working backwards (or upwards), we reduce this formula to a basic sequent:

$$\cfrac{\cfrac{\cfrac{\overline{A, B \Rightarrow B, A}}{A \Rightarrow B, B \to A}\;(\to r)}{\Rightarrow A \to B,\; B \to A}\;(\to r)}{\Rightarrow (A \to B) \lor (B \to A)}\;(\lor r)$$

The basic sequent has a line over it to emphasize that it is provable.

This example is typical of the sequent calculus: start with the desired theorem and work *upward*. Notice that inference rules still have the same logical meaning, namely that the premises (above the line) imply the conclusion (below the line). Instead of matching a rule's premises with facts that we know, we match its conclusion with the formula we want

[3] With minor changes, sequents can instead be lists or multisets.

basic sequent: $A, \Gamma \Rightarrow A, \Delta$

Negation rules:

$$\frac{\Gamma \Rightarrow \Delta, A}{\neg A, \Gamma \Rightarrow \Delta}\,(\neg l) \qquad \frac{A, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \neg A}\,(\neg r)$$

Conjunction rules:

$$\frac{A, B, \Gamma \Rightarrow \Delta}{A \land B, \Gamma \Rightarrow \Delta}\,(\land l) \qquad \frac{\Gamma \Rightarrow \Delta, A \quad \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \land B}\,(\land r)$$

Disjunction rules:

$$\frac{A, \Gamma \Rightarrow \Delta \quad B, \Gamma \Rightarrow \Delta}{A \lor B, \Gamma \Rightarrow \Delta}\,(\lor l) \qquad \frac{\Gamma \Rightarrow \Delta, A, B}{\Gamma \Rightarrow \Delta, A \lor B}\,(\lor r)$$

Implication rules:

$$\frac{\Gamma \Rightarrow \Delta, A \quad B, \Gamma \Rightarrow \Delta}{A \to B, \Gamma \Rightarrow \Delta}\,(\to l) \qquad \frac{A, \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \to B}\,(\to r)$$

Figure 1: Sequent Rules for Propositional Logic

to prove. That way, the form of the desired theorem controls the proof search.

The distributive law $A \lor (B \land C) \simeq (A \lor B) \land (A \lor C)$ is proved (one direction at least) as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{A \Rightarrow A, B} \quad \dfrac{\overline{B, C \Rightarrow A, B}}{B \land C \Rightarrow A, B}\,(\land l)}{A \lor (B \land C) \Rightarrow A, B}\,(\lor l)}{A \lor (B \land C) \Rightarrow A \lor B}\,(\lor r) \qquad \text{similar}}{A \lor (B \land C) \Rightarrow (A \lor B) \land (A \lor C)}\,(\land r)$$

The second, omitted proof tree proves $A \lor (B \land C) \Rightarrow A \lor C$ similarly.

Finally, here is a failed proof of the invalid formula $A \lor B \to B \lor C$.

$$\dfrac{\dfrac{\dfrac{A \Rightarrow B, C \quad \overline{B \Rightarrow B, C}}{A \lor B \Rightarrow B, C}\,(\lor l)}{A \lor B \Rightarrow B \lor C}\,(\lor r)}{\Rightarrow A \lor B \to B \lor C}\,(\to r)$$

The sequent $A \Rightarrow B, C$ has no line over it because it is not valid! The interpretation $A \mapsto 1$, $B \mapsto 0$, $C \mapsto 0$ falsifies it. We have already seen this as Example 1 (page 5).

3.5 Further Sequent Calculus Rules

Structural rules concern sequents in general rather than particular connectives. They are little used in this course, because they are not useful for proof procedures. However, a brief mention is essential in any introduction to the sequent calculus.

The *weakening* rules allow additional formulas to be inserted on the left or right side. Obviously, if $\Gamma \Rightarrow \Delta$ holds then the sequent continues to hold after further assumptions or goals are added. When writing a proof from the bottom up, these rules are useful for discarding unwanted formulas.

$$\frac{\Gamma \Rightarrow \Delta}{A, \Gamma \Rightarrow \Delta}\,(\mathit{weaken{:}l}) \qquad \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, A}\,(\mathit{weaken{:}r})$$

Exchange rules allow formulas in a sequent to be re-ordered. We do not need them because our sequents are sets rather than lists. *Contraction* rules allow formulas to be used more than once, for when writing a proof from the bottom upwards, their effect is to duplicate a formula.

$$\frac{A, A, \Gamma \Rightarrow \Delta}{A, \Gamma \Rightarrow \Delta}\,(\mathit{contract{:}l}) \qquad \frac{\Gamma \Rightarrow \Delta, A, A}{\Gamma \Rightarrow \Delta, A}\,(\mathit{contract{:}r})$$

Because the sets $\{A\}$ and $\{A, A\}$ are identical, we don't need contraction rules either. Moreover, it turns out that we almost never need to use a formula more than once. Exceptions are $\forall x\, A$ (when it appears on the left) and $\exists x\, A$ (when it appears on the right).

The *cut rule* allows the use of lemmas. Some formula A is proved in the first premise, and assumed in the second premise. A famous result, the *cut-elimination theorem*, states that this rule is not required. All uses of it can be removed from any proof (at the cost of exponential blowup).

$$\frac{\Gamma \Rightarrow \Delta, A \quad A, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\,(\mathit{cut})$$

This special case of cut may be easier to understand. We prove lemma A from Γ and use A and Γ together to reach the conclusion B.

$$\frac{\Gamma \Rightarrow B, A \quad A, \Gamma \Rightarrow B}{\Gamma \Rightarrow B}$$

Since Γ contains as much information as A, it is natural to expect that such lemmas should not be necessary, but the cut-elimination theorem is hard to prove.

Historical note Backward proof using the sequent calculus is the foundation of Wang's [**?**] algorithms for propositional and first-order logic. His work was a landmark in the history of artificial intelligence. Curiously enough, historians of AI often highlight more heuristic approaches (notably the Logic Theorist of Newell, Shaw and Simon), done a couple of years earlier, although Wang achieved spectacularly better results.

Exercise 6 Prove the following sequents:

$$\neg\neg A \Rightarrow A$$
$$A \land B \Rightarrow B \land A$$
$$A \lor B \Rightarrow B \lor A$$

Exercise 7 Prove the following sequents:

$$(A \land B) \land C \Rightarrow A \land (B \land C)$$
$$(A \lor B) \land (A \lor C) \Rightarrow A \lor (B \land C)$$
$$\neg(A \lor B) \Rightarrow \neg A \land \neg B$$

Exercise 8 Derive the sequent calculus rules for the connectives $\leftrightarrow$ and $\oplus$ (exclusive or). Note that other connectives must not appear in these rules.

Exercise 9 Prove the following sequents:

$$\Rightarrow (A \land \neg A) \to B$$
$$\Rightarrow ((A \to B) \to A) \to A$$

Extension: a sequent calculus for intuitionistic first-order logic can be obtained by using the one above but imposing the restriction that at no point in a proof may more than one formula appear on the right side. Determine whether the two sequents above can be proved under that condition.

4 First-order Logic

First-order logic (FOL) extends propositional logic to allow reasoning about the members (such as numbers) of some non-empty universe. It uses the quantifiers $\forall$ (*for all*) and $\exists$ (*there exists*). First-order logic has variables ranging over so-called individuals, but not over functions or predicates; such variables are found in second- or higher-order logic.

4.1 Syntax of first-order Logic

Terms stand for individuals while *formulas* stand for truth values. We assume there is an infinite supply of *variables* x, y, … that range over individuals. A *first-order language* specifies symbols that may appear in terms and formulas. A first-order language $\mathcal{L}$ contains, for all $n \geq 0$, a set of n-place *function symbols* f, g, … and n-place *predicate symbols* P, Q, …. These sets may be empty, finite, or infinite.

Constant symbols a, b, … are simply 0-place function symbols. Intuitively, they are names for fixed elements of the universe. It is not required to have a constant for each element; conversely, two constants are allowed to have the same meaning.

Predicate symbols are also called *relation symbols*. Prolog programmers refer to function symbols as *functors*.

Definition 3 The *terms* $t, u, \ldots$ of a first-order language are defined recursively as follows:

- A variable is a term.
- A constant symbol is a term.
- If t_1, …, t_n are terms and f is an n-place function symbol then $f(t_1, \ldots, t_n)$ is a term.

Definition 4 The *formulas* A, B, … of a first-order language are defined recursively as follows:

- If $t_1, \ldots, t_n$ are terms and P is an n-place predicate symbol then $P(t_1, \ldots, t_n)$ is a formula (called an *atomic formula*).
- If A and B are formulas then $\neg A$, $A \land B$, $A \lor B$, $A \to B$, $A \leftrightarrow B$ are also formulas.
- If x is a variable and A is a formula then $\forall x\, A$ and $\exists x\, A$ are also formulas.

Brackets are used in the conventional way for grouping. Terms and formulas are tree-like data structures, not strings.

The quantifiers $\forall x\, A$ and $\exists x\, A$ bind tighter than the binary connectives; thus $\forall x\, A \land B$ is equivalent to $(\forall x\, A) \land B$. Frequently, you will see an alternative quantifier syntax, $\forall x\, .\, A$ and $\exists x\, .\, B$, which binds more weakly than the binary connectives: $\forall x\, .\, A \land B$ is equivalent to $\forall x\, (A \land B)$. The dot is the give-away; look out for it!

Nested quantifications such as $\forall x\, \forall y\, A$ are abbreviated to $\forall xy\, A$.

Example 4 A language for arithmetic might have the constant symbols 0, 1, 2, …, and function symbols $+$, $-$, $\times$, $/$, and the predicate symbols $=$, $<$, $>$, …. We informally may adopt an infix notation for the function and predicate symbols. Terms include 0 and $(x + 3) - y$; formulas include $y = 0$ and $x + y < y + z$.

4.2 Examples of first-order statements

Here are some sample formulas with a rough English translation. English is easier to understand but is too ambiguous for long derivations.

All professors are brilliant:

$$\forall x\, (\text{professor}(x) \to \text{brilliant}(x))$$

The income of any banker is greater than the income of any bedder:

$$\forall xy\, (\text{banker}(x) \land \text{bedder}(y) \to \text{income}(x) > \text{income}(y))$$

Note that $>$ is a 2-place relation symbol. The infix notation is simply a convention.

Every student has a supervisor:

$$\forall x\, (\text{student}(x) \to \exists y\, \text{supervises}(y, x))$$

This does not preclude a student having several supervisors.

Every student's tutor is a member of the student's College:

$$\begin{aligned}\forall xy\, (\text{student}(x) \land \text{college}(y) \land \text{member}(x, y) \\ \to \text{member}(\text{tutor}(x), y))\end{aligned}$$

Formalising the notion of tutor as a function incorporates the assumption that every student has exactly *one* tutor.

A mathematical example: *there exist infinitely many Pythagorean triples*:

$$\forall n\, \exists ijk\, (i > n \land i^2 + j^2 = k^2)$$

Here the superscript 2 refers to the squaring function. Equality ($=$) is just another relation symbol (satisfying suitable axioms) but there are many special techniques for it.

First-order logic requires a non-empty domain: thus $\forall x\, P(x)$ implies $\exists x\, P(x)$. If the domain could be empty, even $\exists x\, \mathbf{t}$ could fail to hold. Note also that $\forall x\, \exists y\, y^2 = x$ is true if the domain is the complex numbers, and is false if the domain is the integers or reals. We determine properties of the domain by asserting the set of statements it must satisfy.

There are many other forms of logic. *Many-sorted first-order logic* assigns types to each variable, function symbol and predicate symbol, with straightforward type checking; types are called *sorts* and denote non-empty domains. *Second-order logic* allows quantification over functions and predicates. It can express mathematical induction by

$$\forall P\, [P(0) \land \forall k\, (P(k) \to P(k + 1)) \to \forall n\, P(n)],$$

using quantification over the unary predicate P. In second-order logic, these functions and predicates must themselves be first-order, taking no functions or predicates as arguments. *Higher-order logic* allows unrestricted quantification over functions and predicates of any order. The list of logics could be continued indefinitely.

4.3 Formal semantics of first-order logic

Let us rigorously define the meaning of formulas. An interpretation of a language maps its function symbols to actual functions, and its relation symbols to actual relations. For example, the predicate symbol "student" could be mapped to the set of all students currently enrolled at the University.

Definition 5 Let $\mathcal{L}$ be a first-order language. An *interpretation* $\mathcal{I}$ of $\mathcal{L}$ is a pair (D, I). Here D is a nonempty set, the *domain* or *universe*. The operation I maps symbols to individuals, functions or sets:

- if c is a constant symbol (of $\mathcal{L}$) then $I[c] \in D$
- if f is an n-place function symbol then $I[f] \in D^n \to D$ (which means $I[f]$ is an n-place function on D)
- if P is an n-place relation symbol then $I[P] \in D^n \to \{1, 0\}$ (equivalently, $I[P] \subseteq D^n$, which means $I[P]$ is an n-place relation on D)

It is natural to regard predicates as truth-valued functions. But in first-order logic, relations and functions are distinct concepts because no term can denote a truth value. One of the benefits of higher-order logic is that relations are a special case of functions, and formulas are simply boolean-valued terms.

An interpretation does not say anything about variables. An environment or *valuation* can be used to represent the values of variables.

Definition 6 A *valuation* V of $\mathcal{L}$ over D is a function from the variables of $\mathcal{L}$ into D. Write $\mathcal{I}_V[t]$ for the value of t with respect to $\mathcal{I}$ and V, defined by

$$\begin{aligned}\mathcal{I}_V[x] &\overset{\text{def}}{=} V(x) \qquad \text{if } x \text{ is a variable}\\ \mathcal{I}_V[c] &\overset{\text{def}}{=} I[c]\\ \mathcal{I}_V[f(t_1,\ldots,t_n)] &\overset{\text{def}}{=} I[f](\mathcal{I}_V[t_1],\ldots,\mathcal{I}_V[t_n])\end{aligned}$$

Write $V\{a/x\}$ for the valuation that maps x to a and is otherwise the same as V. Typically, we modify a valuation one variable at a time. This is a semantic analogue of substitution for the variable x.

4.4 What is truth?

We can define truth in first-order logic. This formidable definition formalizes the intuitive meanings of the connectives. Thus it almost looks like a tautology. It effectively specifies each connective by English descriptions. Valuations help specify the meanings of quantifiers. Alfred Tarski first defined truth in this manner.

Definition 7 Let A be a formula. Then for an interpretation $\mathcal{I} = (D, I)$ write $\models_{\mathcal{I},V} A$ to mean that A is true in $\mathcal{I}$ under V. This is defined by cases on the construction of the formula A:

$\models_{\mathcal{I},V} P(t_1,\ldots,t_n)$ is defined to hold if

$$I[P](\mathcal{I}_V[t_1],\ldots,\mathcal{I}_V[t_n]) = 1$$

(that is, the actual relation $I[P]$ is true for the given values)

$\models_{\mathcal{I},V} t = u$ if $\mathcal{I}_V[t]$ equals $\mathcal{I}_V[u]$ (if = is a predicate symbol of the language, then we insist that it really denotes equality)

$\models_{\mathcal{I},V} \neg B$ if $\models_{\mathcal{I},V} B$ does not hold

$\models_{\mathcal{I},V} B \land C$ if $\models_{\mathcal{I},V} B$ and $\models_{\mathcal{I},V} C$

$\models_{\mathcal{I},V} B \lor C$ if $\models_{\mathcal{I},V} B$ or $\models_{\mathcal{I},V} C$

$\models_{\mathcal{I},V} B \to C$ if $\models_{\mathcal{I},V} B$ does not hold or $\models_{\mathcal{I},V} C$ holds

$\models_{\mathcal{I},V} B \leftrightarrow C$ if $\models_{\mathcal{I},V} B$ and $\models_{\mathcal{I},V} C$ both hold or neither hold

$\models_{\mathcal{I},V} \exists x\, B$ if there exists some $m \in D$ such that $\models_{\mathcal{I},V\{m/x\}} B$ holds (that is, B holds when x has the value m)

$\models_{\mathcal{I},V} \forall x\, B$ if $\models_{\mathcal{I},V\{m/x\}} B$ holds for all $m \in D$

The cases for $\land$, $\lor$, $\to$ and $\leftrightarrow$ follow the propositional truth tables.

Write $\models_{\mathcal{I}} A$ provided $\models_{\mathcal{I},V} A$ for all V. Clearly, if A is closed (contains no free variables) then its truth is independent of the valuation. The definitions of valid, satisfiable, etc. carry over almost verbatim from Sect. 2.2.

Definition 8 Let A be a formula having no free variables.

- An interpretation $\mathcal{I}$ *satisfies* a formula if $\models_{\mathcal{I}} A$ holds.
- A set S of formulas is *valid* if every interpretation of S satisfies every formula in S.
- A set S of formulas is *satisfiable* if there is some interpretation that satisfies every formula in S.
- A set S of formulas is *unsatisfiable* if it is not satisfiable. (Each interpretation falsifies some formula of S.)
- A *model* of a set S of formulas is an interpretation that satisfies every formula in S. We also consider models that satisfy a single formula.

Unlike in propositional logic, models can be infinite and there can be an infinite number of models. There is no chance of proving validity by checking all models. We must rely on proof.

Example 5 The formula $P(a) \land \neg P(b)$ is satisfiable. Consider the interpretation with $D = \{\text{London}, \text{Paris}\}$ and I defined by

$$\begin{aligned}I[a] &= \text{Paris}\\ I[b] &= \text{London}\\ I[P] &= \{\text{Paris}\}\end{aligned}$$

On the other hand, $\forall xy\,(P(x) \land \neg P(y))$ is unsatisfiable because it requires $P(x)$ to be both true and false for all x. Also unsatisfiable is $P(x) \land \neg P(y)$: its free variables are taken to be universally quantified, so it is equivalent to $\forall xy\,(P(x) \land \neg P(y))$.

The formula $(\exists x\, P(x)) \to P(c)$ holds in the interpretation (D, I) where $D = \{0, 1\}$, $I[P] = \{0\}$, and $I[c] = 0$.

(Thus $P(x)$ means "x equals 0" and c denotes 0.) If we modify this interpretation by making $I[c] = 1$ then the formula no longer holds. Thus it is satisfiable but not valid.

The formula $(\forall x\, P(x)) \rightarrow (\forall x\, P(f(x)))$ is valid, for let (D, I) be an interpretation. If $\forall x\, P(x)$ holds in this interpretation then $P(x)$ holds for all $x \in D$, thus $I[P] = D$. The symbol f denotes some actual function $I[f] \in D \rightarrow D$. Since $I[P] = D$ and $I[f](x) \in D$ for all $x \in D$, formula $\forall x\, P(f(x))$ holds.

The formula $\forall xy\, x = y$ is satisfiable but not valid; it is true in every domain that consists of exactly one element. (The empty domain is not allowed in first-order logic.)

Example 6 Let $\mathcal{L}$ be the first-order language consisting of the constant 0 and the (infix) 2-place function symbol $+$. An interpretation $\mathcal{I}$ of this language is any non-empty domain D together with values $I[0]$ and $I[+]$, with $I[0] \in D$ and $I[+] \in D \times D \rightarrow D$. In the language $\mathcal{L}$ we may express the following axioms:

$$x + 0 = x$$
$$0 + x = x$$
$$(x + y) + z = x + (y + z)$$

(Remember, free variables in effect are universally quantified, by the definition of $\models_{\mathcal{I}} A$.) One model of these axioms is the set of natural numbers, provided we give 0 and + the obvious meanings. But the axioms have many other models.[4] Below, let A be some set.

1. The set of all strings (in ML say) letting 0 denote the empty string and + string concatenation.
2. The set of all subsets of A, letting 0 denote the empty set and + union.
3. The set of functions in $A \rightarrow A$, letting 0 denote the identity function and + composition.

Exercise 10 To test your understanding of quantifiers, consider the following formulas: *everybody loves somebody* vs *there is somebody that everybody loves*:

$$\forall x\, \exists y\, \text{loves}(x, y) \tag{1}$$
$$\exists y\, \forall x\, \text{loves}(x, y) \tag{2}$$

Does (1) imply (2)? Does (2) imply (1)? Consider both the informal meaning and the formal semantics defined above.

Exercise 11 Describe a formula that is true in precisely those domains that contain at least m elements. (We say it *characterises* those domains.) Describe a formula that characterises the domains containing fewer than m elements.

Exercise 12 Let $\approx$ be a 2-place predicate symbol, which we write using infix notation as $x \approx y$ instead of $\approx (x, y)$. Consider the axioms

$$\forall x\, x \approx x \tag{1}$$
$$\forall xy\, (x \approx y \rightarrow y \approx x) \tag{2}$$
$$\forall xyz\, (x \approx y \land y \approx z \rightarrow x \approx z) \tag{3}$$

Let the universe be the set of natural numbers, $N = \{0, 1, 2, \ldots\}$. Which axioms hold if $I[\approx]$ is

1. the empty relation, $\emptyset$?
2. the universal relation, $\{(x, y) \mid x, y \in N\}$?
3. the equality relation, $\{(x, x) \mid x \in N\}$?
4. the relation $\{(x, y) \mid x, y \in N \land x + y \text{ is even}\}$?
5. the relation $\{(x, y) \mid x, y \in N \land x + y = 100\}$?
6. the relation $\{(x, y) \mid x, y \in N \land x \leq y\}$?

Exercise 13 Taking $=$ and R as 2-place relation symbols, consider the following axioms:

$$\forall x\, \neg R(x, x) \tag{1}$$
$$\forall xy\, \neg(R(x, y) \land R(y, x)) \tag{2}$$
$$\forall xyz\, (R(x, y) \land R(y, z) \rightarrow R(x, z)) \tag{3}$$
$$\forall xy\, (R(x, y) \lor (x = y) \lor R(y, x)) \tag{4}$$
$$\forall xz\, (R(x, z) \rightarrow \exists y\, (R(x, y) \land R(y, z))) \tag{5}$$

Exhibit two interpretations that satisfy axioms 1–5. Exhibit two interpretations that satisfy axioms 1–4 and falsify axiom 5. Exhibit two interpretations that satisfy axioms 1–3 and falsify axioms 4 and 5. Consider only interpretations that make = denote the equality relation. (This exercise asks whether you can make the connection between the axioms and typical mathematical objects satisfying them. A start is to say that $R(x, y)$ means $x < y$, but on what domain?)

5 Formal Reasoning in First-Order Logic

This section reviews some syntactic notations: free variables versus bound variables and substitution. It lists some of the main equivalences for quantifiers. Finally it describes and illustrates the quantifier rules of the sequent calculus.

5.1 Free vs bound variables

The notion of bound variable occurs widely in mathematics: consider the role of x in $\int f(x)dx$ and the role of k in $\lim_{k=0}^{\infty} a_k$. Similar concepts occur in the λ-calculus. In first-order logic, variables are bound by quantifiers (rather than by λ).

Definition 9 An occurrence of a variable x in a formula is *bound* if it is contained within a subformula of the form $\forall x\, A$ or $\exists x\, A$.

An occurrence of the form $\forall x$ or $\exists x$ is called the *binding occurrence* of x.

An occurrence of a variable is *free* if it is not bound.

A *closed* formula is one that contains no free variables.

A *ground* term, formula, etc. is one that contains no variables at all.

In $\forall x\, \exists y\, R(x, y, z)$, the variables x and y are bound while z is free.

In $(\exists x\, P(x)) \land Q(x)$, the occurrence of x just after P is bound, while that just after Q is free. Thus x has both free and bound occurrences. Such situations can be avoided by renaming bound variables, for example obtaining $(\exists y\, P(y)) \land Q(x)$. Renaming can also ensure that all bound variables in a formula are distinct. The renaming of bound variables is sometimes called *α-conversion*.

[4] Models of these axioms are called *monoids*.

Example 7 Renaming bound variables in a formula preserves its meaning, if done properly. Consider the following renamings of $\forall x\, \exists y\, R(x, y, z)$:

$\forall u\, \exists y\, R(u, y, z)$	OK
$\forall x\, \exists w\, R(x, w, z)$	OK
$\forall u\, \exists y\, R(x, y, z)$	not done consistently
$\forall y\, \exists y\, R(y, y, z)$	clash with bound variable y
$\forall z\, \exists y\, R(z, y, z)$	clash with free variable z

5.2 Substitution

If A is a formula, t is a term, and x is a variable, then $A[t/x]$ is the formula obtained by substituting t for x throughout A. The substitution only affects the *free* occurrences of x. Pronounce $A[t/x]$ as "A with t for x". We also use $u[t/x]$ for substitution in a term u and $C[t/x]$ for substitution in a clause C (clauses are described in Sect. 6 below).

Substitution is only sensible provided all bound variables in A are distinct from all variables in t. This can be achieved by renaming the bound variables in A. For example, if $\forall x\, A$ then $A[t/x]$ is true for all t; the formula holds when we drop the $\forall x$ and replace x by any term. But $\forall x\, \exists y\, x = y$ is true in all models, while $\exists y\, y + 1 = y$ is not. We may not replace x by $y + 1$, since the free occurrence of y in $y + 1$ gets captured by the $\exists y$. First we must rename the bound y, getting say $\forall x\, \exists z\, x = z$; now we may replace x by $y + 1$, getting $\exists z\, y + 1 = z$. This formula is true in all models, regardless of the meaning of the symbols + and 1.

5.3 Equivalences involving quantifiers

These equivalences are useful for transforming and simplifying quantified formulas. They can be to convert formulas into *prenex normal form*, where all quantifiers are at the front, or conversely to move quantifiers into the smallest possible scope.

pulling quantifiers through negation (infinitary de Morgan laws)

$$\neg(\forall x\, A) \simeq \exists x\, \neg A$$
$$\neg(\exists x\, A) \simeq \forall x\, \neg A$$

pulling quantifiers through conjunction and disjunction (provided x is not free in B)

$$(\forall x\, A) \land B \simeq \forall x\, (A \land B)$$
$$(\forall x\, A) \lor B \simeq \forall x\, (A \lor B)$$
$$(\exists x\, A) \land B \simeq \exists x\, (A \land B)$$
$$(\exists x\, A) \lor B \simeq \exists x\, (A \lor B)$$

distributive laws

$$(\forall x\, A) \land (\forall x\, B) \simeq \forall x\, (A \land B)$$
$$(\exists x\, A) \lor (\exists x\, B) \simeq \exists x\, (A \lor B)$$

implication: $A \to B$ as $\neg A \lor B$ (provided x is not free in B)

$$(\forall x\, A) \to B \simeq \exists x\, (A \to B)$$
$$(\exists x\, A) \to B \simeq \forall x\, (A \to B)$$

expansion: $\forall$ and $\exists$ as infinitary conjunction and disjunction

$$\forall x\, A \simeq (\forall x\, A) \land A[t/x]$$
$$\exists x\, A \simeq (\exists x\, A) \lor A[t/x]$$

With the help of the associative and commutative laws for $\land$ and $\lor$, a quantifier can be pulled out of any conjunct or disjunct.

The distributive laws differ from pulling: they replace two quantifiers by one. (Note that the quantified variables will probably have different names, so one of them will have be renamed.) Depending upon the situation, using distributive laws can be either better or worse than pulling. There are no distributive laws for $\forall$ over $\lor$ and $\exists$ over $\land$. If in doubt, do not use distributive laws!

Two substitution laws do not involve quantifiers explicitly, but let us use $x = t$ to replace x by t in a restricted context:

$$(x = t \land A) \simeq (x = t \land A[t/x])$$
$$(x = t \to A) \simeq (x = t \to A[t/x])$$

Many first-order formulas have easy proofs using equivalences:

$$\begin{aligned} \exists x\, (x = a \land P(x)) &\simeq \exists x\, (x = a \land P(a)) \\ &\simeq \exists x\, (x = a) \land P(a) \\ &\simeq P(a) \end{aligned}$$

The following formula is quite hard to prove using the sequent calculus, but using equivalences it is simple:

$$\begin{aligned} \exists z\, (P(z) \to P(a) \land P(b)) &\simeq \forall z\, P(z) \to P(a) \land P(b) \\ &\simeq \forall z\, P(z) \land P(a) \land P(b) \to P(a) \land P(b) \\ &\simeq \mathbf{t} \end{aligned}$$

If you are asked to prove a formula, but no particular formal system (such as the sequent calculus) has been specified, then you may use any convincing argument. Using equivalences as above can shorten the proof considerably. Also, take advantage of symmetries; in proving $A \land B \simeq B \land A$, it obviously suffices to prove $A \land B \models B \land A$.

5.4 Sequent rules for the universal quantifier

Here are the sequent rules for $\forall$:

$$\frac{A[t/x], \Gamma \Rightarrow \Delta}{\forall x\, A, \Gamma \Rightarrow \Delta}\,(\forall l) \qquad \frac{\Gamma \Rightarrow \Delta, A}{\Gamma \Rightarrow \Delta, \forall x\, A}\,(\forall r)$$

Rule $(\forall r)$ holds *provided* x is not free in the conclusion! Note that if x were indeed free somewhere in Γ or Δ, then the sequent would be assuming properties of x. This restriction ensures that x is a fresh variable, which therefore can denote an arbitrary value. Contrast the proof of the theorem $\forall x\, [P(x) \to P(x)]$ with an attempted proof of the invalid formula $P(x) \to \forall x\, P(x)$. Since x is a bound variable, you may rename it to get around the restriction, and obviously $P(x) \to \forall y\, P(y)$ should have no proof.

Rule $(\forall l)$ lets us create many instances of $\forall x\, A$. The exercises below include some examples that require more than one copy of the quantified formula. Since we regard sequents as consisting of sets, we may regard them as containing unlimited quantities of each of their elements. But except for the two rules $(\forall l)$ and $(\exists r)$ (see below), we only need one copy of each formula.

Example 8 In this elementary proof, rule $(\forall l)$ is applied to instantiate the bound variable x with the term $f(y)$. The application of $(\forall r)$ is permitted because y is not free in the conclusion (which, in fact, is closed).

$$\dfrac{\dfrac{\dfrac{}{P(f(y)) \Rightarrow P(f(y))}}{\forall x\, P(x) \Rightarrow P(f(y))}\,(\forall l)}{\forall x\, P(x) \Rightarrow \forall y\, P(f(y))}\,(\forall r)$$

Example 9 This proof concerns part of the law for pulling universal quantifiers out of conjunctions. Rule $(\forall l)$ just discards the quantifier, since it instantiates the bound variable x with the free variable x.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P(x), Q \Rightarrow P(x)}}{P(x) \land Q \Rightarrow P(x)}\,(\land l)}{\forall x\, (P(x) \land Q) \Rightarrow P(x)}\,(\forall l)}{\forall x\, (P(x) \land Q) \Rightarrow \forall x\, P(x)}\,(\forall r)$$

Example 10 The sequent $\forall x\, (A \to B) \Rightarrow A \to \forall x\, B$ is valid provided x is not free in A. That condition is required for the application of $(\forall r)$ below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow A, B} \quad \dfrac{}{A, B \Rightarrow B}}{A,\ A \to B \Rightarrow B}\,(\to l)}{A,\ \forall x\, (A \to B) \Rightarrow B}\,(\forall l)}{A,\ \forall x\, (A \to B) \Rightarrow \forall x\, B}\,(\forall r)}{\forall x\, (A \to B) \Rightarrow A \to \forall x\, B}\,(\to r)$$

What if the condition fails to hold? Let A and B both be the formula $x = 0$. Then $\forall x\, (x = 0 \to x = 0)$ is valid, but $x = 0 \to \forall x\, (x = 0)$ is not valid (it fails under any valuation that sets x to 0).

Note. The proof on the slides of

$$\forall x\, (P \to Q(x)) \Rightarrow P \to \forall y\, Q(y)$$

is essentially the same as the proof above, but uses different variable names so that you can see how a quantified formula like $\forall x\, (P \to Q(x))$ is instantiated to produce $P \to Q(y)$. The proof given above is also correct: the variable names are identical, the instantiation is trivial and $\forall x\, (A \to B)$ simply produces $A \to B$. In our example, B may be any formula possibly containing the variable x; the proof on the slides uses the specific formula $Q(x)$.

5.5 Sequent rules for existential quantifiers

Here are the sequent rules for $\exists$:

$$\dfrac{A, \Gamma \Rightarrow \Delta}{\exists x\, A, \Gamma \Rightarrow \Delta}\,(\exists l) \qquad \dfrac{\Gamma \Rightarrow \Delta, A[t/x]}{\Gamma \Rightarrow \Delta, \exists x\, A}\,(\exists r)$$

Rule $(\exists l)$ holds *provided* x is not free in the conclusion—that is, not free in the formulas of Γ or Δ. These rules are strictly dual to the $\forall$-rules; any example involving $\forall$ can easily be transformed into one involving $\exists$ and having a proof of precisely the same form. For example, the sequent $\forall x\, P(x) \Rightarrow \forall y\, P(f(y))$ can be transformed into $\exists y\, P(f(y)) \Rightarrow \exists x\, P(x)$.

If you have a choice, apply rules that have provisos — namely $(\exists l)$ and $(\forall r)$ — before applying the other quantifier rules as you work upwards. The other rules introduce terms and therefore new variables to the sequent, which could prevent you from applying $(\exists l)$ and $(\forall r)$ later.

Example 11 Figure 2 presents half of the $\exists$ distributive law. Rule $(\exists r)$ just discards the quantifier, instantiating the bound variable x with the free variable x. In the general case, it can instantiate the bound variable with any term.

The restriction on the sequent rules, namely "x is not free in the conclusion", can be confusing when you are building a sequent proof working backwards. One simple way to avoid problems is always to rename a quantified variable if the same variable appears free in the sequent. For example, when you see the sequent $P(x), \exists x\, Q(x) \Rightarrow \Delta$, replace it immediately by $P(x), \exists y\, Q(y) \Rightarrow \Delta$.

Example 12 The sequent

$$\exists x\, P(x) \land \exists x\, Q(x) \Rightarrow \exists x\, (P(x) \land Q(x))$$

is not valid: the value of x that makes $P(x)$ true could differ from the value of x that makes $Q(x)$ true. This comes out clearly in the proof attempt, where we are not allowed to apply $(\exists l)$ twice with the same variable name, x. As soon as we are forced to rename the second variable to y, it becomes obvious that the two values could differ. Turning to the right side of the sequent, no application of $(\exists r)$ can lead to a proof. We have nothing to instantiate x with:

$$\dfrac{\dfrac{\dfrac{\dfrac{P(x), Q(y) \Rightarrow P(x) \land Q(x)}{P(x), Q(y) \Rightarrow \exists x\, (P(x) \land Q(x))}\,(\exists r)}{P(x), \exists x\, Q(x) \Rightarrow \exists x\, (P(x) \land Q(x))}\,(\exists l)}{\exists x\, P(x), \exists x\, Q(x) \Rightarrow \exists x\, (P(x) \land Q(x))}\,(\exists l)}{\exists x\, P(x) \land \exists x\, Q(x) \Rightarrow \exists x\, (P(x) \land Q(x))}\,(\land l)$$

Note On the course website, there is a simple theorem prover called `folderol.ML`. It can prove easy first-order theorems using the sequent calculus, and outputs a summary of each proof. The file includes basic instructions describing how to run it. See `testsuite.ML` contains further instructions and numerous examples.

Exercise 14 Verify the following equivalences by appealing to the truth definition for first-order logic:

$$\neg(\exists x\, P(x)) \simeq \forall x\, \neg P(x)$$
$$(\forall x\, P(x)) \land R \simeq \forall x\, (P(x) \land R)$$
$$(\exists x\, P(x)) \lor (\exists x\, Q(x)) \simeq \exists x\, (P(x) \lor Q(x))$$

Exercise 15 Explain why the following are not equivalences. Are they implications? In which direction?

$$(\forall x\, A) \lor (\forall x\, B) \not\simeq \forall x\, (A \lor B)$$
$$(\exists x\, A) \land (\exists x\, B) \not\simeq \exists x\, (A \land B)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{P(x) \Rightarrow P(x), Q(x)}}{P(x) \Rightarrow P(x) \lor Q(x)}\,(\lor r)}{P(x) \Rightarrow \exists x\, (P(x) \lor Q(x))}\,(\exists r)}{\exists x\, P(x) \Rightarrow \exists x\, (P(x) \lor Q(x))}\,(\exists l) \quad \dfrac{\text{similar}}{\exists x\, Q(x) \Rightarrow \exists x\, (P(x) \lor Q(x))}\,(\exists l)}{\exists x\, P(x) \lor \exists x\, Q(x) \Rightarrow \exists x\, (P(x) \lor Q(x))}\,(\lor l)$$

Figure 2: Half of the $\exists$ distributive law

Exercise 16 Prove $\neg\forall y\, [(Q(a) \lor Q(b)) \land \neg Q(y)]$ using equivalences, and then formally using the sequent calculus.

Exercise 17 Prove the following sequents. Note that the last one requires *two* uses of the $(\forall l)$ rule!

$$(\forall x\, P(x)) \land (\forall x\, Q(x)) \Rightarrow \forall y\, (P(y) \land Q(y))$$
$$\forall x\, (P(x) \land Q(x)) \Rightarrow (\forall y\, P(y)) \land (\forall y\, Q(y))$$
$$\forall x\, [P(x) \to P(f(x))],\ P(a) \Rightarrow P(f(f(a)))$$

Exercise 18 Prove $\forall x\, [P(x) \lor P(a)] \simeq P(a)$.

Exercise 19 Prove the following using the sequent calculus. The last one is difficult and requires two uses of $(\exists r)$.

$$P(a) \lor \exists x\, P(f(x)) \Rightarrow \exists y\, P(y)$$
$$\exists x\, (P(x) \lor Q(x)) \Rightarrow (\exists y\, P(y)) \lor (\exists y\, Q(y))$$
$$\Rightarrow \exists z\, (P(z) \to P(a) \land P(b))$$

6 Clause Methods for Propositional Logic

This section discusses two proof methods in the context of propositional logic.

The *Davis-Putnam-Logeman-Loveland* procedure dates from 1960, and its application to first-order logic has been regarded as obsolete for decades. However, the procedure has been rediscovered and high-performance implementations built. In the 1990s, these "SAT solvers" were applied to obscure problems in combinatorial mathematics, such as the existence of Latin squares. Recently, they have been generalised to SMT solvers, also handling arithmetic, with an explosion of serious applications.

Resolution is a powerful proof method for first-order logic. We first consider *ground* resolution, which works for propositional logic. Though of little practical use, ground resolution introduces some of the main concepts. The resolution method is not natural for hand proofs, but it is easy to automate: it has only one inference rule and no axioms. (For first-order logic, resolution must be augmented with a second rule: factoring.)

Both methods require the original formula to be negated, then converted into CNF. Recall that CNF is a conjunction of disjunction of literals. A disjunction of literals is called a *clause*, and written as a set of literals. Converting the negated formula to CNF yields a set of such clauses. Both methods seek a contradiction in the set of clauses; if the clauses are unsatisfiable, then so is the negated formula, and therefore the original formula is valid.

To *refute* a set of clauses is to prove that they are unsatisfiable. The proof is called a *refutation*.

6.1 Clausal notation

Definition 10 A *clause* is a disjunction of literals

$$\neg K_1 \lor \cdots \lor \neg K_m \lor L_1 \lor \cdots \lor L_n,$$

written as a set

$$\{\neg K_1, \ldots, \neg K_m, L_1, \ldots, L_n\}.$$

A clause is true (in some interpretation) just when one of the literals is true. Thus the empty clause, namely $\{\}$, indicates contradiction. It is normally written $\square$.

Since $\lor$ is commutative, associative, and idempotent, the order of literals in a clause does not matter. The above clause is logically equivalent to the implication

$$(K_1 \land \cdots \land K_m) \to (L_1 \lor \cdots \lor L_n)$$

Kowalski notation abbreviates this to

$$K_1, \cdots, K_m \to L_1, \cdots, L_n$$

and when $n = 1$ we have the familiar Prolog clauses, also known as *definite* or *Horn clauses.*

6.2 The DPLL Method

The Davis-Putnam-Logeman-Loveland (DPLL) method is based upon some obvious identities:

$$\begin{aligned} \mathbf{t} \land A &\simeq A \\ A \land (A \lor B) &\simeq A \\ A \land (\neg A \lor B) &\simeq A \land B \\ A &\simeq (A \land B) \lor (A \land \neg B) \end{aligned}$$

Here is an outline of the algorithm:

1. Delete *tautological* clauses: $\{P, \neg P, \ldots\}$
2. *Unit propagation*: for each unit clause $\{L\}$,
   - delete all clauses containing L
   - delete $\neg L$ from all clauses
3. Delete all clauses containing *pure literals*. A literal L is *pure* if there is no clause containing $\neg L$.

4. If the empty clause is generated, then we have a refutation. Conversely, if all clauses are deleted, then the original clause set is satisfiable.

5. Perform a *case split* on some literal L, and recursively apply the algorithm to the L and $\neg L$ subcases. The clause set is satisfiable if and only if one of the subcases is satisfiable.

This is a *decision procedure*. It must terminate because each case split eliminates a propositional symbol. Modern implementations such as zChaff and MiniSat add various heuristics. They also rely on carefully designed data structures that improve performance by reducing the number of cache misses, for example.

Historical note. Davis and Putnam introduced the first version of this procedure. Logeman and Loveland introduced the splitting rule, and their version has completely superseded the original Davis-Putnam method. Both methods fell into disuse after the invention of resolution and unification, because they cope well with quantifiers. The DPLL algorithm described here works for *propositional logic only*!

Tautological clauses are deleted because they are always true, and thus cannot participate in a contradiction. A pure literal can always be assumed to be true; deleting the clauses containing it can be regarded as a degenerate case split, in which there is only one case.

Example 13 DPLL can show that a formula is not a theorem. Consider the formula $P \lor Q \to Q \lor R$. After negating this and converting to CNF, we obtain the three clauses $\{P, Q\}$, $\{\neg Q\}$ and $\{\neg R\}$. The DPLL method terminates rapidly:

$\{P, Q\}$	$\{\neg Q\}$	$\{\neg R\}$	initial clauses
$\{P\}$		$\{\neg R\}$	unit $\neg Q$
		$\{\neg R\}$	unit P (also pure)
			unit $\neg R$ (also pure)

All clauses have been deleted, so execution terminates. The clauses are satisfiable by $P \mapsto 1$, $Q \mapsto 0$, $R \mapsto 0$. This interpretation falsifies $P \lor Q \to Q \lor R$.

Example 14 Here is an example of a case split. Consider the clause set

$$\begin{array}{ccc} \{\neg Q, R\} & \{\neg R, P\} & \{\neg R, Q\} \\ \{\neg P, Q, R\} & \{P, Q\} & \{\neg P, \neg Q\}. \end{array}$$

There are no unit clauses or pure literals, so we arbitrarily select P for case splitting:

$\{\neg Q, R\}$	$\{\neg R, Q\}$	$\{Q, R\}$	$\{\neg Q\}$	if P is true
	$\{\neg R\}$	$\{R\}$		unit $\neg Q$
	$\{\}$			unit R
$\{\neg Q, R\}$	$\{\neg R\}$	$\{\neg R, Q\}$	$\{Q\}$	if P is false
$\{\neg Q\}$			$\{Q\}$	unit $\neg R$
			$\{\}$	unit $\neg Q$

The empty clause is written $\{\}$ above to make the pattern of execution clearer; traditionally, however, the empty clause is written $\square$. When we encounter a contradiction, we abandon the current case and consider any remaining cases. If all cases are contradictory, then the original set of clauses is unsatisfiable. If they arose from some negated formula $\neg A$, then A is a theorem.

You might find it instructive to download MiniSat[5], which is a very concise open-source SAT solver. It is coded in C++. These days, SAT solvers are largely superseded by SMT solvers, which also handle arithmetic, rays, bit vectors, arrays, bit vectors, etc.

6.3 Introduction to resolution

Resolution combines two clauses containing *complementary literals*. It is essentially the following rule of inference:

$$\frac{B \lor A \quad \neg B \lor C}{A \lor C}$$

To convince yourself that this rule is sound, note that B must be either false or true.

- if B is false, then $B \lor A$ is equivalent to A, so we get $A \lor C$
- if B is true, then $\neg B \lor C$ is equivalent to C, so we get $A \lor C$

You might also understand this rule via transitivity of $\to$

$$\frac{\neg A \to B \quad B \to C}{\neg A \to C}$$

A special case of resolution is when A and C are empty:

$$\frac{B \quad \neg B}{\mathbf{f}}$$

This detects contradictions.

Resolution works with disjunctions. The aim is to prove a contradiction, refuting a formula. Here is the method for proving a formula A:

1. Translate $\neg A$ into CNF as $A_1 \land \cdots \land A_m$.

2. Break this into a set of clauses: $A_1, \ldots, A_m$.

3. Repeatedly apply the resolution rule to the clauses, producing new clauses. These are all consequences of $\neg A$.

4. If a contradiction is reached, we have refuted $\neg A$.

In set notation the resolution rule is

$$\frac{\{B, A_1, \ldots, A_m\} \quad \{\neg B, C_1, \ldots, C_n\}}{\{A_1, \ldots, A_m, C_1, \ldots, C_n\}}$$

Resolution takes two clauses and creates a new one. A collection of clauses is maintained; the two clauses are chosen from the collection according to some strategy, and the new clause is added to it. If $m = 0$ or $n = 0$ then the new clause will be smaller than one of the parent clauses; if $m = n = 0$ then the new clause will be empty. If the empty clause is generated, resolution terminates successfully: we have found a contradiction!

[5] `http://minisat.se/`

6.4 Examples of ground resolution

Let us try to prove

$$P \land Q \to Q \land P$$

Convert its negation to CNF:

$$\neg(P \land Q \to Q \land P)$$

We can combine steps 1 (eliminate →) and 2 (push negations in) using the law $\neg(A \to B) \simeq A \land \neg B$:

$$(P \land Q) \land \neg(Q \land P)$$
$$(P \land Q) \land (\neg Q \lor \neg P)$$

Step 3, push disjunctions in, has nothing to do. The clauses are

$$\{P\} \qquad \{Q\} \qquad \{\neg Q, \neg P\}$$

We resolve $\{P\}$ and $\{\neg Q, \neg P\}$ as follows:

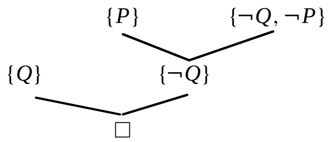
$$\frac{\{P\} \quad \{\neg P, \neg Q\}}{\{\neg Q\}}$$

The resolvent is $\{\neg Q\}$. Resolving $\{Q\}$ with this new clause gives

$$\frac{\{Q\} \quad \{\neg Q\}}{\{\}}$$

The resolvent is the empty clause, properly written as □. We have proved $P \land Q \to Q \land P$ by assuming its negation and deriving a contradiction.

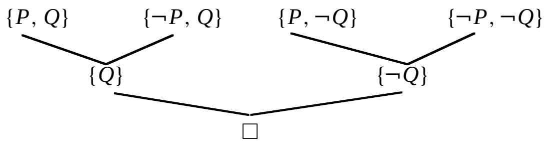
It is nicer to draw a tree like this:



Another example is $(P \leftrightarrow Q) \leftrightarrow (Q \leftrightarrow P)$. The steps of the conversion to clauses is left as an exercise; remember to negate the formula first! The final clauses are

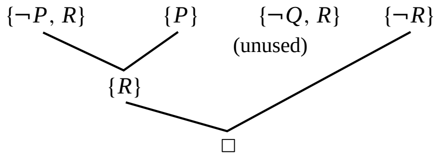
$$\{P, Q\} \qquad \{\neg P, Q\} \qquad \{P, \neg Q\} \qquad \{\neg P, \neg Q\}$$

A tree for the resolution proof is



Note that the tree contains $\{Q\}$ and $\{\neg Q\}$ rather than $\{Q, Q\}$ and $\{\neg Q, \neg Q\}$. If we forget to suppress repeated literals, we can get stuck. Resolving $\{Q, Q\}$ and $\{\neg Q, \neg Q\}$ (keeping repetitions) gives $\{Q, \neg Q\}$, a tautology. Tautologies are useless. Resolving this one with the other clauses leads nowhere. Try it.

These examples could mislead. Must a proof use each clause exactly once? No! A clause may be used repeatedly, and many problems contain redundant clauses. Here is an example:



Redundant clauses can make the theorem-prover flounder; this is a challenge facing the field.

6.5 A proof using a set of assumptions

In this example we assume

$$H \to M \lor N \qquad M \to K \land P \qquad N \to L \land P$$

and prove $H \to P$. It turns out that we can generate clauses separately from the assumptions (taken *positively*) and the conclusion (*negated*).

If we call the assumptions $A_1, \ldots, A_k$ and the conclusion B, then the desired theorem is

$$(A_1 \land \cdots \land A_k) \to B$$

Try negating this and converting to CNF. Using the law $\neg(A \to B) \simeq A \land \neg B$, the negation converts in one step to

$$A_1 \land \cdots \land A_k \land \neg B$$

Since the entire formula is a conjunction, we can separately convert $A_1, \ldots, A_k$, and $\neg B$ to clause form and pool the clauses together.

Assumption $H \to M \lor N$ is essentially in clause form already:

$$\{\neg H, M, N\}$$

Assumption $M \to K \land P$ becomes two clauses:

$$\{\neg M, K\} \qquad \{\neg M, P\}$$

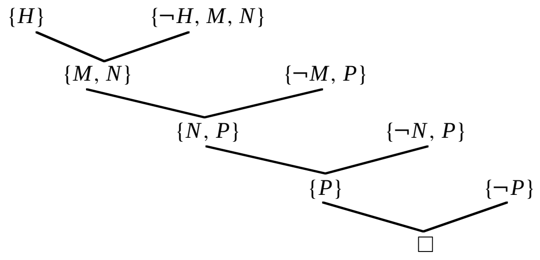
Assumption $N \to L \land P$ also becomes two clauses:

$$\{\neg N, L\} \qquad \{\neg N, P\}$$

The negated conclusion, $\neg(H \to P)$, becomes two clauses:

$$\{H\} \qquad \{\neg P\}$$

A tree for the resolution proof is



The clauses were not tried at random. Here are some points of proof strategy.

Ignoring irrelevance. Clauses $\{\neg M, K\}$ and $\{\neg N, L\}$ lead nowhere, so they were not tried. Resolving with one of these would make a clause containing K or L. There is no way of getting rid of either literal, for no clause contains $\neg K$ or $\neg L$. So this is not a way to obtain the empty clause. (K and L are pure literals.)

Working from the goal. In each resolution step, at least one clause involves the negated conclusion (possibly via earlier resolution steps). We do not attempt to find a contradiction in the assumptions alone, provided (as is often the case) we know them to be satisfiable: any contradiction must involve the negated conclusion. This strategy is called *set of support*. Although largely obsolete, it's very useful when working problems by hand.

Linear resolution. The proof has a linear structure: each resolvent becomes the parent clause for the next resolution step. Furthermore, the other parent clause is always one of the original set of clauses. This simple structure is very efficient because only the last resolvent needs to be saved. It is similar to the execution strategy of Prolog.

Exercise 20 Apply the DPLL procedure to the clause set

$$\{P, Q\} \qquad \{\neg P, Q\} \qquad \{P, \neg Q\} \qquad \{\neg P, \neg Q\}.$$

Exercise 21 Use resolution to prove

$$(A \to B \lor C) \to [(A \to B) \lor (A \to C)].$$

Exercise 22 Explain in more detail the conversion into clauses for the example of §6.5.

Exercise 23 Prove Peirce's law, $((P \to Q) \to P) \to P$, using resolution.

Exercise 24 Use resolution (showing the steps of converting the formula into clauses) to prove these two formulas:

$$(Q \to R) \land (R \to P \land Q) \land (P \to Q \lor R) \to (P \leftrightarrow Q)$$
$$(P \land Q \to R) \land (P \lor Q \lor R) \to ((P \leftrightarrow Q) \to R)$$

Exercise 25 Prove that $(P \land Q) \to (R \land S)$ follows from $P \to R$ and $R \land P \to S$ using linear resolution.

Exercise 26 Convert these axioms to clauses, showing all steps. Then prove Winterstorm → Miserable by resolution:

$$\text{Rain} \land (\text{Windy} \lor \neg\text{Umbrella}) \to \text{Wet}$$
$$\text{Winterstorm} \to \text{Storm} \land \text{Cold}$$
$$\text{Wet} \land \text{Cold} \to \text{Miserable}$$
$$\text{Storm} \to \text{Rain} \land \text{Windy}$$

7 Skolem Functions, Herbrand's Theorem and Unification

Propositional logic is the basis of many proof methods for first-order logic. Eliminating the quantifiers from a first-order formula reduces it "almost" to propositional logic. This section describes how to do so.

7.1 Removing quantifiers: Skolem form

Skolemisation replaces every existentially bound variable by a Skolem constant or function. This transformation does not preserve the meaning of a formula. However, it does preserve *inconsistency*, which is the critical property: resolution works by detecting contradictions.

How to Skolemize a formula

Take a formula in negation normal form. Starting from the outside, follow the nesting structure of the quantifiers. If the formula contains an existential quantifier, then the series of quantifiers must have the form

$$\forall x_1 \cdots \forall x_2 \cdots \forall x_k \cdots \exists y\, A$$

where A is a formula, $k \geq 0$, and $\exists y$ is the leftmost existential quantifier. Choose a k-place function symbol f not present in A (that is, a *new* function symbol). Delete the $\exists y$ and replace all other occurrences of y by $f(x_1, x_2, \ldots, x_k)$. The result is another formula:

$$\forall x_1 \cdots \forall x_2 \cdots \forall x_k \cdots A[f(x_1, x_2, \ldots, x_k)/y]$$

If some existential quantifier is not enclosed in any universal quantifiers, then the formula contains simply $\exists y\, A$ as a subformula. Then this quantifier is deleted and occurrences of y are replaced by a new constant symbol c. The resulting subformula is $A[c/y]$.

Then repeatedly eliminate all remaining existential quantifiers as above. The new symbols are called *Skolem functions* (or Skolem constants).

After Skolemization, the formula has only universal quantifiers. The next step is to throw the remaining quantifiers away. This step is correct because we are converting to clause form, and a clause implicitly includes universal quantifiers over all of its free variables.

We are almost back to the propositional case, except the formula typically contains terms. We shall have to handle constants, function symbols, and variables.

Prenex normal form—where all quantifiers are moved to the front of the formula—would make things easier to follow. However, increasing the scope of the quantifiers prior to Skolemization makes proofs much more difficult. Pushing quantifiers *in* as far as possible, instead of pulling them out, yields a better set of clauses.

Examples of Skolemization

For simplicity, we start with prenex normal form. The affected expressions are underlined.

Example 15 Start with

$$\underline{\exists x}\, \forall y\, \exists z\, R(\underline{x}, y, z)$$

Eliminate the $\exists x$ using the Skolem constant a:

$$\forall y\, \underline{\exists z}\, R(a, y, \underline{z})$$

Eliminate the $\exists z$ using the 1-place Skolem function f:

$$\forall y\, R(a, y, f(y))$$

Finally, drop the $\forall y$ and convert the remaining formula to a clause:

$$\{R(a, y, f(y))\}$$

Example 16 Start with

$$\underline{\exists u}\,\forall v\,\exists w\,\exists x\,\forall y\,\exists z\,((P(h(\underline{u},v))\lor Q(w))\land R(x,h(y,z)))$$

Eliminate the $\exists u$ using the Skolem constant c:

$$\forall v\,\underline{\exists w}\,\exists x\,\forall y\,\exists z\,((P(h(c,v))\lor Q(\underline{w}))\land R(x,h(y,z)))$$

Eliminate the $\exists w$ using the 1-place Skolem function f:

$$\forall v\,\underline{\exists x}\,\forall y\,\exists z\,((P(h(c,v))\lor Q(f(v)))\land R(\underline{x},h(y,z)))$$

Eliminate the $\exists x$ using the 1-place Skolem function g:

$$\forall v\,\forall y\,\underline{\exists z}\,((P(h(c,v))\lor Q(f(v)))\land R(g(v),h(y,\underline{z})))$$

Eliminate the $\exists z$ using the 2-place Skolem function j (the function h is already used!):

$$\forall v\,\forall y\,((P(h(c,v))\lor Q(f(v)))\land R(g(v),h(y,j(v,y))))$$

Dropping the universal quantifiers yields a set of clauses:

$$\{P(h(c,v)),Q(f(v))\}\qquad\{R(g(v),h(y,j(v,y)))\}$$

Each clause is implicitly enclosed by universal quantifiers over each of its variables. So the occurrences of the variable v in the two clauses are independent of each other.

Let's try this example again, first pushing quantifiers in to the smallest possible scopes:

$$\exists u\,\forall v\,P(h(u,v))\lor\exists w\,Q(w)\land\exists x\,\forall y\,\exists z\,R(x,h(y,z))$$

Now the Skolem functions take fewer arguments.

$$\{P(h(c,v)),Q(d)\}\qquad\{R(e,h(y,f(y)))\}$$

The difference between this set of clauses and the previous set may seem small, but in practice it can be huge.

Correctness of Skolemization

Skolemization does *not* preserve meaning. The version presented above does not even preserve validity! For example,

$$\exists x\,(P(a)\to P(x))$$

is valid. (Because the required x is just a.)

Replacing the $\exists x$ by the Skolem constant b gives

$$P(a)\to P(b)$$

This has a different meaning since it refers to a constant b not previously mentioned. And it is not valid! For example, it is false in the interpretation where $P(x)$ means x equals 0 and a denotes 0 and b denotes 1.

Skolemization preserves *consistency*.

- The formula $\forall x\,\exists y\,A$ is satisfiable iff it holds in some interpretation $\mathcal{I}=(D,I)$
- iff for all $x\in D$ there is some $y\in D$ such that A holds
- iff there is some function on D, say $\hat{f}\in D\to D$, such that for all $x\in D$, if $y=\hat{f}(x)$ then A holds
- iff the formula $\forall x\,A[f(x)/y]$ is satisfiable.

If a formula is satisfiable then so is the Skolemized version. If it is unsatisfiable then so is the Skolemized version. And resolution works by proving that a formula is unsatisfiable.

7.2 Herbrand interpretations

A *Herbrand universe* consists of all terms that can be written using the constant and function symbols in a set of clauses S (or quantifier-free formula). The data processed by a Prolog program S is simply its Herbrand universe. A Herbrand interpretation (or model) of S is based on this universe. A fundamental theorem states that for consistency of S we need only consider Herbrand interpretations.

To define the Herbrand universe for the set of clauses S we start with sets of the constant and function symbols in S, *including Skolem functions*.

Definition 11 Let C be the set of all constants in S. If there are none, let $C=\{a\}$ for some constant symbol a of the first-order language. For $n>0$ let $\mathcal{F}_n$ be the set of all n-place function symbols in S and let $\mathcal{P}_n$ be the set of all n-place predicate symbols in S.

The *Herbrand universe* is the set $H=\bigcup_{i\geq 0}H_i$, where

$$\begin{aligned}H_0&=C\\H_{i+1}&=H_i\cup\{f(t_1,\ldots,t_n)\mid t_1,\ldots,t_n\in H_i\text{ and }f\in\mathcal{F}_n\}\end{aligned}$$

Thus, H consists of all the terms that can be written using only the constants and function symbols present in S. There are no variables: the elements of H are ground terms. Formally, H turns out to satisfy the recursive equation

$$H=\{f(t_1,\ldots,t_n)\mid t_1,\ldots,t_n\in H\text{ and }f\in\mathcal{F}_n\}$$

The definition above ensures that C is non-empty. It follows that H is also non-empty, which is necessary.

The elements of H are ground terms. An interpretation (H,I_H) is a *Herbrand interpretation* provided $I_H[t]=t$ for all ground terms t. The point of this peculiar exercise is that we can give meanings to the symbols of S in a purely syntactic way.

Example 17 Given the two clauses

$$\{P(a)\}\qquad\{Q(g(y,z)),\neg P(f(x))\}$$

Then $C=\{a\}$, $\mathcal{F}_1=\{f\}$, $\mathcal{F}_2=\{g\}$ and

$$\begin{aligned}H=\{&a,f(a),g(a,a),f(f(a)),g(f(a),a),\\&g(a,f(a)),g(f(a),f(a)),\ldots\}\end{aligned}$$

Every Herbrand interpretation I_H defines each n-place predicate symbol P to denote some truth-valued function $I_H[P]\in H^n\to\{1,0\}$. We take

$$I_H[P(t_1,\ldots,t_n)]=1$$

if and only if $P(t_1,\ldots,t_n)$ holds in our desired "real" interpretation $\mathcal{I}$ of the clauses. In other words, any specific interpretation $\mathcal{I}=(D,I)$ over some universe D can be mimicked by an Herbrand interpretation. One can show the following two results:

Lemma 12 *Let S be a set of clauses. If an interpretation satisfies S, then an* Herbrand *interpretation satisfies S.*

Theorem 13 *A set S of clauses is unsatisfiable if and only if no Herbrand interpretation satisfies S.*

Equality behaves strangely in Herbrand interpretations. Given an interpretation $\mathcal{I}$, the denotation of $=$ is the set of all pairs of ground terms (t_1, t_2) such that $t_1 = t_2$ according to $\mathcal{I}$. In a context of the natural numbers, the denotation of $=$ could include pairs like $(1 + 1, 2)$ — the two components need not be the same.

7.3 The Skolem-Gödel-Herbrand Theorem

This theorem tells us that unsatisfiability can always be detected by a *finite* process. It provided the original motivation for research into automated theorem proving.

Definition 14 An *instance* of a clause C is a clause that results by replacing variables in C by terms. A *ground instance* of C is an instance of C that contains no variables.

Since the variables in a clause are taken to be universally quantified, every instance of C is a logical consequence of C.

Theorem 15 (Herbrand) *A set S of clauses is unsatisfiable if and only if there is a finite unsatisfiable set S' of ground instances of clauses of S.*

The theorem is valuable because the new set S' expresses the inconsistency in a finite way. However, it only tells us that S' exists; it does not tell us how to derive S'. So how do we generate useful ground instances of clauses? One answer, outlined below, is *unification.*

Example 18 To demonstrate the Skolem-Gödel-Herbrand theorem, consider proving the formula

$$\forall x P(x) \land \forall y\,[P(y) \to Q(y)] \to Q(a) \land Q(b).$$

If we negate this formula, we trivially obtain the following set S of clauses:

$$\{P(x)\} \quad \{\neg P(y), Q(y)\} \quad \{\neg Q(a), \neg Q(b)\}.$$

This set is unsatisfiable. Here is a finite set of ground instances of clauses in S:

$$\{P(a)\} \quad \{P(b)\} \quad \{\neg P(a), Q(a)\}$$
$$\{\neg P(b), Q(b)\} \quad \{\neg Q(a), \neg Q(b)\}.$$

This set reflects the intuitive proof of the theorem. We obviously have $P(a)$ and $P(b)$; using $\forall y\,[P(y) \to Q(y)]$ with a and b, we obtain $Q(a)$ and $Q(b)$. If we can automate this procedure, then we can generate such proofs automatically.

7.4 Unification

Unification is the operation of finding a common instance of two or more terms. Consider a few examples. The terms $f(x, b)$ and $f(a, y)$ have the common instance $f(a, b)$, replacing x by a and y by b. The terms $f(x, x)$ and $f(a, b)$ have no common instance, assuming that a and b are distinct constants. The terms $f(x, x)$ and $f(y, g(y))$ have no common instance, since there is no way that x can have the form y and $g(y)$ at the same time — unless we admit the infinite term $g(g(g(\cdots)))$.

Only variables may be replaced by other terms. Constants are not affected (they remain constant!). Instances of the term $f(t, u)$ must have the form $f(t', u')$, where t' is an instance of t and u' is an instance of u.

Definition 16 A *substitution* is a finite set of replacements

$$[t_1/x_1, \ldots, t_k/x_k]$$

where $x_1, \ldots, x_k$ are distinct variables such that $t_i \neq x_i$ for all $i = 1, \ldots, k$. We use Greek letters ϕ, θ, σ to stand for substitutions.

A substitution $\theta = [t_1/x_1, \ldots, t_k/x_k]$ defines a function from the variables $\{x_1, \ldots, x_k\}$ to terms. Postfix notation is usual for applying a substitution; thus, for example, $x_i\theta = t_i$. Substitution on terms, literals and clauses is defined recursively in the obvious way:

Example 19 The substitution $\theta = [h(y)/x, b/y]$ says to replace x by $h(y)$ and y by b. The replacements occur simultaneously; it does *not* have the effect of replacing x by $h(b)$. Applying this substitution gives

$$f(x, g(u), y)\theta = f(h(y), g(u), b)$$
$$R(h(x), z)\theta = R(h(h(y)), z)$$
$$\{P(x), \neg Q(y)\}\theta = \{P(h(y)), \neg Q(b)\}$$

Definition 17 If ϕ and θ are substitutions then so is their *composition* $\phi \circ \theta$, which satisfies

$$t(\phi \circ \theta) = (t\phi)\theta \qquad \text{for all terms } t$$

Example 20 Let

$$\phi = [j(x)/u, 0/y] \text{ and } \theta = [h(z)/x, g(3)/y].$$

Then $\phi \circ \theta = [j(h(z))/u, h(z)/x, 0/y]$.

Notice that $y(\phi \circ \theta) = (y\phi)\theta = 0\theta = 0$; the replacement $g(3)/y$ has disappeared.

7.5 Unifiers

Definition 18 A substitution θ is a *unifier* of terms t_1 and t_2 if $t_1\theta = t_2\theta$. More generally, θ is a unifier of terms t_1, t_2, …, t_m if $t_1\theta = t_2\theta = \cdots = t_m\theta$. The term $t_1\theta$ is the common instance.

Two terms can only be unified if they have similar structure apart from variables. The terms $f(x)$ and $h(y, z)$ are clearly non-unifiable since no substitution can do anything about the differing function symbols. It is easy to see that θ unifies $f(t_1, \ldots, t_n)$ and $f(u_1, \ldots, u_n)$ if and only if θ unifies t_i and u_i for all $i = 1, \ldots, n$.

Example 21 The substitution $[3/x, g(3)/y]$ unifies the terms $g(g(x))$ and $g(y)$. The common instance is $g(g(3))$. These terms have many other unifiers, such as these:

unifying substitution	common instance
$[f(u)/x, g(f(u))/y]$	$g(g(f(u)))$
$[z/x, g(z)/y]$	$g(g(z))$
$[g(x)/y]$	$g(g(x))$

Note that $g(g(3))$ and $g(g(f(u)))$ are both instances of $g(g(x))$. Thus $g(g(x))$ is more general than $g(g(3))$ and $g(g(f(u)))$. Certainly $g(g(3))$ seems to be arbitrary — neither of the original terms mentions 3! Also important: $g(g(x))$ is as general as $g(g(z))$, despite the different variable names. Let us formalize these intuitions.

Definition 19 The substitution θ is *more general* than ϕ if $\phi = \theta \circ \sigma$ for some substitution σ.

Example 22 Recall the unifiers of $g(g(x))$ and $g(y)$. The unifier $[g(x)/y]$ is more general than the others listed, for

$$\begin{aligned}
[3/x, g(3)/y] &= [g(x)/y] \circ [3/x] \\
[f(u)/x, g(f(u))/y] &= [g(x)/y] \circ [f(u)/x] \\
[z/x, g(z)/y] &= [g(x)/y] \circ [z/x] \\
[g(x)/y] &= [g(x)/y] \circ []
\end{aligned}$$

The last line above illustrates that every substitution θ is more general than itself because $\theta = \theta \circ []$.

Definition 20 A substitution θ is a *most general unifier* (MGU) of terms $t_1, \ldots, t_m$ if θ unifies $t_1, \ldots, t_m$, *and* θ is more general than every other unifier of $t_1, \ldots, t_m$.

Thus if θ is an MGU of terms t_1 and t_2 and $t_1\phi = t_2\phi$ then $\phi = \theta \circ \sigma$ for some substitution σ. The natural unification algorithm returns a most general unifier of two terms.

7.6 A simple unification algorithm

Unification is often presented as operating on the concrete syntax of terms, scanning along character strings. But terms are really tree structures and are so represented in a computer. Unification should be presented as operating on trees. In fact, we need consider only binary trees, since these can represent n-ary branching trees.

Our trees have three kinds of nodes:

- A *variable* $x, y, \ldots$ — can be modified by substitution
- A *constant* $a, b, \ldots$ — handles function symbols also
- A *pair* (t, u) — where t and u are terms

Unification of two terms considers nine cases, most of which are trivial. It is impossible to unify a constant with a pair; in this case the algorithm fails. When trying to unify two constants a and b, if $a = b$ then the most general unifier is []; if $a \neq b$ then unification is impossible. The interesting cases are variable-anything and pair-pair.

Unification with a variable

When unifying a variable x with a term t, where $x \neq t$, we must perform the *occurs check*. If x does not occur in t then the substitution $[t/x]$ has no effect on t, so it does the job trivially:

$$x[t/x] = t = t[t/x]$$

If x *does* occur in t then no unifier exists, for if $x\theta = t\theta$ then the term $x\theta$ would be a proper subterm of itself.

Example 23 The terms x and $f(x)$ are not unifiable. If $x\theta = u$ then $f(x)\theta = f(u)$. Thus $x\theta = f(x)\theta$ would imply $u = f(u)$. We could introduce the infinite term

$$u = f(f(f(f(f(\cdots)))))$$

as a unifier, but this would require a rigorous definition of the syntax and semantics of infinite terms.

Unification of two pairs

Unifying the pairs (t_1, t_2) with (u_1, u_2) requires two recursive calls of the unification algorithm. If θ_1 unifies t_1 with u_1 and θ_2 unifies $t_2\theta_1$ with $u_2\theta_1$ then $\theta_1 \circ \theta_2$ unifies (t_1, t_2) with (u_1, u_2).

It is possible to prove that this process terminates, and that if θ_1 and θ_2 are *most general* unifiers then so is $\theta_1 \circ \theta_2$. If either recursive call fails then the pairs are not unifiable.

As given above, the algorithm works from left to right, but choosing the reverse direction makes no real difference.

Examples of unification

In most of these examples, the two terms have no variables in common. Most uses of unification (including resolution and Prolog) rename variables in one of the terms to ensure this. Such renaming is *not* part of unification itself.

Example 24 Unify $f(x, b)$ with $f(a, y)$. Steps:

> Unify x and a getting $[a/x]$.
>
> Try to unify $b[a/x]$ and $y[a/x]$. These are b and y, so unification succeeds with $[b/y]$.
>
> **Result** is $[a/x] \circ [b/y]$, which is $[a/x, b/y]$.

Example 25 Unify $f(x, x)$ with $f(a, b)$. Steps:

> Unify x and a getting $[a/x]$.
>
> Try to unify $x[a/x]$ and $b[a/x]$. These are distinct constants, a and b. **Fail.**

Example 26 Unify $f(x, g(y))$ with $f(y, x)$. Steps:

> Unify x and y getting $[y/x]$.
>
> Try to unify $g(y)[y/x]$ and $x[y/x]$. These are $g(y)$ and y, violating the occurs check. **Fail.**
>
> But we can unify $f(x, g(y))$ with $f(y', x')$. The failure was caused by having the same variables in both terms.

Example 27 Unify $f(x, x)$ with $f(y, g(y))$. Steps:

> Unify x and y getting $[y/x]$.
>
> Try to unify $x[y/x]$ and $g(y)[y/x]$. But these are y and $g(y)$, where y occurs in $g(y)$. **Fail.** How can $f(x, x)$ and $f(y, g(y))$ have a common instance when the arguments of f must be identical in the first case and different in the second?

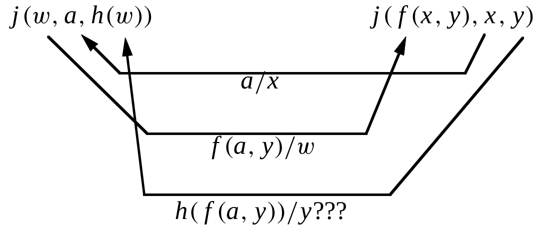
Example 28 Unify $j(w, a, h(w))$ with $j(f(x, y), x, z)$.

> Unify w and $f(x, y)$ getting $[f(x, y)/w]$.
>
> Unify a and x (the substitution has no effect) getting $[a/x]$. Then unify
>
> $h(w)[f(x, y)/w][a/x]$ and $z[f(x, y)/w][a/x]$,
>
> namely $h(f(a, y))$ and z; unifier is
>
> $$[h(f(a, y))/z].$$
>
> **Result** is $[f(x, y)/w] \circ [a/x] \circ [h(f(a, y))/z]$. Performing the compositions, this simplifies to $[f(a, y)/w, a/x, h(f(a, y))/z]$.

Example 29 Unify $j(w, a, h(w))$ with $j(f(x, y), x, y)$. This is the previous example but with a y in place of a z.

> Unify w and $f(x, y)$ getting $[f(x, y)/w]$.
>
> Unify a and x getting $[a/x]$. Then unify
>
> $h(w)[f(x, y)/w][a/x]$ and $y[f(x, y)/w][a/x]$.
>
> These are $h(f(a, y))$ and y, but y occurs in $h(f(a, y))$. **Fail.**

In the diagrams, the lines indicate variable replacements:



Implementation remarks

To unify terms $t_1, t_2, \ldots, t_m$ for $m > 2$, compute a unifier θ of t_1 and t_2, then recursively compute a unifier σ of the terms $t_2\theta, \ldots, t_m\theta$. The overall unifier is then $\theta \circ \sigma$. If any unification fails then the set is not unifiable.

A real implementation does not need to compose substitutions. Most represent variables by pointers and effect the substitution $[t/x]$ by updating pointer x to t. The compositions are cumulative, so this works. However, if unification fails at some point, the pointer assignments must be undone! The algorithm sketched here can take exponential time in unusual cases. Faster algorithms exist, but they are more complex and seldom adopted.

Prolog systems, for the sake of efficiency, omit the occurs check. This can result in circular data structures and looping. It is unsound for theorem proving.

7.7 Examples of theorem proving

These two examples are fundamental. They illustrate how the occurs check enforces correct quantifier reasoning.

Example 30 Consider a proof of

$$(\exists y\, \forall x\, R(x, y)) \to (\forall x\, \exists y\, R(x, y)).$$

For simplicity, produce clauses separately for the antecedent and for the negation of the consequent.

- The antecedent is $\exists y\, \forall x\, R(x, y)$; replacing y by the Skolem constant a yields the clause $\{R(x, a)\}$.
- In $\neg(\forall x\, \exists y\, R(x, y))$, pushing in the negation produces $\exists x\, \forall y\, \neg R(x, y)$. Replacing x by the Skolem constant b yields the clause $\{\neg R(b, y)\}$.

Unifying $R(x, a)$ with $R(b, y)$ detects the contradiction $R(b, a) \land \neg R(b, a)$.

Example 31 In a similar vein, let us try to prove

$$(\forall x\, \exists y\, R(x, y)) \to (\exists y\, \forall x\, R(x, y)).$$

- Here the antecedent is $\forall x\, \exists y\, R(x, y)$; replacing y by the Skolem function f yields the clause $\{R(x, f(x))\}$.
- The negation of the consequent is $\neg(\exists y\, \forall x\, R(x, y))$, which becomes $\forall y\, \exists x\, \neg R(x, y)$. Replacing x by the Skolem function g yields the clause $\{\neg R(g(y), y)\}$.

Observe that $R(x, f(x))$ and $R(g(y), y)$ are not unifiable because of the occurs check. And so it should be, because the original formula is not a theorem! The best way to demonstrate that a formula is not a theorem is to exhibit a counterexample. Here are two:

- The domain is the set of all people who have ever lived. The relation $R(x, y)$ holds if x loves y. The function $f(x)$ denotes the mother of x, and $\{R(x, f(x))\}$ holds because everybody loves their mother. The function $g(x)$ denotes the landlord of x, and $\neg R(g(y), y)$ holds.
- The domain is the set of integers. The relation $R(x, y)$ holds whenever $x = y$. The function f is defined by $f(x) = x$ and $\{R(x, f(x))\}$ holds. The function g is defined by $g(x) = x + 1$ and so $\{\neg R(g(y), y)\}$ holds.

Exercise 27 Skolemize the following formulas, dropping all quantifiers:

$$\exists xy\, \forall z\, \exists w\, P(x, y, z, w)$$
$$\forall u\, (\exists x\, P(x, x) \land \forall v\, \exists y\, Q(u, y))$$
$$\forall u\, \exists xy\, P(x, y)$$

Exercise 28 Consider a first-order language with 0 and 1 as constant symbols, with $-$ as a 1-place function symbol and $+$ as a 2-place function symbol, and with $<$ as a 2-place predicate symbol.

(a) Describe the Herbrand Universe for this language.

(b) The language can be interpreted by taking the integers for the universe and giving 0, 1, $-$, $+$, and $<$ their usual meanings over the integers. What do those symbols denote in the corresponding Herbrand model?

Exercise 29 For each of the following pairs of terms, give a most general unifier or explain why none exists. Do not rename variables prior to performing the unification.

$$\begin{array}{rl} f(g(x), z) & f(y, h(y)) \\ j(x, y, z) & j(f(y, y), f(z, z), f(a, a)) \\ j(x, z, x) & j(y, f(y), z) \\ j(f(x), y, a) & j(y, z, z) \\ j(g(x), a, y) & j(z, x, f(z, z)) \end{array}$$

Exercise 30 Which of the following substitutions are most general unifiers for the terms $f(x, y, z)$ and $f(w, w, v)$?

$$\begin{array}{l} [x/y, x/w, v/z] \\ [y/x, y/w, v/z] \\ [y/x, v/z] \\ [u/x, u/y, u/w, y/z, y/v] \\ [x/y, x/z, x/w, x/v] \end{array}$$

8 First-Order Resolution and Prolog

By means of unification, we can extend resolution to first-order logic. As a special case we obtain Prolog. Other theorem provers are also based on unification. Other applications include polymorphic type checking.

DPLL only works for ground formulas. It can be extended to prove first-order theorems, introducing techniques to replace variables by constant expressions. But while unification and resolution come up with good terms automatically, guessing them heuristically is difficult. We shall exclusively use resolution to solve such problems.

A number of resolution theorem provers can be downloaded for free from the Internet. Some of the main ones include Vampire (`http://www.vprover.org`, SPASS (`http://www.spass-prover.org`) and E (`http://www.eprover.org`). It might be instructive to download one of them and experiment with it.

8.1 Binary resolution

We now define the binary resolution rule with unification:

$$\frac{\{B, A_1, \ldots, A_m\} \quad \{\neg D, C_1, \ldots, C_n\}}{\{A_1, \ldots, A_m, C_1, \ldots, C_n\}\sigma} \quad \text{if } B\sigma = D\sigma$$

As before, the first clause contains B and other literals, while the second clause contains $\neg D$ and other literals. The substitution σ is a unifier of B and D (almost always a *most general* unifier). This substitution is applied to all remaining literals, producing the conclusion.

The variables in one clause are renamed before resolution to prevent clashes with the variables in the other clause. Renaming is sound because the scope of each variable is its clause. Resolution is sound because it takes an instance of each clause — the instances are valid, because the clauses are universally valid — and then applies the propositional resolution rule, which is sound. For example, the two clauses

$$\{P(x)\} \quad \text{and} \quad \{\neg P(g(x))\}$$

yield the empty clause in a single resolution step. This works by renaming variables — say, x to y in the second clause — and unifying $P(x)$ with $P(g(y))$. Forgetting to rename variables is fatal, because $P(x)$ cannot be unified with $P(g(x))$.

8.2 Factoring

Factoring is a separate inference rule, but a vital part of the resolution method. It takes a clause and unifies some literals within it (which must all have the same sign), yielding a new clause. Starting with the clause $\{P(x, b), P(a, y)\}$, factoring derives $\{P(a, b)\}$, since $P(a, b)$ is the result of unifying $P(x, b)$ with $P(a, y)$. This new clause is a unit clause and therefore especially useful. In this case, however, it is logically weaker than the original clause.

Here is the factoring inference rule in its general form:

$$\frac{\{B_1, \ldots, B_k, A_1, \ldots, A_m\}}{\{B_1, A_1, \ldots, A_m\}\sigma} \qquad (\textit{if } B_1\sigma = \cdots = B_k\sigma)$$

Factoring is necessary for completeness. Resolution by itself tends to make clauses longer and longer. Only short clauses are likely to lead to a contradiction. If every clause has at least two literals, then the *only* way to reach the empty clause is with the help of factoring.

The search space is huge. Resolution and factoring can be applied in many different ways at each step of the proof. Modern resolution systems use complex heuristics to manage and limit the search.

Example 32 Prove $\forall x\, \exists y\, \neg(P(y, x) \leftrightarrow \neg P(y, y))$.

Negate and expand the $\leftrightarrow$, getting

$$\begin{aligned} \neg\forall x\, \exists y\, \neg[(\neg P(y, x) \lor \neg P(y, y)) \\ \land (\neg\neg P(y, y) \lor P(y, x))] \end{aligned}$$

Its negation normal form is

$$\exists x\, \forall y\, [(\neg P(y, x) \lor \neg P(y, y)) \land (P(y, y) \lor P(y, x))]$$

Skolemization yields

$$(\neg P(y, a) \lor \neg P(y, y)) \land (P(y, y) \lor P(y, a)).$$

The problem consists of just two clauses:

$$\{\neg P(y, a), \neg P(y, y)\} \qquad \{P(y, y), P(y, a)\}$$

At such a situation, it is a common error to imagine that we can resolve on all the literals at the same time, immediately reaching the empty clause. This is not possible: a resolution step combines *one single pair* of complementary literals. We can, for example, resolve these two clauses on the literal $P(y, a)$. We obtain $\{\neg P(y, y), P(y, y)\}$, which is a tautology and therefore worthless.

Factoring is necessary here. Applying the factoring rule to each of these clauses yields two additional clauses:

$$\{\neg P(a, a)\} \qquad \{P(a, a)\}$$

These are complementary unit clauses, so resolution yields the empty clause. This proof is trivial!

As a general rule, if there are no unit clauses to begin with, then factoring will be necessary. Otherwise, resolution steps will continue to yield clauses that have at least two literals. The only exception is when there are repeated literals, as in the following example.

Example 33 Let us prove $\exists x\,[P \to Q(x)] \land \exists x\,[Q(x) \to P] \to \exists x\,[P \leftrightarrow Q(x)]$. The clauses are

$$\{P, \neg Q(b)\} \quad \{P, Q(x)\} \quad \{\neg P, \neg Q(x)\} \quad \{\neg P, Q(a)\}$$

A short resolution proof follows. The complementary literals are underlined:

$$\begin{array}{llllll}
\text{Resolve} & \{P, \underline{\neg Q(b)}\} & \text{with} & \{P, \underline{Q(x)}\} & \text{getting} & \{P\} \\
\text{Resolve} & \{\neg P, \underline{\neg Q(x)}\} & \text{with} & \{\neg P, \underline{Q(a)}\} & \text{getting} & \{\neg P\} \\
\text{Resolve} & \underline{\{P\}} & \text{with} & \underline{\{\neg P\}} & \text{getting} & \square
\end{array}$$

This proof relies on the set identity $\{P, P\} = \{P\}$. We can view it as a degenerate case of factoring where the literals are identical, with no need for unification.

8.3 Redundant clauses and subsumption

During resolution, the number of clauses builds up dramatically; it is important to delete all redundant clauses.

Each new clause is a consequence of the existing clauses. A contradiction can only be derived if the original set of clauses is unsatisfiable. A clause can be deleted if it does not affect the consistency of the set. A tautology should always be deleted: it is true in all interpretations.

Here is a subtler case. Consider the clauses

$$\{S, R\} \qquad \{P, \neg S\} \qquad \{P, Q, R\}$$

Resolving the first two yields $\{P, R\}$. Since each clause is a disjunction, any interpretation that satisfies $\{P, R\}$ also satisfies $\{P, Q, R\}$. Thus $\{P, Q, R\}$ cannot cause inconsistency, and should be deleted. Put another way, $P \lor R$ implies $P \lor Q \lor R$. Any contradiction derived from $P \lor Q \lor R$ could also be derived from $P \lor R$. This sort of deletion is called *subsumption*; clause $\{P, R\}$ *subsumes* $\{P, Q, R\}$.

Consider factoring the clause $\{R(x), R(a)\}$: the result is $\{R(a)\}$, which is clearly no weaker than what we started with. It is safe to remove the clause $\{R(x), R(a)\}$. However, not every instance of factoring behaves in this way.

Subsumption typically involves unification. In the general case, a clause C subsumes a clause D if $C\theta \subseteq D$ for some substitution θ (if some instance of C implies D). For example, $\{P(x)\}$ subsumes $\{P(a), Q(y)\}$: since x is a variable, $\{P(x)\}$ implies every formula of the form $P(t)$.

8.4 Prolog clauses

Prolog clauses, also called Horn clauses, have at most one positive literal. A *definite* clause is one of the form

$$\{\neg A_1, \ldots, \neg A_m, B\}$$

It is logically equivalent to $(A_1 \land \cdots \land A_m) \to B$. Prolog's notation is

$$B \leftarrow A_1, \ldots, A_m.$$

If $m = 0$ then the clause is simply written as B and is sometimes called a *fact*.

A *negative* or *goal* clause is one of the form

$$\{\neg A_1, \ldots, \neg A_m\}$$

Prolog permits just one of these; it represents the list of unsolved goals. Prolog's notation is

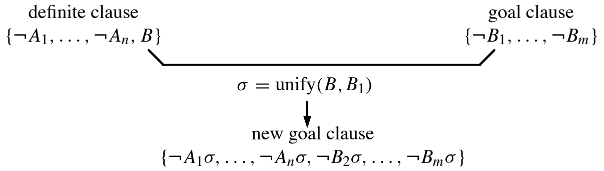
$$\leftarrow A_1, \ldots, A_m.$$

A Prolog database consists of definite clauses. Observe that definite clauses cannot express negative assertions, since they must contain a positive literal. From a mathematical point of view, they have little expressive power; every set of definite clauses is satisfiable! Even so, definite clauses are a natural notation for many problems.

8.5 Prolog computations

A Prolog computation takes a database of definite clauses together with one goal clause. It repeatedly resolves the goal clause with some definite clause to produce a new goal clause. If resolution produces the empty goal clause, then execution succeeds.

Here is a diagram of a Prolog computation step:


definite clause
$\{\neg A_1, \ldots, \neg A_n, B\}$
goal clause
$\{\neg B_1, \ldots, \neg B_m\}$
$\sigma = \text{unify}(B, B_1)$
new goal clause
$\{\neg A_1\sigma, \ldots, \neg A_n\sigma, \neg B_2\sigma, \ldots, \neg B_m\sigma\}$

This is a *linear* resolution (§6). Two program clauses are never resolved with each other. The result of each resolution step becomes the next goal clause; the previous goal clause is discarded after use.

Prolog resolution is efficient, compared with general resolution, because it involves less search and storage. General resolution must consider all possible pairs of clauses; it adds their resolvents to the existing set of clauses; it spends a great deal of effort getting rid of subsumed (redundant) clauses and probably useless clauses. Prolog always resolves some program clause with the goal clause. Because goal clauses do not accumulate, Prolog requires little storage. Prolog never uses factoring and does not even remove repeated literals from a clause.

Prolog has a fixed, deterministic execution strategy. The program is regarded as a list of clauses, not a set; the clauses are tried strictly in order. With a clause, the literals are also regarded as a list. The literals in the goal clause are proved strictly from left to right. The goal clause's first literal is replaced by the literals from the unifying program clause, preserving their order.

Prolog's search strategy is depth-first. To illustrate what this means, suppose that the goal clause is simply $\leftarrow P$ and that the program clauses are $P \leftarrow P$ and $P \leftarrow$. Prolog will resolve $P \leftarrow P$ with $\leftarrow P$ to obtain a new goal clause, which happens to be identical to the original one. Prolog never notices the repeated goal clause, so it repeats the same useless resolution over and over again. Depth-first search means that at every *choice point*, such as between using

$P \leftarrow P$ and $P \leftarrow$, Prolog will explore every avenue arising from its first choice before considering the second choice. Obviously, the second choice would prove the goal trivially, but Prolog never notices this.

8.6 Example of Prolog execution

Here are axioms about the English succession: how y can become King after x.

$$\begin{gathered}\forall x\,\forall y\,(\mathrm{oldestson}(y,x) \land \mathrm{king}(x) \to \mathrm{king}(y))\\ \forall x\,\forall y\,(\mathrm{defeat}(y,x) \land \mathrm{king}(x) \to \mathrm{king}(y))\\ \mathrm{king}(\mathrm{richardIII})\\ \mathrm{defeat}(\mathrm{henryVII}, \mathrm{richardIII})\\ \mathrm{oldestson}(\mathrm{henryVIII}, \mathrm{henryVII})\end{gathered}$$

The goal is to prove king(henryVIII). Now here is the same problem in the form of definite clauses:

$$\begin{gathered}\{\neg\mathrm{oldestson}(y,x), \neg\mathrm{king}(x), \mathrm{king}(y)\}\\ \{\neg\mathrm{defeat}(y,x), \neg\mathrm{king}(x), \mathrm{king}(y)\}\\ \{\mathrm{king}(\mathrm{richardIII})\}\\ \{\mathrm{defeat}(\mathrm{henryVII}, \mathrm{richardIII})\}\\ \{\mathrm{oldestson}(\mathrm{henryVIII}, \mathrm{henryVII})\}\end{gathered}$$

The goal clause is

$$\{\neg\mathrm{king}(\mathrm{henryVIII})\}.$$

Figure 3 shows the execution. The subscripts in the clauses are to rename the variables.

Note how crude this formalization is. It says nothing about the passage of time, about births and deaths, about not having two kings at once. The oldest son of Henry VII, Arthur, died aged 15, leaving the future Henry VIII as the oldest *surviving* son. All formal models must omit some real-world details: reality is overwhelmingly complex.

The Frame Problem in Artificial Intelligence reveals another limitation of logic. Consider writing an axiom system to describe a robot's possible actions. We might include an axiom to state that if the robot lifts an object at time t, then it will be holding the object at time $t + 1$. But we also need to assert that the positions of everything else remain the same as before. Then we must consider the possibility that the object is a table and has other things on top of it. Separation Logic, a variant of Hoare logic, was invented to solve the frame problem, especially for reasoning about linked data structures.

Prolog is a powerful and useful programming language, but it is seldom logic. Most Prolog programs rely on special predicates that affect execution but have no logical meaning. There is a huge gap between the theory and practice of logic programming.

8.7 Prolog and true theorem proving

Prolog-like technologies yield a number of different approaches to automatic theorem proving. In the late 1980s, Stickel [1988] undertook experiments using an alternative to resolution known as *model elimination*, and later he discovered that these techniques could be implemented within Prolog compilers. The point was to exploit the outstanding performance of the era's Prolog implementations to build a faster theorem prover. The key was to translate the set of clauses (obtained in the normal way) into a set of Prolog-like clauses.

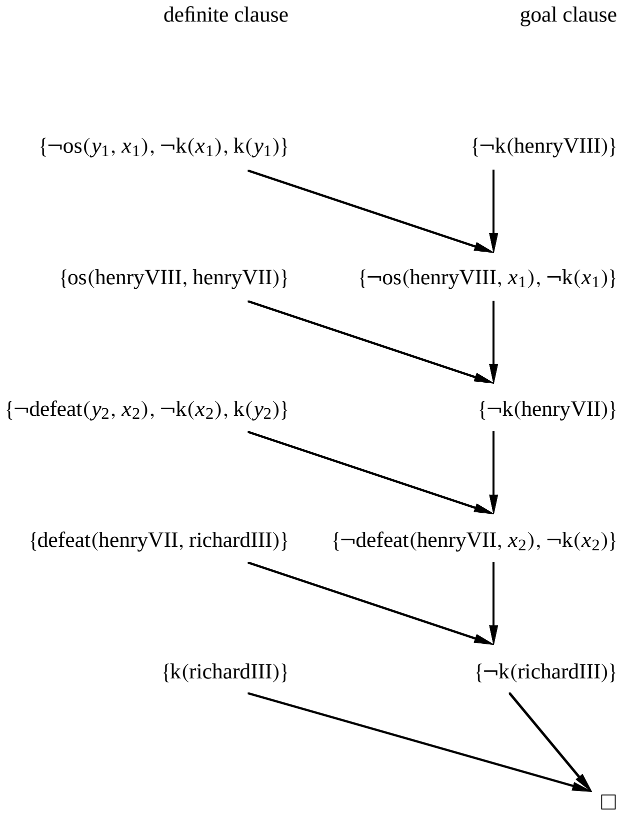

definite clause
goal clause
{¬os(y₁, x₁), ¬k(x₁), k(y₁)}
{¬k(henryVIII)}
{os(henryVIII, henryVII)}
{¬os(henryVIII, x₁), ¬k(x₁)}
{¬defeat(y₂, x₂), ¬k(x₂), k(y₂)}
{¬k(henryVII)}
{defeat(henryVII, richardIII)}
{¬defeat(henryVII, x₂), ¬k(x₂)}
{k(richardIII)}
{¬k(richardIII)}
□

Figure 3: Execution of a Prolog program (os = oldestson, k = king)

Strictly speaking, a Prolog clause has one positive literal (the head of the clause) and zero or more negative literals. Stickel's idea was to replace the clause $\{A_1, \ldots, A_m\}$ by its m *contrapositives*:

$$\begin{aligned} A_1 &\leftarrow \neg A_2, \ldots, \neg A_m\\ A_2 &\leftarrow \neg A_3, \ldots, \neg A_m, \neg A_1\\ &\vdots\\ A_m &\leftarrow \neg A_1, \ldots, \neg A_{m-1}\end{aligned}$$

All we are doing here is giving each of the literals the possibility of being the head, even if it is negative. Doing this for each clause, we get a sort of Prolog program, which now must be executed on a slightly modified Prolog system:

- Unification must be done right (with occurs check).
- Depth-first search must be replaced by depth-first iterative deepening, to ensure completeness.
- The goal $\neg P$ can be regarded as immediately proved if it arises within an attempt to prove P (possibly many layers up). This *extension rule* embodies a form of proof by contradiction: whenever you are trying to prove P, it is legitimate to assume $\neg P$.

Interest in Prolog-based automatic theorem provers declined in the 1990s along with interest in Prolog itself. Today, the conception survives as the *connection calculus*. It retains the idea that every proof step involves one of the original clauses (without however bothering with contrapositives). It replaces Prolog execution by a data structure called a connection graph. The extension rule still applies.

Exercise 31 Convert the following formula into clauses, showing your working. Then present two resolution proofs different from the one shown in Example 33 above.

$$\exists x\, [P \to Q(x)] \land \exists x\, [Q(x) \to P] \to \exists x\, [P \leftrightarrow Q(x)]$$

Exercise 32 Is the clause $\{P(x,b), P(a,y)\}$ logically equivalent to the unit clause $\{P(a,b)\}$? Is the clause $\{P(y,y), P(y,a)\}$ logically equivalent to $\{P(y,a)\}$? Explain both answers.

Exercise 33 Show that every set of definite clauses is satisfiable. (Hint: first consider propositional logic, then extend your argument to first order logic.)

Exercise 34 Convert these formulas into clauses, showing each step: negating the formula, eliminating → and ↔, pushing in negations, Skolemizing, dropping the universal quantifiers, and converting the resulting formula into CNF. Comment on the likely outcome of resolution in each case.

$$\begin{gathered}
(\forall x\, \exists y\, R(x,y)) \to (\exists y\, \forall x\, R(x,y)) \\
(\exists y\, \forall x\, R(x,y)) \to (\forall x\, \exists y\, R(x,y)) \\
\exists x\, \forall yz\, ((P(y) \to Q(z)) \to (P(x) \to Q(x))) \\
\neg\exists y\, \forall x\, (R(x,y) \leftrightarrow \neg\exists z\, (R(x,z) \land R(z,x)))
\end{gathered}$$

Exercise 35 Consider the Prolog program consisting of the definite clauses

$$\begin{aligned}
P(f(x,y)) &\leftarrow Q(x), R(y) \\
Q(g(z)) &\leftarrow R(z) \\
R(a) &\leftarrow
\end{aligned}$$

Describe the Prolog computation starting from the goal clause $\leftarrow P(v)$. Keep track of the substitutions affecting v to determine what answer the Prolog system would return.

Exercise 36 Find a refutation from the following set of clauses using resolution and factoring.

$$\begin{gathered}
\{P(x,b), P(a,y)\} \\
\{\neg P(x,b), \neg P(c,y)\} \\
\{\neg P(x,d), \neg P(a,y)\}
\end{gathered}$$

Exercise 37 Find a refutation from the following set of clauses using resolution and factoring.

$$\begin{gathered}
\{\neg R(x,a), \neg R(x,y), \neg R(y,x)\} \\
\{R(x,f(x)), R(x,a)\} \\
\{R(f(x),x), R(x,a)\}
\end{gathered}$$

Exercise 38 Prove the following formulas by resolution, showing all steps of the conversion into clauses. Remember to negate first!

$$\begin{gathered}
\forall x\, (P \lor Q(x)) \to (P \lor \forall x\, Q(x)) \\
\exists xy\, (R(x,y) \to \forall zw\, R(z,w))
\end{gathered}$$

Note that P is just a predicate symbol, so in particular, x is not free in P.

9 Decision Procedures and SMT Solvers

One of the original objectives of formal logic was to replace argumentation with calculation: to answer mathematical questions mechanically. Unfortunately, this reductionistic approach is unrealistic. Researchers soon found out that most fundamental problems were insoluble, in general. The Halting Problem was found to be undecidable. Gödel proved that no reasonable system of axioms yielded a complete theory for arithmetic. And problem classes that were decidable turned out to be very difficult: propositional satisfiability is NP-complete (and probably exponential), while many other decision procedures are of hyper-exponential complexity, limiting them to the smallest problems.

But as we have seen before, worst-case results can be overly pessimistic. DPLL solves propositional satisfiability for very large formulas. Used in conjunction with other techniques, it yields highly effective program analysis tools that can, in particular, identify program loops that could fail to terminate. Below we shall briefly consider some simple methods for solving systems of arithmetic constraints. These *decision procedures* can be combined with the DPLL method. A modern *Satisfiability Modulo Theories* (SMT) solver brings together a large number of separate, small reasoning subsystems to solve very large and difficult problems. As before, we work with negated problems, which we attempt to show unsatisfiable or else to construct a model.

9.1 Decision procedures and problems

A class of mathematical problems is called *decidable* if there exists an algorithm for determining whether a given problem has a solution or not. Such an algorithm is called a *decision procedure* for that class of problems. For example, it is decidable whether or not a given string is accepted by a given finite state machine.

In theory, we are only interested in yes/no answers, though in practice, many decision procedures return additional information. DPLL answers the yes/no question of whether a set of clauses is satisfiable, but in the "yes" case it also delivers a model, and in the "no" case it can even deliver a proof. A decidable class of mathematical problems is called a *decision problem*.

A number of decidable subcases of first-order logic were identified in the first half of the 20th century. One of the more interesting decision problems is *Presburger arithmetic*: the first-order theory of the natural numbers with addition (and subtraction, and multiplication by constants). There is an algorithm to determine whether a given sentence in Presburger arithmetic is valid or not.

Real numbers behave differently from the natural numbers (where $m < n \iff m + 1 \leq n$) and require their own algorithms. Once again, the only allowed operations are addition, subtraction and constant multiplication. Such a language is called *linear arithmetic*. The validity of linear arithmetic formulas over the reals is also decidable.

Even unrestricted arithmetic (with multiplication) is decidable for the reals. Unfortunately, the algorithms are too complicated and expensive for widespread use. Even Euclidean geometry can be reduced to problems on the reals, and is therefore decidable. Practical decision procedures exist for simple data structures such as arrays and lists.

9.2 Fourier-Motzkin variable elimination

Fourier-Motzkin variable elimination is a classic decision procedure for real (or rational) linear arithmetic. It dates from 1826 and is very inefficient in general, but relatively easy to understand. In the general case, it deals with conjunctions of linear constraints over the reals or rationals:

$$\bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{ij}x_j \leq b_i \tag{6}$$

It works by eliminating variables in succession. Eventually, either a contradiction or a trivial constraint will remain.

The key idea is a technique known as quantifier elimination (QE). We have already seen Skolemization, which removes quantifiers from a formula, but that technique does not preserve the formula's meaning. QE replaces a formula with an equivalent but quantifier-free formula. It is only possible for specific theories, and is generally very expensive.

For the reals, existential quantifiers can be eliminated as follows:

$$\exists x \, (\bigwedge_{i=1}^{m} a_i \leq x \land \bigwedge_{j=1}^{n} x \leq b_j) \iff \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} a_i \leq b_j$$

A system of constraints has many lower bounds, $\{a_i\}_{i=1}^m$, and upper bounds, $\{b_j\}_{j=1}^n$. To eliminate a variable, we need to form every combination of a lower bound with an upper bound. Observe that removing one quantifier replaces a conjunction of $m+n$ inequalities by a conjunction of $m \times n$ inequalities.

Given a problem in the form (6), we first eliminate the variable x_n. We examine each of the constraints, based on the sign of the relevant coefficient, a_{in}, for $i = 1, \ldots, m$. If $a_{in} = 0$ then it does not constrain x_n at all. Otherwise, define $\beta_i = b_i - \sum_{j=1}^{n-1} a_{ij}x_j$. Then

$$\text{If } a_{in} > 0 \text{ then } x_n \leq \frac{\beta_i}{a_{in}}$$

$$\text{If } a_{in} < 0 \text{ then } x_n \geq \frac{\beta_i}{a_{in}}$$

The first case yields an upper bound for x_n while the second case yields a lower bound. The lower bound case can also be written $-x_n \leq -\beta_i/a_{in}$. Now a pair of constraints i and i' involving opposite signs can be added, yielding

$$0 \leq \frac{\beta_i}{a_{in}} - \frac{\beta_{i'}}{a_{i'n}}.$$

Doing this for every pair of constraints eliminates x_n.

Consider the following small set of constraints:

$$x \leq y \qquad x \leq z \qquad -x + y + 2z \leq 0 \qquad -z \leq -1$$

Let's work through the algorithm very informally. The first two constraints give upper bounds for x, while the third constraint gives a lower bound, and can be rewritten as $-x \leq -y - 2z$. Adding it to $x \leq y$ yields $0 \leq -2z$ which can be rewritten as $z \leq 0$. Doing the same thing with $x \leq z$ yields $y + z \leq 0$ which can be rewritten as $z \leq -y$. This leaves us with a new set of constraints, where we have eliminated the variable x:

$$z \leq 0 \qquad z \leq -y \qquad -z \leq -1$$

Now we have two separate upper bounds for z, as well as a lower bound, because we know $z \geq 1$. There are again two possible combinations of a lower bound with an upper bound, and we derive $0 \leq -1$ and $0 \leq -y - 1$. Because $0 \leq -1$ is contradictory, Fourier-Motzkin variable elimination has refuted the original set of constraints.

Many other decision procedures exist, frequently for more restrictive problem domains, aiming for greater efficiency and better integration with other reasoning tools. *Difference arithmetic* is an example: arithmetic constraints are restricted to the form $x - y \leq c$, where c is an integer constant. Satisfiability of a set of difference arithmetic constraints can be determined very quickly by constructing a graph and invoking the Bellman-Ford algorithm to look for a cycle representing the contradiction $0 \leq -1$. In other opposite direction, harder decision problems can handle more advanced applications but require much more computer power.

9.3 Other decidable theories

One of the most dramatic examples of quantifier elimination concerns the domain of *real polynomial arithmetic*:

$$\exists x \, [ax^2 + bx + c = 0] \iff$$
$$b^2 \geq 4ac \land (c = 0 \lor a \neq 0 \lor b^2 > 4ac)$$

The left-hand side asks when a quadratic equation can have solutions, and the right-hand side gives necessary and sufficient conditions, including degenerate cases ($a = c = 0$). But this neat formula is the exception, not the rule. In general, QE yields gigantic formulas. Applying QE to a formula containing no free variables yields a sentence, which simplifies to **t** or **f**, but even then, runtime is typically hyperexponential.

For linear *integer* arithmetic, *every* decision algorithm has a worst-case runtime of at least $2^{2^{cn}}$. People typically use the Omega test or Cooper's algorithm.

There exist decision procedures for arrays, for at least the trivial theory of (where lk is lookup and up is update)

$$lk(up(a, i, v), j) = \begin{cases} v & \text{if } i = j \\ lk(a, j) & \text{if } i \neq j \end{cases}$$

The theory of lists with head, tail, cons is also decidable. Combinations of decidable theories remain decidable under certain circumstances, e.g., the theory of arrays with linear arithmetic subscripts. The seminal publication, still cited today, is Nelson and Oppen [1980].

9.4 Satisfiability modulo theories

Many decision procedures operate on existentially quantified conjunctions of inequalities. An arbitrary formula can be solved by translating it into *dis*junctive normal form (as opposed to the more usual conjunctive normal form) and by eliminating universal quantifiers in favour of negated existential quantifiers. However, these transformations typically cause exponential growth and may need to be repeated as each variable is eliminated.

Satisfiability modulo theories (SMT) is an extension of DPLL to make use of decision procedures, extending their scope while avoiding the problems mentioned above. The idea is that DPLL handles the logical part of the problem while delegating reasoning about particular theories to the relevant decision procedures.

We extend the language of propositional satisfiability to include atomic formulas belonging to our decidable theory (or theories). For the time being, these atomic formulas are not interpreted, so a literal such as $a < 0$ is wholly unrelated to $a > 0$. But the decision procedures are invoked during the execution of DPLL; if we have already asserted $a < 0$, then the attempt to assert $a > 0$ will be rejected by the decision procedure, causing backtracking. Information can be fed back from the decision procedure to DPLL in the form of a new clause, such as $\neg(a < 0) \lor \neg(a > 0)$.

[*Remark*: the Fourier-Motzkin decision procedure eliminates variables, but all decision procedures in actual use can deal with constants such as a as well, and satisfiability-preserving transformations exist between formulas involving constants and those involving quantified variables.]

9.5 SMT example

Let's consider an example. Suppose we start with the following four clauses. Note that a, b, c are constants: variables are not possible with this sort of proof procedure. And the boxes are to remind us that DPLL regards these arithmetic formulas as propositional variables. DPLL knows absolutely nothing about arithmetic.

$$\begin{array}{ll} \{\boxed{c = 0}, \boxed{2a < b}\} & \{\boxed{b < a}\} \\ \{\boxed{3a > 2}, \boxed{a < 0}\} & \{\neg\boxed{c = 0}, \neg\boxed{b < a}\} \end{array}$$

Unit propagation using $\{\boxed{b < a}\}$ yields three clauses:

$$\{\boxed{c = 0}, \boxed{2x < b}\} \quad \{\boxed{3a > 2}, \boxed{a < 0}\} \quad \{\neg\boxed{c = 0}\}$$

Unit propagation using $\neg\boxed{c = 0}$ yields two clauses:

$$\{\boxed{2a < b}\} \quad \{\boxed{3a > 2}, \boxed{a < 0}\}$$

Unit propagation using $\boxed{2a < b}$ yields just this:

$$\{\boxed{3a > 2}, \boxed{a < 0}\}$$

Now a case split on the literal $\boxed{3a > 2}$ returns a "model":

$$\boxed{b < a} \land \neg\boxed{c = 0} \land \boxed{2a < b} \land \boxed{3a > 2}.$$

But the arithmetic decision procedure finds this combination contradictory and returns a new clause:[6]

$$\{\neg\boxed{b < a}, \neg\boxed{2a < b}, \neg\boxed{3a > 2}\}$$

Finally, DPLL returns another model:

$$\boxed{b < a} \land \neg\boxed{c = 0} \land \boxed{2a < b} \land \boxed{a < 0}.$$

As the arithmetic decision procedure finds this to be satisfiable, we learn that the original set of clauses is also satisfiable. A decision procedure may also return specific values for a, b and c.

Case splitting operates as usual for DPLL. But note that pure literal elimination would make no sense here, as there are connections between literals (consider $a < 0$ and $a > 0$ again) that are not visible at the propositional level.

9.6 Final remarks

Here we have seen the concepts of *over-approximation* and *counterexample-driven refinement*. They are frequently used to extend SAT solving to richer domains than propositional logic. By over-approximation we mean that every model of the original problem assigns truth values to the enriched "propositional letters" (such as $a > 0$), yielding a model of the propositional clauses obtained by ignoring the underlying meaning of the propositional atoms. As above, any claimed model is then somehow checked against the richer problem domain, and the propositional model is then iteratively refined. But if the propositional clauses are unsatisfiable, then so is the original problem.

SMT solvers are the focus of great interest at the moment, and have largely superseded SAT solvers (which they incorporate and generalise). One of the most popular SMT solvers is Z3, a product of Microsoft Research but free for non-commercial use. Others include Yices and CVC4. They are applied to a wide range of problems, including hardware and software verification, program analysis, symbolic software execution, and hybrid systems verification.

Exercise 39 In Fourier-Motzkin variable elimination, any variable not bounded both above and below is deleted from the problem. For example, given the set of constraints

$$3x \geq y \quad x \geq 0 \quad y \geq z \quad z \leq 1 \quad z \geq 0$$

the variables x and then y can be removed (with their constraints), reducing the problem to $z \leq 1 \land z \geq 0$. Explain how this happens and why it is correct.

Exercise 40 Apply Fourier-Motzkin variable elimination to the set of constraints

$$x \geq z \quad y \geq 2z \quad z \geq 0 \quad x + y \leq z.$$

Exercise 41 Apply Fourier-Motzkin variable elimination to the set of constraints

$$x \leq 2y \quad x \leq y + 3 \quad z \leq x \quad 0 \leq z \quad y \leq 4x.$$

Exercise 42 Apply the SMT algorithm sketched above to the following set of clauses:

$$\{c = 0, c > 0\} \quad \{a \neq b\} \quad \{c < 0, a = b\}$$

[6] But without $c = 0$: the decision procedure can see that it's irrelevant.

10 Binary Decision Diagrams

A binary decision tree represents the truth table of a propositional formula by binary decisions, namely if-then-else expressions over the propositional letters. (In the relevant literature, propositional letters are called *variables*.) Unfortunately, a decision tree may contain much redundancy. A *binary decision diagram* is a directed acyclic graph, sharing identical subtrees. An *ordered* binary decision diagram is based upon giving an ordering $<$ to the variables: they must be tested in order. Further refinements ensure that each propositional formula is mapped to a unique diagram, for a given ordering. We get a compact and canonical representation of the truth table of any formula.

The acronym BDD for binary decision diagram is well-established in the literature. However, many earlier papers use OBDD or even ROBDD (for "reduced ordered binary decision diagram") synonymously.

A BDD must satisfy the following conditions:

- *ordering*: if P is tested before Q, then $P < Q$ (thus in particular, P cannot be tested more than once on a single path)
- *uniqueness*: identical subgraphs are stored only once (to do this efficiently, hash each node by its variable and pointer fields)
- *irredundancy*: no test leads to identical subgraphs in the 1 and 0 cases (thanks to uniqueness, redundant tests can be detected by comparing pointers)

For a given variable ordering, the BDD representation of each formula is unique: BDDs are a *canonical form*. Canonical forms usually lead to good algorithms — for a start, you can test whether two things are equivalent by comparing their canonical forms.

The BDD of a tautology is 1. Similarly, that of any unsatisfiable formula is 0. To check whether two formulas are logically equivalent, convert both to BDDs and then — thanks to uniqueness — simply compare the pointers.

A recursive algorithm converts a formula to a BDD. All the logical connectives can be handled directly, including $\to$ and $\leftrightarrow$. (Exclusive-or is also used, especially in hardware examples.) The expensive transformation of $A \leftrightarrow B$ into $(A \to B) \land (B \to A)$ is unnecessary.

Here is how to convert a conjunction $A \land A'$ to a BDD. In this algorithm, ${}_XP_Y$ is a decision node that tests the variable P, with a true-link to X and a false-link to Y. In other words, ${}_XP_Y$ is the BDD equivalent of the decision "if P then X else Y".

1. Recursively convert A and A' to BDDs Z and Z'.
2. Check for trivial cases. If $Z = Z'$ (pointer comparison) then the result is Z; if either operand is 0, then the result is 0; if either operand is 1, then the result is the other operand.
3. In the general case, let $Z = {}_XP_Y$ and $Z' = {}_{X'}P'_{Y'}$. There are three possibilities:

   (a) If $P = P'$ then build the BDD ${}_{X\land X'}P_{Y\land Y'}$ recursively. This means convert $X \land X'$ and $Y \land Y'$ to BDDs U and U', then construct a new decision node from P to them. Do the usual simplifications. If $U = U'$ then the resulting BDD for the conjunction is U.

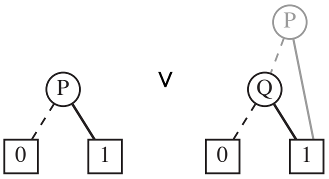
   (b) If $P < P'$ then build the BDD ${}_{X\land Z'}P_{Y\land Z'}$. When building BDDs on paper, it is easier to pretend that the second decision node also starts with P: assume that it has the redundant decision ${}_{Z'}P_{Z'}$ and proceed as in (a).

   (c) If $P > P'$, the approach is analogous to (b).

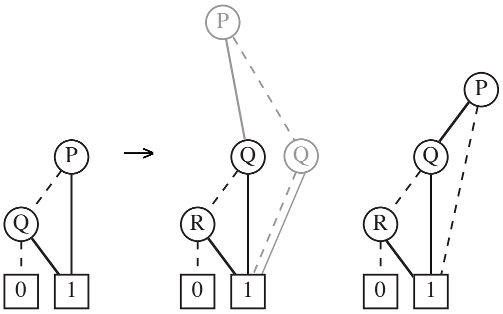
Other connectives, even $\oplus$, are treated similarly, differing only in the base cases. The negation of the BDD ${}_XP_Y$ is ${}_{\neg X}P_{\neg Y}$. In essence we copy the BDD, and when we reach the leaves, exchange 1 and 0. The BDD of $Z \to \mathbf{f}$ is the same as the BDD of $\neg Z$.

During this processing, the same input (consisting of a connective and two BDDs) may be transformed into a BDD repeatedly. Efficient implementations therefore have an additional hash table, which associates inputs to the corresponding BDDs. The result of every transformation is stored in the hash table so that it does not have to be computed again.

Example 34 We apply the BDD Canonicalisation Algorithm to $P \lor Q \to Q \lor R$. First, we make tiny BDDs for P and Q. Then, we combine them using $\lor$ to make a small BDD for $P \lor Q$:



The BDD for $Q \lor R$ has a similar construction, so we omit it. We combine the two small BDDs using $\to$, then simplify (removing a redundant test on Q) to obtain the final BDD.



The new construction is shown in grey. In both of these examples, it appears over the rightmost formula because its variables come later in the ordering.

The final diagram indicates that the original formula is always true except if P is true while Q and R are false. When you have such a simple BDD, you can easily check that it is correct. For example, this BDD suggests the formula evaluates to 1 when P is false, and indeed we find that the formula simplifies to $Q \to Q \lor R$, which simplifies further to 1.

Huth and Ryan [2004] present a readable introduction to BDDs. A classic but more formidable source of information is Bryant [1992].

Exercise 43 Compute the BDD for each of the following formulas, taking the variables as alphabetically ordered:

$$P \land Q \to Q \land P \qquad P \lor Q \to P \land Q$$
$$\neg(P \lor Q) \lor P \qquad \neg(P \land Q) \leftrightarrow (P \lor R)$$

Exercise 44 Verify these equivalences using BDDs:

$$\begin{aligned}(P \land Q) \land R &\simeq P \land (Q \land R)\\ (P \lor Q) \lor R &\simeq P \lor (Q \lor R)\\ P \lor (Q \land R) &\simeq (P \lor Q) \land (P \lor R)\\ P \land (Q \lor R) &\simeq (P \land Q) \lor (P \land R)\end{aligned}$$

Exercise 45 Verify these equivalences using BDDs:

$$\begin{aligned}\neg(P \land Q) &\simeq \neg P \lor \neg Q\\ (P \leftrightarrow Q) \leftrightarrow R &\simeq P \leftrightarrow (Q \leftrightarrow R)\\ (P \lor Q) \to R &\simeq (P \to R) \land (Q \to R)\end{aligned}$$

11 Modal Logics

Modal logic allows us to reason about statements being "necessary" or "possible". Some variants are effectively about time (temporal logic) where a statement might hold "henceforth" or "eventually".

There are many forms of modal logic. Each one is based upon two parameters:

- W is the set of *possible worlds* (machine states, future times, ...)
- R is the *accessibility relation* between worlds (state transitions, flow of time, ...)

The pair (W, R) is called a *modal frame*.

The two *modal operators*, or *modalities*, are $\Box$ and $\Diamond$:

- $\Box A$ means A is *necessarily true*
- $\Diamond A$ means A is *possibly true*

Here "necessarily true' means "true in all worlds accessible from the present one". The modalities are related by the law $\neg\Diamond A \simeq \Box\neg A$; in words, "it is not possible that A is true" is equivalent to "A is necessarily false".

Complex modalities are made up of strings of the modal operators, such as $\Box\Box A$, $\Box\Diamond A$, $\Diamond\Box A$, etc. Typically many of these are equivalent to others; in $S4$, an important modal logic, $\Box\Box A$ is equivalent to $\Box A$.

11.1 Semantics of propositional modal logic

Here are some basic definitions, with respect to a particular frame (W, R):

An *interpretation* I maps the propositional letters to subsets of W. For each letter P, the set $I(P)$ consists of those worlds in which P is regarded as true.

If $w \in W$ and A is a modal formula, then $w \Vdash A$ means A is true in world w. This relation is defined as follows:

$$\begin{aligned}w \Vdash P &\iff w \in I(P)\\ w \Vdash \Box A &\iff v \Vdash A \text{ for all } v \text{ such that } R(w, v)\\ w \Vdash \Diamond A &\iff v \Vdash A \text{ for some } v \text{ such that } R(w, v)\\ w \Vdash A \lor B &\iff w \Vdash A \text{ or } w \Vdash B\\ w \Vdash A \land B &\iff w \Vdash A \text{ and } w \Vdash B\\ w \Vdash \neg A &\iff w \Vdash A \text{ does not hold}\end{aligned}$$

This definition of truth is more complex than we have seen previously (§2.2), because of the extra parameters W and R. We shall not consider quantifiers at all; they really complicate matters, especially if the universe is allowed to vary from one world to the next.

For a particular frame (W, R), further relations can be defined in terms of $w \Vdash A$:

$$\begin{aligned}\models_{W,R,I} A &\text{ means } w \Vdash A \text{ for all } w \text{ under interpretation } I\\ \models_{W,R} A &\text{ means } w \Vdash A \text{ for all } w \text{ and all } I\end{aligned}$$

Now $\models A$ means $\models_{W,R} A$ for all frames. We say that A is *universally valid*. In particular, all tautologies of propositional logic are universally valid.

Typically we make further assumptions on the accessibility relation. We may assume, for example, that R is transitive, and consider whether a formula holds under all such frames. More formulas become universally valid if we restrict the accessibility relation, as they exclude some modal frames from consideration. The purpose of such assumptions is to better model the task at hand. For instance, to model the passage of time, we might want R to be reflexive and transitive; we could even make it a linear ordering, though branching-time temporal logic is popular.

11.2 Hilbert-style modal proof systems

Start with any proof system for propositional logic. Then add the *distribution* axiom

$$\Box(A \to B) \to (\Box A \to \Box B)$$

and the *necessitation* rule:

$$\frac{A}{\Box A}$$

There are no axioms or inference rules for $\Diamond$. The modality is viewed simply as an abbreviation:

$$\Diamond A \stackrel{\text{def}}{=} \neg\Box\neg A$$

The distribution axiom clearly holds in our semantics. The propositional connectives obey their usual truth tables in each world. If A holds in all worlds, and $A \to B$ holds in all worlds, then B holds in all worlds. Thus if $\Box A$ and $\Box(A \to B)$ hold then so does $\Box B$, and that is the essence of the distribution axiom.

The necessitation rule states that all theorems are necessarily true. In more detail, if A can be proved, then it holds in all worlds; therefore $\Box A$ is also true.

The modal logic that results from adding the distribution axiom and necessitation rule is called K. It is a pure modal logic, from which others are obtained by adding further axioms. Each axiom corresponds to a property that is assumed to hold of all accessibility relations. Here are just a few of the main ones:

T	$\Box A \to A$	(reflexive)
4	$\Box A \to \Box\Box A$	(transitive)
B	$A \to \Box\Diamond A$	(symmetric)

Logic T includes axiom T: reflexivity. Logic $S4$ includes axioms T and 4: reflexivity and transitivity. Logic $S5$ includes axioms T, 4 and B: reflexivity, transitivity and symmetry; these imply that the accessibility relation is an equivalence relation, which is a strong condition.

Other conditions on the accessibility relation concern forms of *confluence*. One such condition might state that if w_1 and w_2 are both accessible from w then there exists some v that is accessible from both w_1 and w_2.

11.3 Sequent Calculus Rules for $S4$

We shall mainly look at $S4$, which is one of the mainstream modal logics. It's more intuitive than many of the other variants, and has a particularly clean sequent calculus.

As mentioned above, $S4$ assumes that the accessibility relation is reflexive and transitive. Think of the flow of time. Here are some $S4$ statements with their intuitive meanings:

- $\Box A$ means "A will be true from now on".
- $\Diamond A$ means "A will be true at some point in the future", where the future includes the present moment.
- $\Box\Diamond A$ means "$\Diamond A$ will be true from now on". At any future time, A must become true some time afterwards. In short, A will be true infinitely often.
- $\Box\Box A$ means "$\Box A$ will be true from now on". At any future time, A will continue to be true. So $\Box\Box A$ and $\Box A$ have the same meaning in $S4$.

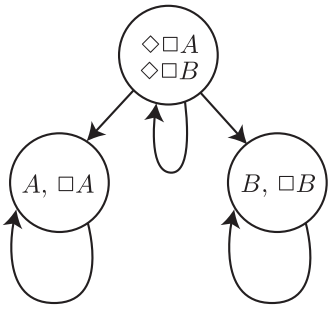


Figure 4: Counterexample to $\Diamond\Box A \land \Diamond\Box B \to \Diamond\Box(A \land B)$

The "time" described by S4 allows multiple futures, which can be confusing. For example, $\Diamond\Box A$ intuitively means "eventually A will be true forever". You might expect $\Diamond\Box A$ and $\Diamond\Box B$ to imply $\Diamond\Box(A \land B)$, since eventually A and B should both have become true. However, this property fails because time can split, with A becoming true in one branch and B in the other (Fig. 4). Note in particular that $\Box\Diamond A$ is stronger than $\Diamond\Box A$, and means "in all futures, eventually A will be true forever".

The sequent calculus for $S4$ extends the usual sequent rules for propositional logic with additional ones for $\Box$ and $\Diamond$. Four rules are required because the modalities may occur on either the left or right side of a sequent.

$$\frac{A,\Gamma\Rightarrow\Delta}{\Box A,\Gamma\Rightarrow\Delta}\,(\Box l)\qquad\qquad \frac{\Gamma^*\Rightarrow\Delta^*,A}{\Gamma\Rightarrow\Delta,\Box A}\,(\Box r)$$

$$\frac{A,\Gamma^*\Rightarrow\Delta^*}{\Diamond A,\Gamma\Rightarrow\Delta}\,(\Diamond l)\qquad\qquad \frac{\Gamma\Rightarrow\Delta,A}{\Gamma\Rightarrow\Delta,\Diamond A}\,(\Diamond r)$$

The $(\Box r)$ rule is analogous to the necessitation rule. But now A may be proved from other formulas. This introduces complications. Modal logic is notorious for requiring strange conditions in inference rules. The symbols Γ^* and Δ^* stand for sets of formulas, defined as follows:

$$\Gamma^* \overset{\text{def}}{=} \{\Box B \mid \Box B \in \Gamma\}$$
$$\Delta^* \overset{\text{def}}{=} \{\Diamond B \mid \Diamond B \in \Delta\}$$

In effect, applying rule $(\Box r)$ in a backward proof throws away all left-hand formulas that do not begin with a $\Box$ and all right-hand formulas that do not begin with a $\Diamond$.

If you consider why the $(\Box r)$ rule actually holds, it is not hard to see why those formulas must be discarded. If we forgot about the restriction, then we could use $(\Box r)$ to infer $A \Rightarrow \Box A$ from $A \Rightarrow A$, which is ridiculous. The restriction ensures that the proof of A in the premise is independent of any particular world.

The rule $(\Diamond l)$ is an exact dual of $(\Box r)$. The obligation to discard formulas forces us to plan proofs carefully. If rules are applied in the wrong order, vital information may have to be discarded and the proof will fail.

11.4 Some sample proofs in $S4$

A few examples will illustrate how the $S4$ sequent calculus is used.

The distribution axiom is assumed in the Hilbert-style proof system. Using the sequent calculus, we can prove it (I omit the $(\to r)$ steps):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A\Rightarrow A}\qquad\dfrac{}{B\Rightarrow B}}{A\to B, A\Rightarrow B}\,(\to l)}{A\to B,\Box A\Rightarrow B}\,(\Box l)}{\Box(A\to B),\Box A\Rightarrow B}\,(\Box l)}{\Box(A\to B),\Box A\Rightarrow \Box B}\,(\Box r)$$

Intuitively, why is this sequent true? We assume $\Box(A \to B)$: from now on, if A holds then so does B. We assume $\Box A$: from now on, A holds. Obviously we can conclude that B will hold from now on, which we write formally as $\Box B$.

The order in which you apply rules is important. Working backwards, you must first apply rule $(\Box r)$. This rule discards non-$\Box$ formulas, but there aren't any. If you first apply $(\Box l)$, removing the boxes from the left side, then you will get stuck:

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{now what?}}{\Rightarrow B}\,?}{A\to B, A\Rightarrow \Box B}\,(\Box r)}{A\to B,\Box A\Rightarrow \Box B}\,(\Box l)}{\Box(A\to B),\Box A\Rightarrow \Box B}\,(\Box l)$$

Applying $(\Box r)$ before $(\Box l)$ is analogous to applying $(\forall r)$ before $(\forall l)$. The analogy because $\Box A$ has an implicit universal quantifier: for all accessible worlds.

The following two proofs establish the modal equivalence $\Box\Diamond\Box\Diamond A \simeq \Box\Diamond A$. Strings of modalities, like $\Box\Diamond\Box\Diamond$ and $\Box\Diamond$, are called *operator strings*. So the pair of results establish an operator string equivalence. The validity of this particular equivalence is not hard to see. Recall that $\Box\Diamond A$ means that A holds infinitely often. So $\Box\Diamond\Box\Diamond A$ means that $\Box\Diamond A$ holds infinitely often — but that can only mean that A holds infinitely often, which is the meaning of $\Box\Diamond A$.

Now, let us prove the equivalence. Here is the first half of the proof. As usual we apply $(\Box r)$ before $(\Box l)$. Dually, and analogously to the treatment of the $\exists$ rules, we apply $(\Diamond l)$ before $(\Diamond r)$:

$$\begin{array}{cl}
\hline
\Diamond A \Rightarrow \Diamond A & \\ \hline
\Box\Diamond A \Rightarrow \Diamond A & (\Box l) \\ \hline
\Diamond\Box\Diamond A \Rightarrow \Diamond A & (\Diamond l) \\ \hline
\Box\Diamond\Box\Diamond A \Rightarrow \Diamond A & (\Box l) \\ \hline
\Box\Diamond\Box\Diamond A \Rightarrow \Box\Diamond A & (\Box r)
\end{array}$$

The opposite entailment is easy to prove:

$$\begin{array}{cl}
\hline
\Box\Diamond A \Rightarrow \Box\Diamond A & \\ \hline
\Box\Diamond A \Rightarrow \Diamond\Box\Diamond A & (\Diamond r) \\ \hline
\Box\Diamond A \Rightarrow \Box\Diamond\Box\Diamond A & (\Box r)
\end{array}$$

Logic $S4$ enjoys many operator string equivalences, including $\Box\Box A \simeq \Box A$. And for every operator string equivalence, its dual (obtained by exchanging $\Box$ with $\Diamond$) also holds. In particular, $\Diamond\Diamond A \simeq \Diamond A$ and $\Diamond\Box\Diamond\Box A \simeq \Diamond\Box A$ hold. So we only need to consider operator strings in which the boxes and diamonds alternate, and whose length does not exceed three.

The distinct $S4$ operator strings are therefore $\Box$, $\Diamond$, $\Box\Diamond$, $\Diamond\Box$, $\Box\Diamond\Box$ and $\Diamond\Box\Diamond$.

Finally, here are two attempted proofs that fail — because their conclusions are not theorems! The modal sequent $A \Rightarrow \Box\Diamond A$ states that if A holds now then it necessarily holds again: from each accessible world, another world is accessible in which A holds. This formula is valid if the accessibility relation is symmetric; then one could simply return to the original world. The formula is therefore a theorem of $S5$ modal logic, but not $S4$.

$$\begin{array}{cl}
\Rightarrow A & \\ \hline
\Rightarrow \Diamond A & (\Diamond r) \\ \hline
A \Rightarrow \Box\Diamond A & (\Box r)
\end{array}$$

Here, the modal sequent $\Diamond A, \Diamond B \Rightarrow \Diamond(A \wedge B)$ states that if A holds in some accessible world, and B holds in some accessible world, then both A and B hold in some accessible world. It is a fallacy because those two worlds need not coincide. The $(\Diamond l)$ rule prevents us from removing the diamonds from both $\Diamond A$ and $\Diamond B$; if we choose one we must discard the other:

$$\begin{array}{cl}
B \Rightarrow A \wedge B & \\ \hline
B \Rightarrow \Diamond(A \wedge B) & (\Diamond r) \\ \hline
\Diamond A, \Diamond B \Rightarrow \Diamond(A \wedge B) & (\Diamond l)
\end{array}$$

The topmost sequent may give us a hint as to why the conclusion fails. Here we are in a world in which B holds, and we are trying to show $A \wedge B$, but there is no reason why A should hold in that world.

The sequent $\Diamond\Box A, \Diamond\Box B \Rightarrow \Diamond\Box(A \wedge B)$ is not valid because A and B can become true in different futures. However, the sequents $\Diamond\Box A, \Box\Diamond\Box B \Rightarrow \Diamond\Box(A \wedge B)$ and $\Box\Diamond\Box A, \Box\Diamond\Box B \Rightarrow \Box\Diamond\Box(A \wedge B)$ are both valid.

Exercise 46 Why does the dual of an operator string equivalence also hold?

Exercise 47 Prove the sequents $\Diamond(A \vee B) \Rightarrow \Diamond A, \Diamond B$ and $\Diamond A \vee \Diamond B \Rightarrow \Diamond(A \vee B)$, thus proving the equivalence $\Diamond(A \vee B) \simeq \Diamond A \vee \Diamond B$.

Exercise 48 Prove the sequent $\Diamond(A \to B), \Box A \Rightarrow \Diamond B$.

Exercise 49 Prove the equivalence $\Box(A \wedge B) \simeq \Box A \wedge \Box B$.

Exercise 50 Prove $\Box\Diamond\Box A, \Box\Diamond\Box B \Rightarrow \Box\Diamond\Box(A \wedge B)$.

12 Tableaux-Based Methods

There is a lot of redundancy among the connectives $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$, $\forall$, $\exists$. We could get away using only three of them (two if we allowed exclusive-or), but use the full set for readability. There is also a lot of redundancy in the sequent calculus, because it was designed to model human reasoning, not to be as small as possible.

One approach to removing redundancy results in the resolution method. Clause notation replaces the connectives, and there is only one inference rule. A less radical approach still removes much of the redundancy, while preserving much of the natural structure of formulas. The resulting formalism, known as the *tableau calculus*, is often adopted by proof theorists because of its simplicity. Adding unification produces yet another formalism known as *free-variable tableaux*; this form is particularly amenable to implementation. Both formalisms use proof by contradiction.

12.1 Simplifying the sequent calculus

The usual formalisation of first-order logic involves seven connectives, or nine in the case of modal logic. For each connective the sequent calculus has a left and a right rule. So, apart from the structural rules (basic sequent and cut) there are 14 rules, or 18 for modal logic.

Suppose we allow only formulas in negation normal form. This immediately disposes of the connectives $\to$ and $\leftrightarrow$. Really $\neg$ is discarded also, as it is allowed only on propositional letters. So only four connectives remain, six for modal logic.

The greatest simplicity gain comes in the sequent rules. The only sequent rules that move formulas from one side to the other (across the $\Rightarrow$ symbol) are the rules for the connectives that we have just discarded. Half of the sequent rules can be discarded too. It makes little difference whether we discard the left-side rules or the right-side rules.

Let us discard the right-side rules. The resulting system allows sequents of the form $A \Rightarrow$. It is a form of refutation system (proof by contradiction), since the formula A has the same meaning as the sequent $\neg A \Rightarrow$. Moreover, a basic sequent has the form of a contradiction. We have created a new formal system, known as the *tableau calculus*.

$$\frac{}{\neg A, A, \Gamma \Rightarrow}\ (basic) \qquad \frac{\neg A, \Gamma \Rightarrow \quad A, \Gamma \Rightarrow}{\Gamma \Rightarrow}\ (cut)$$

$$\frac{A, B, \Gamma \Rightarrow}{A \land B, \Gamma \Rightarrow}\ (\land l) \qquad \frac{A, \Gamma \Rightarrow \quad B, \Gamma \Rightarrow}{A \lor B, \Gamma \Rightarrow}\ (\lor l)$$

$$\frac{A[t/x], \Gamma \Rightarrow}{\forall x\, A, \Gamma \Rightarrow}\ (\forall l) \qquad \frac{A, \Gamma \Rightarrow}{\exists x\, A, \Gamma \Rightarrow}\ (\exists l)$$

Rule $(\exists l)$ has the usual proviso: it holds *provided* x is not free in the conclusion!

We can extend the system to $S4$ modal logic by adding just two further rules, one for $\Box$ and one for $\Diamond$:

$$\frac{A, \Gamma \Rightarrow}{\Box A, \Gamma \Rightarrow}\ (\Box l) \qquad \frac{A, \Gamma^* \Rightarrow}{\Diamond A, \Gamma \Rightarrow}\ (\Diamond l)$$

As previously, Γ^* is defined to erase all non-$\Box$ formulas:

$$\Gamma^* \stackrel{\text{def}}{=} \{\Box B \mid \Box B \in \Gamma\}$$

We have gone from 14 rules to four, ignoring the structural rules. For modal logic, we have gone from 18 rules to six.

A simple proof will illustrate how the tableau calculus works. Let us prove $\forall x\,(A \to B) \Rightarrow A \to \forall x\, B$, where x is not free in A. We must negate the formula, convert it to NNF and finally put it on the left side of the arrow. The resulting sequent is $A \land \exists x\, \neg B,\ \forall x\,(\neg A \lor B) \Rightarrow$. Elaborate explanations should not be necessary because this tableau calculus is essentially a subset of the sequent calculus described in §5.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{A,\ \neg B,\ \neg A \Rightarrow} \quad \cfrac{}{A,\ \neg B,\ B \Rightarrow}}{A,\ \neg B,\ \neg A \lor B \Rightarrow}\ (\lor l)}{A,\ \neg B,\ \forall x\,(\neg A \lor B) \Rightarrow}\ (\forall l)}{A,\ \exists x\,\neg B,\ \forall x\,(\neg A \lor B) \Rightarrow}\ (\exists l)}{A \land \exists x\,\neg B,\ \forall x\,(\neg A \lor B) \Rightarrow}\ (\land l)$$

12.2 The free-variable tableau calculus

Some proof theorists adopt the tableau calculus as their formalisation of first-order logic. It has the advantages of the sequent calculus, without the redundancy. But can we use it as the basis for a theorem prover? Implementing the calculus (or indeed, implementing the full sequent calculus) requires a treatment of quantifiers. As with resolution, a good computational approach is to combine unification with Skolemization.

First, consider how to add unification. The rule $(\forall l)$ substitutes some term for the bound variable. Since we do not know in advance what the term ought to be, instead substitute a free variable. The variable ought to be fresh, not used elsewhere in the proof:

$$\frac{A[z/x], \Gamma \Rightarrow}{\forall x\, A, \Gamma \Rightarrow}\ (\forall l)$$

Then allow unification to instantiate variables with terms. This should occur when trying to solve any goal containing two formulas, $\neg A$ and B. Try to unify A with B, producing a basic sequent. Instantiating a variable updates the entire proof tree.

Up until now, we have treated rule $(\exists l)$ backward proofs as creating a fresh variable. That will no longer do: we now allow variables to become instantiated by terms. To eliminate this problem, we do not include $(\exists l)$ in the free-variable tableau calculus; instead we Skolemize the formula. All existential quantifiers disappear, so we can discard rule $(\exists l)$. This version of the tableau method is known as the *free-variable* tableau calculus.

Warning: if you wish to use unification, you absolutely must also use Skolemization. If you use unification without Skolemization, then you are trying to use two formalisms at the same time and your proofs will be nonsense! This is because unification is likely to introduce variable occurrences in places where they are forbidden by the side condition of the existential rule.

The Skolemised version of $\forall y\, \exists z\, Q(y,z) \land \exists x\, P(x)$ is $\forall y\, Q(y, f(y)) \land P(a)$. The subformula $\exists x\, P(x)$ goes to $P(a)$ and not to $P(g(y))$ because it is outside the scope of the $\forall y$.

12.3 Proofs using free-variable tableaux

Let us prove the formula $\exists x\, \forall y\, [P(x) \to P(y)]$. First negate it and convert to NNF, getting $\forall x\, \exists y\, [P(x) \land \neg P(y)]$. Then Skolemize the formula, and finally put it on the left side of the arrow. The sequent to be proved is $\forall x\, [P(x) \land \neg P(f(x))] \Rightarrow$. Unification completes the proof by creating a basic sequent; there are two distinct ways of doing so:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{z \mapsto f(y) \text{ or } y \mapsto f(z)}{P(y),\ \neg P(f(y)),\ P(z),\ \neg P(f(z)) \Rightarrow}\ basic}{P(y),\ \neg P(f(y)),\ P(z) \land \neg P(f(z)) \Rightarrow}\ (\land l)}{P(y),\ \neg P(f(y)),\ \forall x\, [P(x) \land \neg P(f(x))] \Rightarrow}\ (\forall l)}{P(y) \land \neg P(f(y)),\ \forall x\, [P(x) \land \neg P(f(x))] \Rightarrow}\ (\land l)}{\forall x\, [P(x) \land \neg P(f(x))] \Rightarrow}\ (\forall l)$$

In the first inference from the bottom, the universal formula is retained because it must be used again. In principle, universally quantified formulas ought always to be retained, as they may be used any number of times. I normally erase them to save space.

Pulling quantifiers to the front is not merely unnecessary; it can be harmful. Skolem functions should have as few arguments as possible, as this leads to shorter proofs. Attaining this requires that quantifiers should have the smallest possible scopes; we ought to push quantifiers in, not pull them out. This is sometimes called *miniscope* form.

For example, the formula $\exists x\, \forall y\, [P(x) \to P(y)]$ is tricky to prove, as we have just seen. But putting it in miniscope form makes its proof trivial. Let us do this step by step:

Negate; convert to NNF:	$\forall x\, \exists y\, [P(x) \land \neg P(y)]$
Push in the $\exists y$:	$\forall x\, [P(x) \land \exists y\, \neg P(y)]$
Push in the $\forall x$:	$\forall x\, P(x) \land \exists y\, \neg P(y)$
Skolemize:	$\forall x\, P(x) \land \neg P(a)$

The formula $\forall x\, P(x) \land \neg P(a)$ is obviously unsatisfiable.

Here is its refutation in the free-variable tableau calculus:

$$\cfrac{\cfrac{\cfrac{y \mapsto a}{P(y),\ \neg P(a) \Rightarrow}\ \textit{basic}}{\forall x\, P(x),\ \neg P(a) \Rightarrow}\ (\forall l)}{\forall x\, P(x)\ \land\ \neg P(a) \Rightarrow}\ (\land l)$$

A failed proof is always illuminating. Let us try to prove the invalid formula

$$\forall x\, [P(x) \lor Q(x)] \to [\forall x\, P(x) \lor \forall x\, Q(x)].$$

Negation and conversion to NNF gives

$$\exists x\, \neg P(x) \land \exists x\, \neg Q(x) \land \forall x\, [P(x) \lor Q(x)].$$

Skolemization gives $\neg P(a) \land \neg Q(b) \land \forall x\, [P(x) \lor Q(x)]$.

The proof fails because a and b are distinct constants. It is impossible to instantiate y to both simultaneously. The following proof omits the initial $(\land l)$ steps.

$$\cfrac{\cfrac{\cfrac{y \mapsto a}{\neg P(a), \neg Q(b), P(y) \Rightarrow} \qquad \cfrac{y \mapsto b???}{\neg P(a), \neg Q(b), Q(y) \Rightarrow}}{\neg P(a),\ \neg Q(b),\ P(y) \lor Q(y) \Rightarrow}\ (\lor l)}{\neg P(a),\ \neg Q(b),\ \forall x\, [P(x) \lor Q(x)] \Rightarrow}\ (\forall l)$$

12.4 Tableaux-based theorem provers

A tableau represents a partial proof as a set of *branches* of formulas. Each formula on a branch is *expanded* until this is no longer possible (and the proof fails) or until the proof succeeds.

Expanding a conjunction $A \land B$ on a branch replaces it by the two conjuncts, A and B. Expanding a disjunction $A \lor B$ splits the branch in two, with one branch containing A and the other branch B. Expanding the quantification $\forall x\, A$ extends the branch by a formula of the form $A[t/x]$. If a branch contains both A and $\neg A$ then it is said to be *closed*. When all branches are closed, the proof has succeeded.

A tableau can be viewed as a compact, graph-based representation of a set of sequents. The branch operations above correspond to sequent rules in an obvious way.

Quite a few theorem provers have been based upon free-variable tableaux. The simplest is due to Beckert and Posegga [1994] and is called lean$T^A P$. The entire program appears below! Its deductive system is similar to the reduced sequent calculus we have just studied. It relies on some Prolog tricks, and is certainly not pure Prolog code. It demonstrates just how simple a theorem prover can be. lean$T^A P$ does not outperform big resolution systems. But it quickly proves some fairly hard theorems.

```
prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,[B|UnExp],Lits,FreeV,VarLim).
prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,UnExp,Lits,FreeV,VarLim),
    prove(B,UnExp,Lits,FreeV,VarLim).
prove(all(X,Fml),UnExp,Lits,FreeV,VarLim) :- !,
    \+ length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove(Fml1,UnExp1,Lits,[X1|FreeV],VarLim).
prove(Lit,_,[L|Lits],_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove(Lit,[],Lits,_,_)).
prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
    prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).
```

The first clause handles conjunctions, the second disjunctions, the third universal quantification. The fourth line handles literals, including negation. The fifth line brings in the next formula to be analyzed.

You are not expected to memorize this program or to understand how it works in detail.

Exercise 51 Use the free variable tableau calculus to prove these formulas:

$$(\exists y\, \forall x\, R(x,y)) \to (\forall x\, \exists y\, R(x,y))$$
$$(P(a,b) \lor \exists z\ P(z,z)) \to \exists xy\, P(x,y)$$
$$(\exists x\, P(x) \to Q) \to \forall x\, (P(x) \to Q)$$

Exercise 52 Compare the sequent calculus, the free-variable tableau calculus and resolution by using each of them to prove the following formula:

$$(P(a,b) \lor \exists z\, P(z,z)) \to \exists xy\, P(x,y)$$

(From the 2012 exam, Paper 6 Question 6.)

References

B. Beckert and J. Posegga. leanTAP: Lean, tableau-based theorem proving. In A. Bundy, editor, *Automated Deduction — CADE-12 International Conference*, LNAI 814, pages 793–797. Springer, 1994.

R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *Computing Surveys*, 24(3): 293–318, Sept. 1992.

M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322186.322198.

M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.