MSc Module CS612

Notes on Automated Reasoning

Rajeev Goré
Modified by Martin Peim and Alan Williams

October 2006

Abstract

These notes are designed to be an introductory course in Automated Reasoning for Computer Science students. Thus they assume no prior knowledge of logic, but assume some knowledge of computer science. Please email any feedback about errors, ommissions or typographical errors to `alanw@cs.man.ac.uk`.

Contents

1 Classical Propositional Logic

This section is intended as an introductory course in propositional logic for students with no prior knowledge of formal logic. I have assumed some basic familiarity with computer science and set notation. It would be useful to read these notes in conjunction with an introductory text book such as [1]. It is also especially important to attempt as many exercises as you can in order to familiarise yourself with each concept — [1] contains many worked examples and exercises, and the pre-course material for the CS612 MSc module has additional exercises for you to try.

As in any field, mathematical logic is full of jargon and apparently impenetrable notation. The purpose of these notes is to introduce this jargon and to explain the notation. Unfortunately, there is no generally accepted notation for some of the jargon and so I have had to make my own choices.

The study of formal logic (in the "West") originated in Greek times when philosophers tried to formalise the arguments we use in natural language discourse. The simplest formalisation is called Classical (two-valued) Logic and this is the subject of these notes. One of the most basic tenets of Classical Logic is that every statement is either true or false; this is called the law of the excluded middle. Note that there are many "non-classical" approaches which are becoming very fashionable in Computer Science. For example, there exist equally formal and rigorous non-classical logics where statements may have a third value standing for "undefined" or "unknown". There are also non-classical logics where the truth value of a statement may change over time.

1.1 Syntax

We shall use two languages. One is the **object** level language which is the language of the logic itself (and one which we shall rarely use). The other is a **meta-language** which is used to speak about the logic. We shall use the words "denoted by" when we explain the meta-linguistic notation for an object level concept. We shall use $=$ as a meta-linguistic symbol when describing the shape of some of the objects.

Our object level language consists of **propositions**, **connectives**, **formulae** and punctuation symbols "(" and ")" as defined below.

Let $\mathcal{P}$ be a set of arbitrary symbols say A, B possibly with subscripts (e.g. A_1) or prime signs (e.g. A'). Any element of $\mathcal{P}$ is called an **atomic proposition** or an **atom**. The elements of $\mathcal{P}$ are from the object language.

The symbols $\wedge, \vee, \neg, \rightarrow$ are called logical connectives and are also from the object language. In propositional logic they stand for:

Symbol	Logical Name	Linguistic Origin
$\wedge$	logical conjunction	"and"
$\vee$	logical disjunction	"(inclusive) or"
$\neg$	logical negation	"not"
$\rightarrow$	logical implication	"if ... then ..."

Note that these do not correspond exactly to *linguistic connectives*. For example, linguistically, we often say things like “red or brown” to mean “red or brown or both” in one context but to mean “either red or brown but not both” in other contexts. Each logical connective has a single meaning *defined* by its semantics; see later.

A **formula** is defined as follows:

any member of $\mathcal{P}$ is a formula (called an **atomic formula**);

both **t** and **f** are (also atomic) formulae;

if F and G are formulae then so are: $(F \wedge G)$, $(F \vee G)$, $(\neg F)$, $(\neg F)$, $(F \rightarrow G)$.

The following are formulae: $A \wedge B$, $(\neg A) \vee A$, $\neg(B \rightarrow A)$.

The following are not formulae: $\wedge AB$, $\wedge \vee B$, $\neg \rightarrow B$.

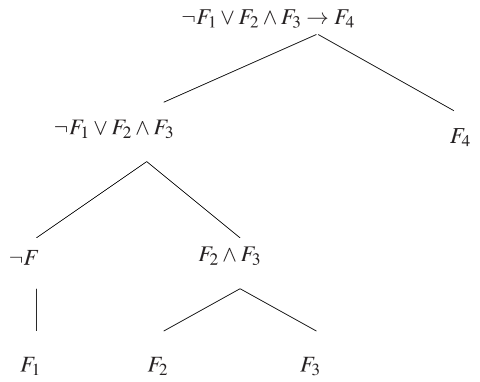
For later use, a **literal** is either an atomic formula or the negation of an atomic formula. Thus both A and $\neg A$ are literals but $\neg\neg A$ is not a literal. The literals A and $\neg A$ form a *complementary pair*.

If L is any literal then we define $\overline{L}$ to be the **complement** of L: if $L = A$ then $\overline{L} = \neg A$ and if $L = \neg A$ then $\overline{L} = A$. Note that $\overline{L}$ is also a literal and that literals L and $\overline{L}$ form a complementary pair.

We will use capital letters like F, G to denote arbitrary formulae, sometimes with subscripts; thus these are part of the meta-language. Sometimes we want to specify some of the structure of a formula so we say that expressions like $F \wedge G$ and $F \vee G$ are also formulae although these are really part of the meta-language since they stand for any formula whose main connective happens to be $\wedge$ and $\vee$ respectively. Thus, we often use the logical connectives as if they were part of the meta-language.

To reduce punctuation parentheses we use an implicit precedence for the way the connectives bind their arguments. The **precedence** is: $\neg$ binds tighter than $\wedge$ binds tighter than $\vee$ binds tighter than $\rightarrow$. For example, $\neg F_1 \vee F_2 \wedge F_3 \rightarrow F_4$ should be read as $((\neg F_1) \vee (F_2 \wedge F_3)) \rightarrow F_4$.

Another way to display a formula unambiguously is to use a parse tree or formation tree. The formation tree of the formula given above is:



In fact, this “abstract” syntax is probably how a formula will be “seen” by a theorem-proving system. The crucial point about a propositional formula is that it is either atomic or has a uniquely defined top-level connective with a uniquely defined sequence of subformulae as operands. The notation with the parentheses is for the benefit of human readers only!

1.2 Semantics

The semantics (or meaning) for classical propositional logic is given in terms of *interpretations* (of the atomic propositions). The idea is that the truth (or falsehood) of a formula depends on the truth of the propositions it contains. Formally an **interpretation** is a function v that assigns one of the **truth values T** or **F** to each of the atomic propositions. So we can write

$$v : \mathcal{P} \mapsto \{\mathbf{T}, \mathbf{F}\}.$$

If $v(A) = \mathbf{T}$ then we say that A is *true* under the interpretation v. If $v(A) = \mathbf{F}$ then we say that A is *false* under the interpretation v. In the classical propositional logic which we are dealing with here each proposition must be given a definite truth value—there is no “maybe”.

Any interpretation v of the atomic propositions can be extended to an interpretation of arbitrarily complex formulae by the following rules where F and G are arbitrary formulae:

$v(\mathbf{t}) = \mathbf{T}$ (for any v);

$v(\mathbf{f}) = \mathbf{F}$ (for any v);

$v(\neg F) = \mathbf{T}$ if $v(F) = \mathbf{F}$;

$v(\neg F) = \mathbf{F}$ if $v(F) = \mathbf{T}$;

$v(F \wedge G) = \mathbf{T}$ if $v(F) = \mathbf{T}$ and $v(G) = \mathbf{T}$;

$v(F \wedge G) = \mathbf{F}$ if $v(F) = \mathbf{F}$ or $v(G) = \mathbf{F}$ or both;

$v(F \vee G) = \mathbf{T}$ if $v(F) = \mathbf{T}$ or $v(G) = \mathbf{T}$ or both;

$v(F \vee G) = \mathbf{F}$ if $v(F) = \mathbf{F}$ and $v(G) = \mathbf{F}$;

$v(F \rightarrow G) = \mathbf{T}$ if $v(F) = \mathbf{F}$ or $v(G) = \mathbf{T}$ or both;

$v(F \rightarrow G) = \mathbf{F}$ if $v(F) = \mathbf{T}$ and $v(G) = \mathbf{F}$.

These are *definitions* of how to extend interpretations to arbitrary formulae and they should be used as a "look-up" table from left to right. They are carefully designed to ensure that any arbitrary formula evaluates to *either* **T** or **F** as long as the underlying interpretation on atomic formulae is two-valued (often called Boolean after George Boole).

For example, if the interpretation v puts $v(A) = \mathbf{T}$ and $v(B) = \mathbf{F}$ then we can evaluate the formulae given below:

interpretation	atomic formulae					
-	A	B	$A \vee B$	$B \rightarrow A$	$\neg(A \vee B)$	$(A \vee B) \rightarrow (B \rightarrow A)$
v	**T**	**F**	**T**	**T**	**F**	**T**

1.3 An aside

Note that $\wedge$ corresponds to our intuitions about "and" while $\vee$ corresponds to our intuitions of "inclusive or" but implication does not always correspond to our intuitions about "if then". For example, $A \rightarrow B$ is *defined* to be true under any interpretation that makes A false or that makes B true whereas "if A then B" in English usually connotes some sort of *causality* as in "if you don't eat your vegetables then you won't get any dessert".

1.4 More Definitions

A formula F is **logically equivalent** to a formula G if and only if $v(F) = v(G)$ for *all* interpretations v. We shall write this as $F =\mid\models G$ (the reason for the apparently cumbersome notation is explained below).

The formula F is a **subformula** of G if the formation tree of F appears as a subtree of the formation tree of G. The formula F is a **proper subformula** of G if F is a subformula of G but $F \neq G$. Note that if F is a subformula of G then F may occur more than once in G, e.g $F = A$ and $G = (A \wedge (A \rightarrow B))$.

An **occurrence of** F **in** G is one particular instance of the formation tree for F in the formation tree for G. Note that an occurrence is a *particular* subtree of another tree.

If F is a subformula of G and F' is an arbitrary formula then the **substitution of** F' **for** F **in** G is denoted by $G\{F \leftarrow F'\}$ and is the result of *simultaneously* replacing *all* occurrences of F in G by F'.

For example, if $F = A$, $G = (A \wedge (A \rightarrow B))$ and $F' = (A \wedge B)$ then $G\{F \leftarrow F'\} = (A \wedge B) \wedge ((A \wedge B) \rightarrow B)$.

Theorem 1 *Suppose F is a subformula of G and suppose A' is an arbitrary formula such that $F =\mid\models F'$. Then $G =\mid\models G\{F \leftarrow F'\}$.*

The following is a list of some of the more important logical equivalences in propositional logic:

$$
\begin{array}{ll}
F \vee G =\mid\models G \vee F & F \wedge G =\mid\models G \wedge F \\
F \vee F =\mid\models F & F \wedge F =\mid\models F \\
F =\mid\models \neg\neg F & F \rightarrow G =\mid\models \neg F \vee G \\
F \rightarrow G =\mid\models \neg(F \wedge \neg G) & F \rightarrow G =\mid\models \neg G \rightarrow \neg F \\
F \wedge G =\mid\models \neg(\neg F \vee \neg G) & F \vee G =\mid\models \neg(\neg F \wedge \neg G) \\
F \rightarrow \mathbf{f} =\mid\models \neg F & \neg(F \rightarrow G) =\mid\models (F \wedge \neg G)
\end{array}
$$

1.5 Satisfiability, Unsatisfiability and Validity

A formula F is **satisfiable** iff there exists at least one interpretation v such that $v(F) = \mathbf{T}$.

A formula F is **falsifiable** iff there exists at least one interpretation v such that $v(F) = \mathbf{F}$.

A formula F is **valid** iff $v(F) = \mathbf{T}$ for all interpretations v. We write $\models F$ to mean F is valid. If F is valid then it is called a **tautology**.

A formula F is **unsatisfiable** iff $v(F) = \mathbf{T}$ for no interpretation v. That is, iff $v(F) = \mathbf{F}$ for all interpretations v.

Theorem 2 *A formula F is valid if and only if (iff) $\neg F$ is unsatisfiable. Hence F is satisfiable iff $\neg F$ is falsifiable.*

A set of formulae $X = \{F_1, F_2, \cdots, F_k\}$ is **(jointly) satisfiable** if there exists some interpretation v such that $v(F_1) = v(F_2) = \cdots = v(F_k) = \mathbf{T}$.

The parentheses around "jointly" indicate that this word is usually omitted. So we shall usually just say "X is satisfiable" instead of the more pedantic "set X is jointly satisfiable".

Theorem 3 *If $X = \{F_1, F_2, \cdots, F_n\}$ and X is satisfiable then $X \setminus \{F_i\}$ is satisfiable for all i with $1 \leq i \leq n$. Here $X \setminus \{F\}$ denotes the set obtained by deleting (all copies of) F from X.*

Theorem 4 *If $X = \{F_1, F_2, \cdots, F_n\}$ is satisfiable and G is valid then $X \cup \{G\}$ is satisfiable. Here $X \cup \{G\}$ denotes the set obtained by adding G to the set X.*

Theorem 5 *If $X = \{F_1, F_2, \cdots, F_n\}$ is unsatisfiable then $X \cup \{G\}$ is unsatisfiable for any formula G.*

Theorem 6 *If $X = \{F_1, F_2, \cdots, F_n\}$ is unsatisfiable and some F_i is valid, where $1 \leq i \leq n$, then $X \setminus \{F_i\}$ is also unsatisfiable.*

1.6 Logical Consequence

Suppose $X = \{F_1, F_2, \cdots, F_n\}$ is a set of formulae and F is a formula. Then F is a **logical consequence** of X iff every interpretation that (jointly) satisfies X also satisfies F. If we write $X \models F$ to denote this notion then:

$$X \models F \text{ iff for all interpretations } v, \text{ if } v(F_1) = v(F_2) = \cdots v(F_n) = \mathbf{T} \text{ then } v(F) = \mathbf{T}.$$

Observe that F is valid ($\models F$) if and only if it is a logical consequence of the empty set of formulae ($\emptyset \models F$). This is because every interpretation satisfies the empty set trivially—there's nothing for it to do.

Note that the interpretation v remains fixed as we transfer attention from X to F. Note also that we have to consider only those interpretations that make all the formulae in X true to determine if F is a logical consequence of X. In particular, if X is unsatisfiable then we can conclude that $X \models F$ for *any* formula F.

Theorem 7 *$X \models F$ iff the set $X \cup \{\neg F\}$ is unsatisfiable.*

We shall see that this observation is the basis of most automated deduction systems. We usually omit the extra braces and write $X \cup \{\neg F\}$ as $X \cup \neg F$.

It should be clear that if F and G are logically equivalent formulae then they are logical consequences of each other, in the sense that $\{F\} \models G$ and $\{G\} \models F$. It is customary to drop the brackets around singleton sets and write $\{F\} \models G$ as $F \models G$. Hence the odd notation for logical equivalence $F \mathrel{=\!\!|\!\models} G$.

Theorem 8 *Let $X = \{F_1, F_2, \cdots, F_n\}$. Then $X \models F$ iff $\models F_1 \wedge F_2 \wedge \cdots \wedge F_n \rightarrow F$.*

Theorem 9 *Let $X = \{F_1, F_2, \cdots, F_n\}$. If $X \models F$ and G is any formula then $X \cup \{G\} \models F$.*

Theorem 10 *Let $X = \{F_1, F_2, \cdots, F_n\}$. If $X \models F$ and G is any valid formula then $X \setminus \{G\} \models F$.*

Logical consequence is an extremely important concept because it gives us a way of formalising reasoning of the form "if we assume X is true then F is also true".

1.7 Computing Logical Consequence

Suppose that we have a systematic way to attempt to construct an interpretation that (jointly) satisfies some given set of formulae Y. If this systematic attempt terminates with success then we can say that Y is satisfiable. But, if this systematic attempt terminates with failure then we can declare Y to be unsatisfiable. In particular, if $Y = X \cup \{\neg F\}$ then we can declare $X \cup \{\neg F\}$ to be unsatisfiable or satisfiable; but then $X \models F$ or $X \not\models F$ respectively.

Thus what we would like is a purely *syntactic* test for satisfiability (so that a computer can carry out these (syntactic) manipulations).

2 Truth Tables

The most obvious method for computing logical consequence is called the method of truth tables and we have already seen it in action on page 4. To determine if $X \models F$ for some finite set X we examine every possible interpretation v, first to check if v satisfies X and then to check whether it also satisfies F. This involves only a finite amount of work (fortunately!) since the truth value of a formula depends only on the truth values of the propositions contained in it, so we only need consider interpretations of a finite number of propositions. The idea is as follows:

1. consider (draw) the formation tree for every formula in X and also the formation tree for the formula F;
2. the leaves of these trees are the atomic formulae which appear in $X \cup F$ since they cannot be decomposed any further;
3. suppose these atomic formulae are $A_1, A_2, \cdots A_n$;
4. the relevant interpretations are all the valuations that map $\{A_1, A_2, \cdots A_n\} \mapsto \{\mathbf{T}, \mathbf{F}\}$;
5. there are 2^n such interpretations (this is just the number of different binary numbers that can be made from n bits!).
6. now consider each such interpretation in turn discarding any which make some member of X false (since we only want the ones that satisfy X);
7. if all the remaining interpretations also satisfy F (i.e. make it true) then we can declare that $X \models F$; otherwise declare $X \not\models F$.

Example 1 *Consider the formula $(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$. The formation trees have been omitted but parts of the truth table are given below:*

interpretation	*atomic formulae*						
-	A	B	$A \rightarrow B$	$\neg B$	$\neg A$	$\neg B \rightarrow \neg A$	$(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$
v_1	***F***	***F***	***T***	***T***	***T***	***T***	***T***
v_2	***F***	***T***	***T***	***F***	***T***		
v_3	***T***	***T***	***T***	***F***	***F***		
v_4	***T***	***F***	***F***	***T***	***F***		

2.1 Exercises: Propositional Logic Introduction

Prove each of the following formulae using truth tables.

You should also return to prove these formulae using other proof methods introduced during the course, in particular resolution and, later on, semantic tableaux.

1. $A, (A \to B) \models B$
2. $A, (A \to B), (B \to C) \models C$
3. $\models ((A \to B) \land (B \to C)) \to (A \to C)$
4. $A \lor B =\!\mid\models B \lor A$
5. $A \lor (B \lor C) =\!\mid\models (A \lor B) \lor C$
6. $(A \to C), (B \to C) \models (A \land B) \to C$
7. $A \to (B \to C) =\!\mid\models (A \land B) \to C$
8. $(A \to B), (A \to C) \models A \to (B \land C)$
9. $A \to (B \to (C \to D)) \models C \to (A \to (B \to D))$
10. $A \land (B \lor C) \models (A \land B) \lor (A \land C)$
11. $(A \lor B) \land (A \lor C) \models A \lor (B \land C)$
12. $\models A \to (B \to C) \to ((A \to B) \to (A \to C))$
13. $\models (A \to B) \to ((C \lor A) \to (C \lor B))$
14. $B, \neg(A \land B) \models \neg A$
15. $\neg(\neg A \lor \neg B) \models A$
16. $\neg(A \land B) =\!\mid\models \neg A \lor \neg B$
17. $(\neg A \to C), (\neg B \to C) \models \neg(A \land B) \to C$
18. $\models (((A \land B) \land C) \land D) \lor (E \to E)$
19. $\models (\neg A \land \neg B) \lor (\neg A \land B) \lor (A \land \neg C) \lor (A \land C)$
20. $A \to B \land B \to A =\!\mid\models (A \land B) \lor (\neg A \land \neg B)$

3 Prolog

This section will give a brief introduction to Prolog, the most famous Logic programming language. It will not attempt to give a comprehensive description, but will only cover the fundamentals of the language. For comprehensive coverage, the reader should refer to one of the many books on Prolog (e.g. [2], [3]). There are also numerous on-line tutorials, manuals and examples available.

The main purpose of discussing Prolog is so that we can later explore its relationship with resolution, in section 9. It is also clear that Prolog is a useful tool in its own right for expressing and solving computational problems, in particular those related to artificial intelligence.

3.1 Using the Prolog System

We give a very brief overview of using the SWI-Prolog system[1].

To start the system type:

```
pl
```

This will produce the following response:

```
Welcome to SWI-Prolog (Version 4.0.10)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

The '`?-`' prompt is indicating that the system is ready to accept a goal from the user (terminated by a full-stop). In order to quit the Prolog system, type '`halt`'.

As the Welcome message says, there is an on-line help system available — simply type '`help`', or '`help(Topic)`'. Sections in the on-line manual can be accessed by giving the section number, for example:

```
?- help.
?- help(help).
?- help(2).
```

There is also a large amount of useful documentation available from the SWI-Prolog web-site.

The first thing you will wish to do is to enter facts or rules.

[1] The SWI-Prolog system is available from `http://www.swi-prolog.org` for Linux and Windows.

Firstly, you can enter facts or rules by typing them directly into the system. Finish input by typing ‘`Ctrl-D`’.

For example, to enter the facts ‘`parent(sue,toby)`’ and ‘`parent(sue,jules)`’, type:

```
?- [user].
|: parent(sue,toby) .
|: parent(sue,jules) .
% user compiled 0.01 sec, 540 bytes

Yes
?-
```

Clearly, for anything but the smallest programs, it would be best to create a text file separately and then to load it into the system.

For example, you can load a file called `examples.pl` by typing:

```
?- ['examples.pl'].
```

You could also enter a comma-separated list of files here.

You can inspect the definitions already added by typing ‘`listing`’.

Finally, the `trace` predicate starts the trace facility which shows the rules and facts visited while the system attempts to satisfy a goal.

3.2 Programs: Facts, Rules and Procedures

A Prolog program consists of a number of *facts* and *rules*. The program is invoked by providing it with a set of *goal* claues. the system will attempt to determine if these can be deduced from the program. The goal clauses may also contain variables. In this case the system will determine whether the clauses can be deduced, given an assignment of literals to the variables. There may be more than one assignment under which the goal clauses can be deduced.

The *term* is the basic syntactic construct within Prolog, defined recursively:

- Ground Terms:

  Constants: strings beginning with a lower-case letter

  Variables: strings beginning with an upper-case letter, or an underscore.
- Compound Terms: $functor(arg_1, \ldots, arg_k)$
- A *functor* is a string beginning with a lower-case letter

Facts define relationships between objects and are predicates which are intepreted as true. For example:


```
parent(jim,rita).
parent(jim,rajeev).
```

are two facts which can be thought of as saying that `jim` is `parent` of `rita` and of `rajeev`. A Prolog program may have several `parent` facts, providing a *database* of information.

Simple *goals* can now be given. For example if the Prolog system contained just the above fact then the goal:

```
:- parent(jim,rita).
```

could be deduced and the system would return the result `Yes`. However the goal

```
:- parent(jim,alan).
```

could not be deduced and the system would return `No`.

Given the goal

```
:- parent(jim,X).
```

containing the variable `X`, the system would attempt to deduce this by trying substituting various terms for `X`. It would be able to succeed here and return first with `X = rita` and then with `X = rajeev`. Here we are asking the system to calculate who `jim` is the parent of.

Sets of *rules* now can be added which allow more information to be deduced, showing further connections between facts. Rules have the following structure:

```
<head> :- <body>
```

where `<head>` is a clause and `<body>` is a comma-separated list of clauses, all possibly (and usually) containing variables. Notice that a goal is simply a headless rule.

When a set of rules each have the same-name clause as their head, then they form a *procedure*. For example, the following rules define a procedure for determining whether one person is the ancestor of another, assuming that some `parent` facts are available:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Rules are interpreted in the following way: *if* all of the goals in the body hold, *then* the head literal holds, possibly requiring some substitution of variables. In order to find if the goals in the body of the clause hold, then they need to be matched with the head of another clause, in effect a call to another procedure. In this case the rules for that procedure will be used in an attempt to deduce the body clause.

The above two rules represent our intuition about when a person X is an ancestor of person Y: this is either because X is a parent of Y, or because X is a parent of some other person Z who themselves is an ancestor of Y. Notice that the definition is naturally recursive, with the ancestor goal in the body of the second rule being a recursive 'call' to the ancestor procedure.

Now let us include some more parent facts, repeating the whole program so far:

The order of the rules and facts is important, since Prolog will try them in the order given until it manages to find one that matches the current goal clause under consideration. If it fails to match one of the goals, it will then *back-track* to the last point where it made a choice of rule or fact and try again with another.

We can now find out whether the following goals would succeed, and what substitutions of any variables contained within them would be required:

```
:- ancestor(roy,sue).
:- ancestor(jim,john).
:- ancestor(jim,X).
:- ancestor(X,toby).
:- ancestor(frances,john).
```

3.3 Built-Ins: Arithmetic and Lists

A comprehensive range of standard logical, integer and real operators is included in the SWI Prolog system. We will not attempt to cover them in detail here — see, for example, the on-line help system for a guide (type apropos(arithmetic)).

SWI-Prolog provides a built-in *list* datatype. It uses the standard convention for its representation, by defining the *empty list* operator and the *cons* operator:

```
[]
[1|[2]]
[1|[hello, world,[2,3]]]
[Head|Tail]
```

For the *cons*, the second argument must itself be a list (or a variable, as in the above example, which can be bound to a list). Also, as would be expected from an untyped language, and in contrast to languages such as SML, a Prolog list can contain elements of different 'types' — essentially the elements of a list are all terms.

SWI-Prolog includes a large number of built-in list operators — details of these can be found, for example, via the on-line help system (type apropos(list)).

3.4 Some of The Classic Example Prolog Procedures

Apart from an obsession with family trees (an ideal area for finding examples about relations), a second rich vein of examples concerns lists. Clearly, this is because the list structure is the

simplest recursive data type (and is also very useful — just consider the Lisp programming language, which has nothing else).

We will include some of the classics here, many of which are built into standard Prolog system. For more examples, the reader should refer to the Prolog text books[3, 2].

A procedure to find the length of a list illustrates several useful Prolog features:

- pattern-matching for lists, in particular a non-empty list cannot match the first rule
- a recursive procedure, having a base case rule and a recursive rule
- arithmetic expression evaluation, using is.
- the importance of goal order
- as given here, arguably a poor choice of variable names!

```
mylength([],0).
mylength([X|L1],N) :- mylength(L1,M), N is M+1.
```

The member procedure shows how to test for inequality of terms:

```
member(X, [X|L1]).
member(X, [Y|L1]) :- X \== Y, member(X,L1).
```

Notice the similar structure to mylength. Similarly, with finding the prefix of a list:

```
prefix([],L2).
prefix([X|L1],[X|L2]) :- prefix(L1,L2).
```

The append procedure again has a *result* variable. Here, the base rule ensures that this is set to equal the second list at the end of the recursion. Each recursive rule call transfers the head of the first list to the head of the accumulated list. In this rule, L3 holds the result so far, i.e. the effect of appending the tail of the first list L1 onto L2:

```
myappend([],L3,L3).
myappend([X|L1],L2,[X|L3]) :- myappend(L1,L2,L3).
```

The 'standard' use of list append would be to give two concrete list arguments and a third variable argument to collect the result of the append. It is interesting to observe what happens when append is used in a different way, for example:

```
myappend(X,Y,[1,2,3]).
```

Now follow two ways of implementing a list reversal procedure. The first (very inefficient) method, myrev, uses list append, to add the head of the list X on to the end of the result so far L3.

```
myreverse([],[]).
myreverse([X|L1],L2) :- myreverse(L1,L3), myappend(L3,[X],L2).

myrev([], L2, L2).
myrev([X|L1],Acc,L3) :- myrev(L1,[X|Acc],L3).

rev2([], L3,L3).
rev2([X|L1],Acc,L3) :- rev2(L1,[X|Acc],L3).
myrev2(L1,Res) :- rev2(L1, [], Res).
```

The second implementation `myrev2` uses an auxiliary procedure `rev2`. Viewing this operationally, an *accumulator variable* `Acc` is used to hold the result so far. `L3` will hold the result which is accumulated in `Acc`, as each element `X` is pushed onto it, until the base-case fact. Here, the accumlator is 'passed' to the result and `Res` is eventually assigned to this as the recursion is unwound. It is very illuminating to trace the evaluation of these procedures.

Finally, we give an implementation of an integer list partition procedure, which could be used as the basis of an implementation of the *quicksort* algorithm:

```
partition(X,[],[],[]).
partition(X,[Y|L2],[Y|Smaller],Bigger) :- Y < X,
              partition(X,L2,Smaller,Bigger).
partition(X,[Y|L2],Smaller,[Y|Bigger]) :- Y >= X,
              partition(X,L2,Smaller,Bigger).
```

This takes an integer *pivot* element `X` and an integer list, and then splits the list into two, one containing all elements less than `X`, the other containing all elements greater than or equal to `X`. Essentially there are three different disjoint cases: the first base-case rule when the empty list is reached, the second when the head of the input list is less than the pivot and the third when the head is greater than or equal to the pivot.

3.5 User-Defined Operators

When defining new operators, Prolog provides the facility, using the `op` predicate, for defining the precedence and whether they are infix (for binary operators) or prefix:

```
op(N,opname,template)
```

`N` is an integer in the range 0. . . 1200, giving the precedence class for the operator, where *lower* values give an operator a *higher* precedence. `opname` is the name of the operator (this can be a list). `template` is used to indicate associativity and position, with the following patterns:

- `fx`, `fy` for prefix operators
- `xfy`, `xfx`, `yfx` for infix operators, where `yfx` is left associate `xfy` is right associate

`x` indicates that the argument must be a term with lower *precedence class* and `y` is a term with same or lower *precedence class*.

The `display` predicate is useful to show the structure of a term. When using `op` it is important not to forget to precede it with '`:-`', since `op` is a predefined predicate.

For example, the following declarations define the unary `succ` operator and infix binary `plus` operator:

```
?- [user].
|: :- op(300,fx,succ).
|: :- op(400,xfy,plus).
|:

?- display(succ x plus succ y).
plus(succ(x), succ(y))
?- display(x plus y plus z).
plus(x, plus(y, z))
```

However, if the relative precedence of the two operators is reversed then the above terms will be badly-formed:

```
?- [user].
|: :- op(400,fx,succ).
|: :- op(300,xfy,plus).
|:

?- display(succ x plus succ y).
ERROR: Syntax error: Operator priority clash
ERROR: display(succ x plus
ERROR: ** here **
ERROR:  succ y) .

?- display(succ (x plus (succ y))).
succ(plus(x, succ(y)))
```

Note that, as usual, the problem can be solved by the insertion of parentheses.

3.6 Finally, Output (and Input Too)

The built-in predicate `write(X)` prints out the argument to the terminal, while `read(X)` reads a value in from the terminal, binding `X` to it (the input must be terminated by a full-stop); for example:

```
write('Hello World').
read(Z), Y is Z, X is 2*Y,
   write('Z is ', Z, ', X is ', X), nl.
```

3.7 Exercises: Getting Started with Prolog

The following exercises should help you gain basic familiarisation with the Prolog system. You may also wish to experiment further with the programs given in the main text:

1. Enter the following facts directly into the Prolog system:

```
parent(sue,toby) .
parent(sue,jules) .
```

2. Turn on the `trace` and try the following goals, typing ';' after each one to look for possible alternative solutions:

```
parent(sue,toby).
parent(X,toby).
parent(sue,X).
parent(Y,X).
parent(alan,Z).
```

3. Write the following program to a text file and load it into the Prolog system:

```
female(X) :- mother(X) .
female(X) :- sisterof(X,Y) .
mother(sue) .
sisterof(harriet, toby) .
father(john) .
```

4. Try the following goals, again typing ';' after each one to look for possible alternative solutions:

```
female(sue).
female(john).
female(harriet).
female(toby).
```

5. Write the following program to a text file:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
parent(sue,georgia) .
parent(sue,harriet) .
parent(frances,sue) .
```

   Load the file into the Prolog system.

6. Try the following goals, again typing ';' after each one to look for possible alternative solutions:

```
parent(sue,X).
ancestor(sue,george).
ancestor(george,sue).
ancestor(frances,george).
```


7. Now try the goal `ancestor(frances,X)`

8. Change the order of the rules and facts in the program and observe the result.

9. Write a program `inclist(X,Y)` which adds 1 to each element in the integer list `X` giving the result in `Y`. For example, the goal `inclist([1,2,3], Z)` would give `Z = [2,3,4]`.

10. **Trace Results**: Use `trace` to observe and record the execution of the `ancestor` program from Question 5/6, in particular to generate multiple solutions.

11. **Evaluation of Logical Formulae**: Write a program to evaluate propositional logical formulae. You will need to declare your own logical operators. For example, you can define the conjunction operator '/\' by;

```
:- op(400,yfx,/\).
```

    In particular you need to decide how to represent atomic propositions and assignment of truth values to them.

    Then test your program on a variety of logical formulae. Include at least negation, conjunction, disjunction, implication and bi-implication. For atomic propositional letters, you will need to declare whether they are true or not. For example if `p` is true and `q` is not true then the goal

```
:- eval((p /\ q) => p) .
```

    could be refuted.

12. **Example Evaluations**: Use `trace` to observe the execution of the program from Question 11.

    Put your program in a file called `proplog-eval.pl` and your test results in a file called `proplog-eval-results`.

4 Classical Propositional Resolution

Resolution is another method for testing satisfiability. It is based on the following property of classical logic: $\{F \vee A, G \vee \neg A\} \models F \vee G$.

4.1 Normal Forms and Clause Form

Recall that a **literal** is an atomic formula (e.g. A) or the negation of an atomic formula (e.g. $\neg A$). We will use L (possibly subscripted) to denote literals.

Recall that if L is a literal then $\overline{L}$ is the **complement** of L, defined by: if $L = A$ then $\overline{L} = \neg A$ and if $L = \neg A$ then $\overline{L} = A$. Note that $\overline{L}$ is also a literal. The literals L and $\overline{L}$ form a **complementary pair** for any literal L.

A formula is a **conjunction** if it is of the form $F1 \wedge F2 \wedge \cdots \wedge Fn$ where each Fi is an arbitrary formula, $1 \leq i \leq n$. Each Fi is a **conjunct** (note that this says nothing about the structure of Fi itself).

A formula is a **disjunction** if it is of the form $F_1 \vee F_2 \vee \cdots \vee F_m$ where each F_j is an arbitrary formula, $1 \leq j \leq m$. Each F_j is a **disjunct** (note that this says nothing about the structure of F_j itself).

A formula is in **conjunctive normal form** (CNF) if it is a conjunction of disjunctions of literals. That is, if it is of the form $F_1 \wedge F_2 \wedge \cdots \wedge F_n$ where each conjunct F_i is of the form $L_1 \vee L_2 \vee \cdots \vee L_m$ and each L_j is a literal.

A formula is in **disjunctive normal form** (DNF) if it is a disjunction of conjunctions of literals. That is, if it is of the form $D_1 \vee D_2 \vee \cdots \vee D_m$ where each disjunct D_i is of the form $L_1 \wedge L_2 \wedge \cdots \wedge L_n$.

Theorem 11 *Every formula F of classical propositional logic has a logically equivalent formula F' where F' is in CNF.*

The procedure given below achieves the conversion and also acts as a proof since in each step we merely replace a formula by a logically equivalent one.

1. eliminate all connectives except for $\neg$, $\wedge$ and $\vee$ by using the logical equivalents in terms of $\neg$, $\wedge$ and $\vee$; i.e. $(F \to G) =\!|\!\models (\neg F \vee G)$;
2. move negation signs as far in as possible by using the laws $\neg(F \wedge G) =\!|\!\models (\neg F \vee \neg G)$ and $\neg(F \vee G) =\!|\!\models (\neg F \wedge \neg G)$;
3. replace all $\neg\neg F$ by F;
4. eliminate any conjunction inside a disjunction by the distributivity laws $F \vee (G \wedge C) =\!|\!\models (F \vee G) \wedge (F \vee C)$ and $(F \wedge G) \vee C =\!|\!\models (F \vee C) \wedge (G \vee C)$.

Because of the particular structure of formulae in CNF we can use a set notation for them as follows:

A **clausal formula** is a disjunction of literals $L_1 \vee L_2 \vee \cdots \vee L_n$. The logical equivalences $F \vee F =\!|\!\models F$ and $(F \vee G) =\!|\!\models (G \vee F)$ and $F \vee (G \vee H) =\!|\!\models (F \vee G) \vee H$ allow us to reorder the literals and to eliminate duplicates in a clausal formula without changing its semantic behaviour. Hence the only important property of a clausal formula is whether any particular literal is present in it or not. So we can represent clausal formulae by *sets* of literals.

A **clause**, then, is a set of literals. We use the notation $[L_1; L_2; \ldots; L_n]$ to denote the clause containing the literals $L_1, L_2, \ldots, L_n$ since we wish to discuss sets of clauses and it is less confusing (hopefully!) if we distinguish between the different levels of set membership. The idea is to remember that ";" stands for $\vee$ and to use "[" and "]" as set delimiters instead of the braces "{" and "}".

A clausal formula can be represented by a clause, but the same clause stands for many syntactically different formulae. However all these formulae are logically equivalent. For example $[A; \neg B]$ stands for $A \vee \neg B$, $\neg B \vee A$, $\neg B \vee A \vee \neg B$ and so on. We can extend the notion of an interpretation satisfying (sets of) formulae to that of satisfying (sets of) clauses in the obvious way. So an interpretation v satisfies a (non-empty) clause C (i.e. $v(C) = \mathbf{T}$) if and only if $v(L) = \mathbf{T}$ for some literal $L \in C$. It is a useful convention to extend this definition to the empty clause, so that no interpretation satisfies $[\,]$ (there is no element which v could satisfy). In other words, $[\,]$ is a contradiction.

A clause is a **unit clause** if it consists of a single literal e.g. $[\neg A]$.

Using the corresponding logical equivalences for $\wedge$ to the ones we used for $\vee$, any formula in CNF can be written as a set of clauses $\{C_1, C_2, \cdots, C_m\}$. The idea is to remember that "," stands for $\wedge$.

A formula in CNF when written in this way is in **clausal form**. Notice that the clausal form of a CNF formula is not, strictly speaking, a formula. For example the formula $(A \vee B) \wedge \neg A \wedge \neg B$ can be written in the clausal form $\{[A; B], [\neg A], [\neg B]\}$. You should convince yourself that, while no interpretation satisfies $\{[\,]\}$, every interpretation satisfies $\{\}$.

Given a formula F, we can convert F into CNF using the algorithm described above, and write the resulting formula in clausal form. The resulting set of clauses is called *the* clausal form of F. Although there may be many different sets of clauses which are logically equivalent to F we use the expression "the clausal form of F" to mean the one which is obtained by this particular syntactic method.

A pair of clauses C_1 and C_2 are **clashing clauses** if $L \in C_1$ and $\overline{L} \in C_2$ for some literal L. They **clash** on the complementary pair of literals L and $\overline{L}$. Their **resolvent** is the clause obtained by deleting L from C_1 to give C_1', deleting $\overline{L}$ from C_2 to give C_2' and then forming the union of C_1' and C_2'. Note that this **resolvent** $Res(C_1, C_2)$ is also a clause. The clauses C_1 and C_2 are known as the **parent clauses** of $Res(C_1, C_2)$. Actually this notation is a little sloppy, since C_1 and C_2 may clash on more than one literal, so they may have more than one resolvent. It doesn't really matter, though, since in that case all the resolvents of C_1 and C_2 are tautologies.

Theorem 12 (Resolution rule for propositional logic) $C_1, C_2 \models Res(C_1, C_2)$.

In fact, the following curious result is true.

Theorem 13 *The resolvent $Res(C_1,C_2)$ is satisfiable iff the parent clauses C_1 and C_2 are jointly satisfiable.*

The "if" part of this theorem follows immediately from theorem 12, but the "only if" part is a bit more subtle. It is a good exercise to try and prove this theorem, though the result is not very important.

Given a set of clauses N_0 the procedure given below is called the **resolution procedure**.

1. beginning with $i = 0$ repeat
2. assume that we have formed N_i;
3. choose a pair of clashing clauses C_1 and C_2 from N_i;
4. let $C = Res(C_1,C_2)$;
5. if $C = []$ then terminate (N_0 is not satisfiable);
6. else add C to N_i to give N_{i+1};
7. until $N_i = N_{i+1}$ for all ways of choosing C_1 and C_2
8. terminate (N_0 is satisfiable).

The following paragraphs describe some improvements which can be made to the efficiency of the resolution method.

Let N and N' be two sets of clauses. We write $N \approx N'$ to denote that N is satisfiable iff N' is satisfiable. Note the "iff". Also note that $\approx$ is not the same as logical equivalence $=\!|\!\models$. The difference is that $F =\!|\!\models G$ means that an interpretation v that satisfies F must *itself* satisfy G (and vice-versa); whereas $N \approx N'$ means that if some interpretation v satisfies N then there exists *some* (possibly different) interpretation that satisfies N' (and vice-versa).

We will now consider some simplifications that transform a set of clauses N into a simpler set of clauses N'. The simplifications preserve satisfiability; that is $N \approx N'$.

Lemma 1 (Purity Deletion) *Suppose a literal L appears in N but its complement $\overline{L}$ does not appear in N. Delete all the clauses that contain L from N and call the remaining set of clauses N'. Then $N \approx N'$.*

Lemma 2 (Unit Propagation) *Suppose N contains a unit clause $\{L\}$; that is $N = \{C_1,C_2,\cdots,L,\cdots,C_n\}$. Delete all the clauses that contain L and delete $\overline{L}$ from each remaining clause. Call the resulting set of clauses N'. Then $N \approx N'$.*

This lemma can be used to reduce the size of the clause set at any stage of the resolution process.

Lemma 3 (Tautology Deletion) *If N contains a clause C and both L and $\overline{L}$ are members of C then delete C from N to give N'. Then $N \approx N'$ (in fact $N =\!|\!\models N'$, since $\models C$).*

We can use this lemma to eliminate such clauses from the initial clause set and to avoid adding any resolvents which have this form (see the remark above about clause pairs with more than one resolvent).

A clause C_1 **subsumes** another clause C_2 if $C_1 \subseteq C_2$; that is if all the literals from C_1 also appear in C_2.

Lemma 4 (Subsumption Deletion) *If N contains two clauses C_1 and C_2, and C_1 subsumes C_2 then delete C_2 (the bigger clause) from N to give N'. Then $N \approx N'$. (In fact $N =\!|\!\models N'$, since $C_1 \models C_2$.*

We can use this rule on the initial set of clauses as above. If a resolvent is subsumed by clause already in the set, then there is no need to add it. And if a resolvent subsumes any existing clauses, we can use the rule to remove them from the set.

4.2 Soundness and Completeness of Propositional Resolution

Let us write $CF(Y)$ to denote the clausal form of a set of formulae Y — that is, the set union of the clausal forms of the elements of Y. Now, if we put $N_0 = CF(X \cup \{\neg F\})$ in the resolution procedure given above then we can use the resolution procedure to decide if $X \cup \{\neg F\}$ is satisfiable. That is, we can test if $X \models F$.

There is a **refutation** of a set of clauses N_0 iff the resolution procedure derives the empty clause $[]$ (step 5) from N_0. Let us write $X \vdash_{\mathcal{R}} F$ iff there is a refutation of $N_0 = CF(X \cup \neg F)$.

Soundness of $\mathcal{R}$ w.r.t. the semantics of PC: If $X \vdash_{\mathcal{R}} F$ then $X \models F$. That is, if there is a refutation of $CF(X \cup \{\neg F\})$ then F is a logical consequence of X.

Completeness of $\mathcal{R}$ w.r.t. the semantics of PC: If $X \models F$ then $X \vdash_{\mathcal{R}} F$. That is, if F is a logical consequence of X then there is a refutation of $CF(X \cup \{\neg F\})$.

4.3 Resolution as a decision procedure

The resolution method provides us with (yet!) another decision procedure for propositional logic. That is, an algorithm which can tell us in a finite number of steps whether any logical implication $X \models F$ holds. Since resolution is (sound and) complete, we only have to check that the procedure always terminates.

Termination: Given a finite set of clauses N_0 there are only a finite number of different literals to work with. Thus there are a finite number of (new) clauses that can be formed during the resolution procedure. (Remember that clauses are sets so they do not contain duplicates.) Sooner or later we must exhaust this supply of clauses. If we find the empty clause in this process then N_0 is unsatisfiable else N_0 is satisfiable.

4.4 Different forms of resolution

Various strategies are used to reduce the work involved in searching for a pair of clashing clauses in a resolution theorem proving system (remember we may be dealing with hundreds or thousands of clauses). This is done by restricting the application of the resolution rule to certain sets of clauses.

Linear resolution: We start with some initial set of clauses N_0. An element of N_0 is designated as the *centre* clause. From now on, each resolvent becomes the new centre clause and each resolution step must involve the current centre clause. This means that we only have to search one set of clauses at each step. Linear resolution is (sound and) complete with respect to the semantics of classical logic. That is, if N_0 is unsatisfiable, there is a refutation of N_0 by linear resolution. However, there is no “magic formula” for choosing an appropriate centre clause to start with of for choosing the right clause to resolve with at each stage. So if we get stuck in our proof by deriving a clause which does not clash with any other clause, we may have to back-track to the last place where we made a choice and make a different one. We will see more of linear resolution and back-tracking when we look at logic programming.

Input resolution: We start with some initial set of clauses N_0. The resolution procedure generates new clauses as it proceeds. Input resolution demands that one of the clashing literals C_1 or C_2 is always chosen from N_0. Input resolution is (sound) but is not complete with respect to the semantics of classical logic.

Set of support resolution: Suppose N is a set of clauses and suppose that T is a subset of N such that we know that $N \setminus T$ is satisfiable. The resolution procedure is restricted so that both clashing clauses may not come from $N \setminus T$. The set elements of T (including any new resolvents, which must be added to T) are said to be **supported**. The intuition behind this rule is that if we want to test $X \models F$ then X is probably satisfiable; hence resolving on two clauses from $CF(X)$ will not yield the empty clause. So if $N = CF(X \cup \neg F)$ and $T = CF(\neg F)$ then $N \setminus T = CF(X)$ is satisfiable. Set of support is (sound and) complete with respect to the semantics of classical logic.

4.5 Exercises

Use propositional resolution to prove the following:

1. $A, (A \to B) \models B$
2. $A \to (B \to B') =\!\mid\models (A \wedge B) \to B'$
3. $\models (((A \wedge B) \wedge B') \wedge D) \vee (E \to E)$
4. $(A \to (B_1 \to B_2)), (\neg(A \wedge B_1) \to B_2) \models B_2$
5. During resolution, why can you use the ‘approximates’ relation ($S \approx S'$) to simplify the sets of clauses?

5 Propositional Tableau Systems

We have seen that the method of truth tables is a *decision procedure* for determining whether a propositional formula F is a logical consequence of a set of formulae X. That is, it always terminates (eventually!) and gives a definite answer to all such questions. However, for problems involving a large number of propositions the amount of calculation required is prohibitive. The method essentially consists of searching for an interpretation which satisfies—that is, makes them evaluate to **T**—all the formulae in $X \cup \{\neg F\}$. If the search is successful then $X \not\models F$, otherwise $X \models F$. The *tableau method* which we describe below is a more systematic procedure for searching for a model—a satisfying interpretation—for a set of formulae.

5.1 Notation

In the notes which follow $\mathbf{PC}$ stands for propositional calculus, which is another name for the propositional logic defined in §1. In view of the logical equivalences $A \vee (B \vee C) \mathrel{=}\!\mid\!\models (A \vee B) \vee C$ and $A \wedge (B \wedge C) \mathrel{=}\!\mid\!\models (A \wedge B) \wedge C$ we can afford to be a little careless in parenthesizing such formulae. The notation $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ will be used as an abbreviation for $A_1 \wedge (A_2 \wedge (\ldots \wedge A_n))$. The tableau rules described below often involve selecting one or two elements from a set. It is convenient to use expressions such as X, F, G to denote $X \cup \{F, G\}$ and so on.

5.2 Syntax of tableau systems

Our tableau systems work with *finite* sets of formulae, we use the following notational conventions:

X, Y, Z denote finite sets of (well formed) formulae;

if $X = \{F_1, \cdots, F_n\}$ then $\neg X = \{\neg F_1, \cdots \neg F_n\}$;

A tableau system is a collection of tableau (inference) rules. A tableau rule consists of a **numerator** above the line and a list of **denominators** (below the line):

$$\frac{\mathcal{N}}{\mathcal{D}_1 \mid \mathcal{D}_2 \mid \cdots \mid \mathcal{D}_k} \quad \text{general form of a tableau rule}$$

The denominators $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_k$ are separated by vertical bars. The numerator $\mathcal{N}$ is a finite set of formulae and so is each denominator $\mathcal{D}_i$.

Figure 1 (page 25) shows a common tableau system[2] for $\mathbf{PC}$.

A **tableau** for X is a finite tree T whose nodes are labelled with formulae, or 'Closed'. T has initial path whose nodes are labelled $F_1 \ldots F_n$. The tableau is then constructed by *repeatedly* using the tableau rules as follows:

[2] Note: the rules have been labelled to be consistent with [1], but do not include rules (4) and (9) for bi-implication.

$$(\wedge)\ \frac{X, P \wedge Q}{X, P, Q} \qquad (\vee)\ \frac{X, P \vee Q}{X, P \mid X, Q}$$

$$(\rightarrow)\ \frac{X, P \rightarrow Q}{X, \neg P \mid X, Q} \qquad (\neg)\ \frac{X, \neg\neg P}{X, P}$$

$$(\neg\wedge)\ \frac{X, \neg(P \wedge Q)}{X, \neg P \mid X, \neg Q} \qquad (\neg\vee)\ \frac{X, \neg(P \vee Q)}{X, \neg P, \neg Q}$$

$$(\neg\rightarrow)\ \frac{X, \neg(P \rightarrow Q)}{X, P, \neg Q} \qquad (\mathbf{f})\ \frac{X, p, \neg p}{\mathbf{f}}$$

Figure 1: Tableau rules for $\mathcal{CPC}$

1. choose a leaf node n in the current tree and another node n' on the path from the root of the tree to n, where n' is labelled with formula F;
2. try to apply Rule Closed to F, i.e. if F is some literal L and a complementary literal $\overline{L}$ labels some node on the path, then apply the rule to n
3. otherwise, choose a tableau rule with (non-literal) numerator F and denominators $\mathcal{F}_1, \cdots, \mathcal{F}_2$;
4. create k children nodes $n_1, \cdots n_k$ for n and label node n_i. with formula $\mathcal{F}_i$.

   (note: k will either be 1 or 2!)

A branch of a tableau is **closed** if its leaf node is labelled Closed; otherwise it is **open**.

A tableau is **closed** if all its branches are closed; otherwise it is **open**.

A branch is **completed** if no tableau rule is applicable to any node on the path from the root to its leaf node. A tableau is **completed** if all its branches are completed.

We write $X \vdash_{\mathcal{T}} F$ iff there is a closed tableau for the set $X \cup \neg F$. We write $X \not\vdash_{\mathcal{T}} F$ iff there is no closed tableau for the set $X \cup \neg F$.

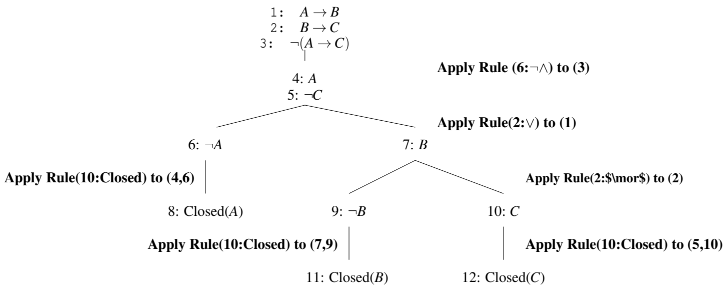
5.3 Example

Semantic tableaux are usually presented as trees, where child nodes are produced from a parent node by application of a tableau rule to some formula lying on the path from root to parent.

For example, consider buidling the semantic tableau for the set of formulae:

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

This would produce the following:



This tableau is *closed* since all branches are closed. this means the original formulae (1), (2), (3) are *unsatisfiable*.

It is important to note that new nodes can *only* be introduced into the tree by application of one of the given semantic tableaux rules to an existing formula in the tree. It is *not* possible to apply arbitrary logical equivalences, for example adding the formula $\neg A \vee B$ to the tree, as a logical equivalence to $A \to B$.

Finally, do not confuse semantic tableaux with the syntax trees shown in Figure 1.1 — although they both appear as trees, they are concerned with *completely* different topics!

5.4 Soundness and Completeness of Propositional Tableaux

Let us write $X \vdash_{\mathcal{T}} F$ (pronounced "X $\mathcal{T}$-deduces F") iff there exists a closed tableau for $X \cup \neg F$. Thus $\vdash_{\mathcal{T}}$ is a purely syntactic deductibility relationship between a set of formulae X and a single formula F.

On the other hand the tableau procedure was designed to test $X \models F$ using the semantics of $\mathbf{PC}$. How can we be sure that it does this ? We could be sure if we could guarantee that $X \vdash_{\mathcal{T}} F$ iff $X \models F$. This is known as the **soundness and completeness of (a syntactic deduction system) $\mathcal{T}$ with respect to the semantics of $\mathbf{PC}$**.

Soundness: $\mathcal{T}$ is sound (with respect to the semantics of $\mathbf{PC}$) if $X \vdash_{\mathcal{T}} F$ implies $X \models F$. That is, if there is a closed tableau for $X \cup \neg F$ then $X \models F$.

Completeness: $\mathcal{T}$ is complete (with respect to the semantics of $\mathbf{PC}$) if $X \models F$ implies $X \vdash_{\mathcal{T}} F$. That is, if F is a logical consequence of X then there is a closed tableau for $X \cup \neg F$. Instead we usually prove that if $X \not\vdash_{\mathcal{T}} F$ then $X \not\models F$. That is, if there is no closed tableau for $X \cup \neg F$ then there exists an interpretation that satisfies X but falsifies F.

5.5 Tableaux as a decision procedure

The tableau procedure we have outlined forms a decision procedure for $\mathbf{PC}$. There are three things we need to show to conclude this:

1 termination: the procedure must terminate for all inputs;

2 soundness: it gives only correct answers;

3 completeness: it gives all (correct) answers.

In every rule in Figure 1 the formula shown explicitly in the numerator is known as the **main formula**. For example, the main formula of the $(\wedge)$ rule is $(F \wedge G)$.

Termination: Every tableau rule in Figure 1 strictly simplifies the main formula into simpler formulae. So a sequence of tableau rule applications will eventually result in a set of literals. Now, either (Closed) is applicable, in which case the branch closes, or it is not, in which case the branch remains open. Either way the branch cannot be extended further. Since this applies to all branches, the procedure terminates (for all finite sets).

Suppose we are asked to decide if $X \models F$. Construct a completed tableau for $X \cup \neg F$. Gy the above argument any such attempt must terminate with either a closed tableau or an open tableau. If the tableau is closed then the soundness of $\mathcal{T}$ tells us that $X \models F$; else (it must be open) the completeness of $\mathcal{T}$ tells us that $X \not\models F$. Hence $\mathcal{T}$ is a decision procedure for deciding logical consequence in $\mathbf{PC}$.

5.6 The efficiency of tableaux

If X contains more than one non-literal formula then we have a choice of which tableau rule to apply. Does the order of rule applications make a difference? Suppose we are given the set $\{A \wedge B, \neg A, A_1 \wedge A_2 \wedge A_3\}$. Clearly a tableau that attacks $A \wedge B$ first will close immediately. But a tableau that attacks $A_1 \wedge A_2 \wedge A_3$ will take longer to close. Thus, the order of rule applications is very important. But there are no theoretical results to indicate a perfect strategy that will work for any given set X.

In many text books (e.g. Ben-Ari) it is claimed that tableaux are *clearly* better than truth tables since truth tables must explore all 2^n interpretations for a given formula built from the atomic formula $A_1, A_2, \cdots A_n$ whereas a tableau will only explore some of these. This is wrong! The point is that the complexity of truth tables depends on the *number of atomic formulae* that are used to make up the formulae in X whereas the complexity of tableaux depends on the *structure of the formulae* in X.

In most cases, tableaux will be faster than truth tables. But there exist formulae which are bad for tableau but good for truth tables. One such example is the formula $F = ((A \vee B_1) \wedge (A \vee \neg B_1) \wedge (\neg A \vee B_2) \wedge (\neg A \vee \neg B_2))$.

5.7 Extracting Counter-models or Proofs from Tableaux

The tableau rules can be given an even stronger reading:

an interpretation v satisfies a denominator iff v satisfies the numerator.

Hence, all the sets in an open branch are satisfied by the *same* interpretation v that satisfies the open leaf of the branch; that is the interpretation

$$v(A_i) = \mathbf{T} \text{ if } l_i = A_i \text{ and } v(A_i) = \mathbf{F} \text{ if } l_i = \neg A_i$$

which satisfies every literal in the open leaf. Thus an open tableau (branch) for $X \cup \neg F$ gives us an actual interpretation v that satisfies X and falsifies F. Such an interpretation is called a **counter-model** for $X \models F$.

On the other hand, a closed tableau for $X \cup \neg F$ gives us a proof that $X \models F$ since can reason from the root down to each (closed) leaf using the usual downward reading of the tableau rules.

5.8 A Modification to the basic tableau calculus

The rule (Closed) is only applicable to literals A and $\neg A$ thus forcing every tableau to be completed. It can be replaced by the more liberal rule

$$(\mathbf{f}')\ \frac{X, P, \neg P}{\mathbf{f}}$$

where F (and hence $\neg F$) is now an arbitrary formula instead of an atomic formula. The new rule allows us to close branches earlier while retaining all properties of the tableau we have seen.

5.9 Exercises

Use semantic tableaux to prove the following:

1. $A, (A \to B) \models B$:
2. $A, (A \to B_1), (B_1 \to B_2) \models B_2$
3. $A \leftrightarrow B = \mid \models (A \wedge B) \vee (\neg A \wedge \neg B)$
4. Construct a valid propositional formula that is 'inefficient' to check using semantic tableaux.
5. Compare the relations logical deduction ($\vdash_{\mathcal{S}}$, with respect to a deduction system $\mathcal{S}$) and semantic entailment/logic consequence ($\models$)?
6. What is the difference between logical deduction ($\vdash$) and logical implication ($\to$)?
7. Why should a reasoning system be sound?

6 First Order Predicate Logic

The major disadvantage of classical propositional logic is a lack of expressive power. The atomic formulae of propositional logic are "too atomic"—they are just statements which may be true or false but which have no internal structure. There is no way to relate one atomic formula to another. In first-order logic the atomic formulae are interpreted as statements about relationships between objects.

For example, suppose we want to talk about people and we want to say "Sanjay is male" and "Rohini is female". In propositional logic we would have to say something like "Sanjay-is-male" and "Rohini-is-female".

Another way is to invent higher level entities, similar to propositions, which can take a person's name as an argument. For example we might write "*male*(*Sanjay*)" and "*female*(*Rohini*)" where "*male*" and "*female*" are these higher level entities and "*Sanjay*" and "*Rohini*" are arguments. Such higher level propositions are called *predicates*. We might also want a predicate which takes two arguments to express the statement that Sanjay and Rohini are brother and sister, e.g. "*siblings*(*Sanjay*, *Rohini*)". The names *Sanjay* and *Rohini*, which are used to refer to specific objects in our domain of study are called *constants*.

Now consider that we may want to say something like "everybody is male or female". One way is to allow predicates to take variable arguments. For example we might say "*male*(x)" and "*female*(y)". But we now need a device to say "for all values of the argument x". One way is by introducing a symbol called a *quantifier*: $\forall x$. Thus our statement becomes $\forall x(\mathit{male}(x) \vee \mathit{female}(x))$. Note that there is no way to express such a statement in propositional logic.

Now, predicates are supposed to take truth values of true or false but we can also allow another class of objects that return values of the underlying domain; in this case the domain of people. For example we might say "*father*(*Rohini*)" to mean the person who is (or was) the father of Rohini. Such objects are known as *functions*. We can then make such statements as $\forall x(\mathit{male}(\mathit{father}(x)))$.

The introduction of predicates, variables, constants, functions and quantifiers gives predicate logic. Note that we need different sets of function, constant, and predicate symbols for each application area. The ones given above are useful for talking about family relationships between people but not for talking about arithmetic, say.

6.1 Syntax of first-order predicate logic

A **language** $\mathcal{L}$ for predicate logic is a collection of predicate symbols, function symbols and constant symbols. Each predicate symbol must be assigned a non-negative integer called its **arity** (that is, its number of arguments). A predicate whose arity is n is called n-ary. Similarly, each function symbol in $\mathcal{L}$ must be assigned a positive arity (the role of 0-ary functions is played by constants).

Whatever language we are working in, we always have available the set $\mathcal{X} = \{x, y, z, \cdots\}$ of variable symbols.

A **term** of $\mathcal{L}$ is defined by:

any constant a in $\mathcal{L}$ is a term;

any variable $x \in \mathcal{X}$ is a term;

if $t_1, t_2, \cdots, t_n$ are terms and f is an n-ary function symbol in $\mathcal{L}$ then $f(t_1, t_2, \cdots, t_n)$ is a term.

So terms are built by repeatedly applying functions to variables and constants. A term which does not contain any variables is called a **ground term**.

An **atomic formula** of $\mathcal{L}$ is defined by:

t and **f** are atomic formulae;

if $t_1, t_2, \cdots, t_n$ are terms and $p \in \mathcal{P}$ is an n-ary predicate symbol then $p(t_1, t_2, \cdots, t_n)$ is an atomic formula where $n \geq 1$. If p is a 0-ary predicate symbol, then p() is an atomic formula.

Notice that a function symbol can appear inside another term but a predicate symbol cannot appear inside a term or inside an atomic formula.

The following are terms of $\mathcal{L}$ where $\mathcal{L}$ has constants a and b, a unary (1-ary) function f and binary (2-ary) functions g and h: a; y; $f(a)$; $f(f(a))$ and $g(b, h(a, x))$.

If p is a unary and q a binary predicate, the following are atomic formulae: $p(a)$; $p(f(a))$; $q(g(a, b), f(f(x)))$.

A **formula** of $\mathcal{L}$ is now defined by:

any atomic formula is a formula;

if F and G are formulae and x is a variable then the following are formulae: $(F \wedge G)$; $(F \vee G)$; $(\neg F)$; $(\neg F)$; $(F \rightarrow G)$; $\forall\, x\, F$; $\exists\, x\, F$.

The symbol $\forall$ is called the *universal quantifier* and the symbol $\exists$ is called the *existential quantifier*. Read $(\forall x\, F)$ as "for all x, F" and $(\exists\, x\, F)$ as "there exists an x such that f". The precedence of the quantifiers is the same as that of negation.

If a variable x appears as part of a quantified subformula as in $(\forall\, x\, F)$ or $(\exists\, x\, F)$ then it is said to be a **bound variable**. A variable that is not bound is a **free variable**. Note that a variable may appear both free and bound in a formula; for example, $p(x) \wedge (\forall x q(x))$. The following more formal definition therefore considers each occurrence of a variable.

The **free variable occurrences** in a formula are defined as follows:

1. the free variable occurrences in an atomic formula are all the variable occurrences in that formula;
2. the free variable occurrences in $\neg F$ are the free variable occurrences in F;
3. the free variable occurrences in $F \vee G$ are the free variable occurrences in F together with the free variable occurrences in G;

4. the free variable occurrences in $F \wedge G$ are the free variable occurrences in F together with the free variable occurrences in G;
5. the free variable occurrences in $F \rightarrow G$ are the free variable occurrences in F together with the free variable occurrences in G;
6. the free variable occurrences in $\forall xF$ and $\exists xF$ are the free variable occurrences in F except for occurrences of x.

If a variable occurrence is not free then it is said to be **bound**.

The following are formulae: $(\forall x \exists y p(x,y))$; $\exists y(p(a,y) \rightarrow q(b,x))$. The first contains only bound variable occurrences whereas the second contains a free variable occurrence x.

Instead of $\forall x(\forall y(F))$ we usually write $(\forall x, y\ F)$. A formula with no free variable occurrences is said to be **closed**. A formula which is not closed is said to be **open**. If the free variable occurrences of a formula F are $\{x_1, x_2, \cdots, x_k\}$ then we write $F(x_1, x_2, \cdots, x_k)$ to denote this. Then $(\forall x_1, x_2, \cdots, x_k F)$ is the **universal closure** of F and $(\exists x_1, x_2, \cdots, x_k F)$ is the **existential closure** of F.

If F is a formula and t is a term then $F[x \leftarrow t]$ is obtained by replacing every *free* occurrence of x in F by the term t. Thus $(\forall x\ p(x))[x \leftarrow a] = (\forall x\ p(x))$ since x is not free in $(\forall x\ p(x))$. If we are given a quantified formula $\forall x\ F(x)$ then $F[x \leftarrow t]$ is denoted by $F(t)$ and is called an **instance** of $\forall x\ F(x)$. For example, $p(a)$ is an instance of $\forall x\ p(x)$.

Notice that the use of $F(x)$ is not the same as $p(x)$. The former is just notation to allow us to say that x is free in F whereas the latter is a formula of predicate logic with x as an argument to the unary predicate symbol p.

6.2 Domains, Relations and Functions

In propositional logic we can assign a truth value to a formula once we know the truth values of its propositions. The semantics of predicate logic are more complicated. We want to interpret the function symbols as functions and predicate symbols as relations over some set D of objects. We review the necessary definitions in this section.

A **domain** D is just a non-empty set. When we want to speak of elements of this domain we shall use $\{d_1, d_2, \cdots\}$.

An n-ary **relation** R on a domain D is a set of tuples of the form $(d_1, d_2, \cdots, d_n)$ where each d_i is from D and there are exactly n elements in each tuple. For example, a binary relation is a collection of pairs each of the form (d_1, d_2). We can obtain a predicate $R'(\cdots)$ which takes n arguments from a relation by defining:

$$R'(d_1, d_2, \cdots, d_n) = \mathbf{T} \text{ iff } (d_1, d_2, \cdots, d_n) \in R.$$

By convention, a 0-ary relation R on D is just a truth value. To justify this, there is only one 0-tuple $()$ over D which is either in the set or not, so we can interpret R as **T** if $() \in R$ and **F** otherwise. If you don't like the justification then just accept the convention!

For example, let SQ be the set of all binary tuples over the natural numbers where the first element is a number and the second is its square. Thus:

$$SQ = \{(x,y) \mid y = x^2, x \in \mathcal{N}, y \in \mathcal{N}\}$$

Then, we can define a predicate from SQ as: $SQ(d_1, d_2)$ iff $(d_1, d_2) \in SQ$.

An **n-ary function** F on domain D is a mapping from tuples of the form $(d_1, d_2, \cdots, d_n)$ to elements of the domain. Thus $F(d_1, d_2, \cdots, d_n)$ is an element of D.

6.3 Semantics of first-order predicate logic

In order to assign a truth value to a predicate logic formula for some language $\mathcal{L}$, we must interpret the predicate, function and constant symbols of $\mathcal{L}$ as relations and functions over some domain D, and then see if the formula is true in this interpretation. Such an interpretation is called a structure for $\mathcal{L}$. If the formula F contains free variables, we must also assign a value in D to each free variable of F. The reason for insisting that D is non-empty is that we would like a formula such as $\exists x(p(x) \vee \neg p(x))$ to be valid, but this would not be the case if we allowed structures where D was empty.

Let $\mathcal{L}$ be a language for predicate logic. A **structure** for $\mathcal{L}$ is a quadruple $\mathcal{M} = (D, R, F, C)$ where:

- D is a (non-empty) domain;
- R is an assignment of a k-ary relation $p^{\mathcal{M}}$ on D to each k-ary predicate symbol p of $\mathcal{L}$;
- F is an assignment of a k-ary function $f^{\mathcal{M}}$ on D to each k-ary function symbol f of $\mathcal{L}$;
- C is an assignment of an element $a^{\mathcal{M}}$ from D to each constant symbol a of $\mathcal{L}$.

If $\mathcal{M} = (D, R, F, C)$ is a structure for $\mathcal{L}$, an **assignment** s over $\mathcal{M}$ is a choice of value $s(x) \in D$ for each variable $v \in \mathcal{X}$. If s is an assignment, x is a variable and d is an element of D, let $s[x \mapsto d]$ be the assignment which agrees with s except that it gives the value d to x.

Given an assignment s over $\mathcal{M}$, each term t can be given a value $t^{\mathcal{M},s} \in D$ as follows:

- $a^{\mathcal{M},s} = a^{\mathcal{M}}$ if a is a constant;
- $x^{\mathcal{M},s} = s(x)$ if x is a variable.
- $f(t_1, t_2, \ldots, t_n)^{\mathcal{M},s} = f^{\mathcal{M}}(t_1^{\mathcal{M},s}, t_2^{\mathcal{M},s}, \ldots, t_n^{\mathcal{M},s})$ if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms.

Note that the values are elements of D since each function $f^{\mathcal{M}}$ takes values in D.

We can then give a truth value $v_{\mathcal{M},s}(F) \in \{\mathbf{T}, \mathbf{F}\}$ to each formula F as follows:

- $v_{\mathcal{M},s}(\mathbf{t}) = \mathbf{T}$;
- $v_{\mathcal{M},s}(\mathbf{f}) = \mathbf{F}$;

- $v_{\mathcal{M},s}(p(t_1,\ldots,t_n)) = T$ iff $(t_1^{\mathcal{M},s},\ldots,t_n^{\mathcal{M},s}) \in p^{\mathcal{M}}$, where p is an n-ary predicate symbol. If $n = 0$ then $v_{\mathcal{M},s}(p())$ is just the truth value $p^{\mathcal{M}}$ (see the remarks above concerning 0-ary relations);
- $v_{\mathcal{M},s}(\neg F) =$ **T** iff $v_{\mathcal{M},s}(F) =$ **F** and similarly for the other connectives $\wedge$, $\vee$ and $\rightarrow$;
- $v_{\mathcal{M},s}(\forall x\, F) =$ **T** iff for all $d \in D$, $v_{\mathcal{M},s[x\mapsto d]}(F) =$ **T**;
- $v_{\mathcal{M},s}(\exists x\, F) =$ **T** iff for some $d \in D$, $v_{\mathcal{M},s[x\mapsto d]}(F) =$ **T**.

If $v_{\mathcal{M},s}(F) =$ **T** we sometimes write $\models_{\mathcal{M},s} F$. It should be clear that $v_{\mathcal{M},s}(F)$ only depends on the values given by s to the *free* variables of F. Hence if F is a closed formula then $v_{\mathcal{M},s}(F)$ does not depend on s. In this case we write $\models_{\mathcal{M}} F$ (read F is **true** in $\mathcal{M}$) if $v_{\mathcal{M},s} =$ **T** for some (and hence for every) assignment s. In this case we also say that $\mathcal{M}$ is a **model** for F.

A closed formula is **satisfiable** if it is true in some structure and is **valid** ($\models F$) if it is true in all structures. Similar definitions of falsifiable, unsatisfiable, logical consequence and logical equivalence for closed formulae can be obtained from the propositional definitions. For the purpose of this course we will not need to extend these notions to open formulae.

As an example of the way the semantics works, consider the formula $\forall x \forall y(p(x,y) \rightarrow p(y,x))$, the structure $\mathcal{M} = (\textit{people}, \{p \mapsto \textit{siblings}\}, \emptyset, \emptyset)$ and any assignment s:

- $v_{\mathcal{M},s}(\forall x \forall y(p(x,y) \rightarrow p(y,x)) =$ **T** iff
- for all $d \in D$, $v_{\mathcal{M},s[x\mapsto d]}(\forall y(p(x,y) \rightarrow p(y,x))) =$ **T** iff
- for all $d \in D$, for all $d' \in D$, $v_{\mathcal{M},s[x\mapsto d][y\mapsto d']}(p(x,y) \rightarrow p(y,x)) =$ **T** iff
- for all $d \in D$, for all $d' \in D$, if $v_{\mathcal{M},s[x\mapsto d][y\mapsto d']}(p(x,y) =$ **T**
  then $v_{\mathcal{M},s[x\mapsto d][y\mapsto d']}p(y,x) =$ **T** iff
- for all $d \in D$, for all $d' \in D$, if $(d,d') \in$ *siblings* then $(d',d) \in siblings$ iff
- for all people d and d', if d is a sibling of d', then d' is a sibling of d.
- which is clearly true since the sibling relation(ship) is symmetric in its arguments.

It should be clear that this formula will evaluate to **T** in any structure that assigns a symmetric relation to p, and will evaluate to **F** under any interpretation that assigns an non-symmetric relation to p e.g. $\{people, \textit{father-son}, \emptyset, \emptyset\}$. Notice the way that the result of the above calculation is independent of s.

6.4 Exercises

1. For formula F and structure $\mathcal{M}$, determine the valuation $v_{\mathcal{M},s}(F)$ for any assignment s:

$$F = \forall x \forall y \forall z((p(x,y) \wedge p(y,z)) \rightarrow p(x,z))$$
$$\mathcal{M} = (people, \{p \mapsto parentof\}, \{\}, \{\})$$

2. In the above, why does the assignment s, giving values in domain D to variables, have no effect on the outcome of the valuation

7 Unification

Both propositional tableau and propositional resolution depend on the ability to detect a contradiction at the propositional level; that is, a search for a pair $p, \neg p$. The generalisation to predicates becomes complicated because of the presence of variable and terms, and therefore requires complex pattern matching. The mathematical theory of pattern matching is known as unification.

7.1 Substitutions, Unifiers and Most General Unifiers

A **substitution** is a set of pairs, denoted $[x_1/t_1, x_2/t_2, \ldots, x_n/t_n]$, where each x_i is a variable from $\mathcal{X}$ and each t_i is a term distinct from x_i. The empty substitution $\emptyset$ is known as the identity substitution. We will use lower case Greek letters like θ, σ and μ to denote substitutions.

If s is a term and θ is a substitution then the result of applying θ to s is denoted by $s\theta$ and is the result of simultaneously replacing all occurrences of x_i in s by t_i. For example, $f(x)[x/g(y)] = f(g(y))$ and $f(x)[x/g(x)] = f(g(x))$ and $f(x,y)[x/y, y/x] = f(y,x)$. Notice, however, that $(f(x,y)[x/y])[y/x] = f(x,x)$.

Two terms t and s are **unifiable** if there is a substitution θ such that $t\theta = s\theta$. Such a substitution is known as a **unifier** for s and t. Thus $f(x)$ and $f(a)$ are unifiable because the substitution $[x/a]$ gives $f(x)[x/a] = f(a)[x/a] = f(a)$. This substitution is then a unifier for $f(x)$ and $f(a)$.

If $\theta = [x_1/t_1, x_2/t_2, \ldots, x_n/t_n]$ and $\sigma = [y_1/s_1, y_2/s_2, \ldots, y_m/s_m]$ are two substitutions then their **composition** $\theta \circ \sigma$ is obtained by:

1. applying σ to each of the t_i and then deleting any pairs of the form x_i/x_i, thus giving θ';
2. deleting from σ every y_j/s_j where $y_j = x_k$ for some k, thus giving σ';
3. adding all the elements of σ' to θ', thus giving $\theta \circ \sigma$.

For example, let $\theta_1 = [y/f(z), z/b]$ and let $\sigma_1 = [z/a, y/c]$. The composition $\theta_1 \circ \sigma_1 = [y/f(a), z/b]$ whereas the composition $\sigma_1 \circ \theta_1 = [y/c, z/a]$. This shows that composition is not commutative (i.e. the order of composition matters).

Theorem 14 $(t\theta)\mu = t(\theta \circ \mu)$ *for all terms t and all substitutions θ and μ.*

A pair of terms may have more than one unifier. However, some unifiers are better than others. A unifier θ for s and t is a **most general unifier** (MGU) for s and t if: for all unifiers σ of s and t there exists a substitution μ such that $\sigma = \theta \circ \mu$. That is, if every unifier for s and t can be obtained from θ by composition with another substitution.

For example, the terms $g(x, f(y), b)$ and $g(x, f(b), b)$ have a unifier $\theta = [x/a, y/b]$ since $g(x, f(y), b)[x/a, y/b] = g(x, f(b), b)[x/a, y/b] = g(a, f(b), b)$ but a most general unifier is $\sigma = [y/b]$. That is, $g(x, f(y), b)[y/b] = g(x, f(b), b)[y/b] = g(x, f(b), b)$ *and* $\theta = \sigma \circ [x/a]$. Intuitively, an MGU is a unifier that does the *least* amount of work to unify two terms (and therefore loses the least amount of information in a resolution step—see §8). We use the expression "a" (rather

than "the") MGU since there may be more than one. For example $[x/y]$, $[y/x]$ and $[x/z, y/z, z/x]$ are all MGUs for $f(x)$ and $f(y)$. As a (tricky) exercise in substitution composition, try showing that any of these MGUs can be obtained from any other by composition with a third substitution.

Theorem 15 *If two terms are unifiable then they have a most general unifier.*

7.2 Computing Most General Unifiers

Clearly it would be useful to have an algorithm for finding the MGU given two terms. The intuition behind any such algorithm is a search for a mismatch in the given terms and then a procedure to compute a substitution that eliminates that mismatch.

More formally, consider the formation tree of each term and begin to compare the nodes in a depth-first leftmost-first manner until two nodes are found that are not identical. The nodes will be the roots for two terms s_1 and t_1; these form a **disagreement pair** (s_1, t_1).

We will use the following **unification algorithm**:

> $unify(s : term, t : term) : substitution;$
>
> Input: two terms s and t;
>
> Output: an MGU σ for s and t if they are unifiable else failure;
>
> if s and t are identical then terminate with $\sigma = \theta$;
>
> else compute a disagreement pair (s_1, t_1);
>
> if s_1 is a variable and s_1 does not occur in t_1 then (recursively) call $unify(s[s_1/t_1], t[s_1/t_1]$. If this call fails, then fail. If it succeeds with MGU μ, then succeed and return with $\sigma = [s_1/t_1] \circ \mu$
>
> else if t_1 is a variable and t_1 does not occur in s_1 then (recursively) call $unify(s[t_1/s_1], t[t_1/s_1]$. If this call fails, then fail. If it succeeds with MGU μ, then succeed and return with $[t_1/s_1] \circ \sigma$
>
> else fail.

For example, if we call unify with $s = f(x, g(b,c))$, $t = f(z, g(z,c))$ then:

> the first disagreement pair is (x, z);
>
> since x is a variable and it does not occur in z we form the substitution $\theta_1 = [x/z]$ and recursively call unify to unify $f(z, g(b,c))$ and $f(z, g(z,c))$;
>
> this recursive call leads to a disagreement pair (b, z);
>
> since b is not a variable we enter the else;
>
> since z is a variable and z does not occur in b we form the substitution $\theta_2 = [z/b]$ and recursively call unify on $f(b, g(b,c))$ and $f(b, g(b,c))$;
>
> this call to unify detects that the two terms are identical and so succeeds with MGU $[\,]$;

The original terms are therefore unifiable with MGU $\theta_1 \circ \theta_2 = [x/b, z/b]$.

If both of the terms in the disagreement pair are variables, then it doesn't really matter which one we substitute for which. The choice to substitute the second one for the first in the unification algorithm was an arbitrary one. We will see a case where one choice is more useful than the other in §9 on logic programming.

The condition that one term not occur in another term is called the "occurs check" and prevents the unification of $f(x)$ and $f(f(x))$. To see why it is needed, run the unification algorithm, minus the occurs check, on $f(x)$ and $f(f(x))$. The occurs check tells us that these terms are not unifiable.

Some of the example substitutions of the second paragraph of Section 7.1 fail the occurs check and thus could not have been obtained by the unification algorithm. Nevertheless, they are legal substitutions.

The unification algorithm can be extended to unify more than just two terms. It can also be extended trivially to unify atomic formulae since atomic formulae look just like terms except that the top-level "function" is really a predicate symbol. In particular it can be used to unify two atomic formulae. For example $p(x, f(b))$ and $p(a, f(z))$ have an MGU $\sigma = [x/a, z/b]$. Some care must be exercised, however, when extending the theory of substitutions to general formulae because of problems with the capture of free variables in the scope of quantifiers. For example, what should $(\forall x p(x,y))[y/x]$ be?

The unification algorithm, when used on two atomic formulae of the form $p(t_1, t_2, \ldots, t_n)$ and $\neg p(s_1, s_2, \ldots, s_n)$ by ignoring the $\neg$ sign forms the basis of resolution for predicate logic. That is, the MGU of $p(t_1, t_2, \ldots, t_n)$ and $p(s_1, s_2, \ldots, s_n)$ gives us a substitution (read conditions) which would make $p(t_1, t_2, \ldots, t_n)$ and $\neg p(s_1, s_2, \ldots, s_n)$ a contradictory pair of atoms.

7.3 Exercises

1. Let $p(u, f(v), g(x), y)$,
   $\theta = [u/f(y), v/x]$,
   $\sigma = [x/g(u), v/h(y), y/a]$
   Show that (i) $(p\theta)\sigma = p(\theta\sigma)$ and that (ii) $(p\sigma)\theta = p(\sigma\theta)$
2. For $t_2 = f(g(h(x), v), g(x, y), g(z, f(x, y, u)))$
   and $\theta_2 = [x/h(y, a), y/b, z/g(c)]$ find $t_2\theta_2$

If possible, find the most general unifier of the following sets of formulae:

1. $p(x, f(b))$, $p(f(a), y)$
2. $p(x, y)$, $q(x, y)$
3. $p(x, f(x))$, $p(f(a), y)$
4. $p(x, f(x))$, $p(f(y), y)$
5. $p(x, y, f(a))$ $p(y, f(z), x)$

8 First Order Resolution

The resolution rule can be extended to first-order classical logic as follows.

8.1 Prenex and Clausal Forms

Recall the definitions of literal, complement of a literal, CNF etc. from §4. These definitions carry over to predicate logic except that we now have a more general definition of an atomic formula (see §6).

A closed formula is in **prenex conjunctive normal form** if it is of the form:

$$Q_1 x_1 \cdots Q_k x_k \; M$$

where each Q_i is a quantifier, M is a formula in CNF (and hence contains no quantifiers) and $x_1, \ldots x_k$ are all the free variables of M. M is called the **matrix** of the formula.

A formula is in **preclausal form** if it is in prenex conjunctive normal form and each Q_i is a *universal* quantifier.

Since all the quantifiers in a formula in preclausal form are universal we usually omit them and write down just the matrix M. That is, we use M to represent the universal closure of M. Furthermore, we know that the matrix M is in CNF so we can write M as a set of clauses (where a clause is a set of literals) as we did for propositional CNF formulae.

For example, $\forall x \; \forall y \; \forall z \; (p(x) \wedge (\neg q(y) \vee r(z)) \wedge (p(f(a)) \vee q(g(b))))$ is in preclausal form and can be written as:

$$\{[p(x)], [\neg q(y); r(z)], [p(f(a)); q(g(b))]\}.$$

A formula written in this way is said to be in **clausal form**.

But what about existential quantifiers you say? The following theorem justifies the use of clausal form for checking satisfiability.

Theorem 16 (Skolem) *There is a purely syntactic procedure which, given a closed formula A, produces a formula A' which is in preclausal form such that A is satisfiable if and only if A' is satisfiable.*

Note that the theorem only speaks of satisfiability. It does *not* say that A and A' are logically equivalent. In fact they are not necessarily even formulae over the same language. The following section gives an outline of such a conversion procedure.

8.2 Conversion to Clausal Form

The conversion to preclausal form proceeds in two parts. The first part is to convert the given formula into an equivalent formula in prenex CNF. The second part is to convert the result into preclausal form. Although the resulting formula may not be logically equivalent to the given one,

if there is a model for the preclausal formula it is easy to convert it into a model for the given formula.

The conversion to prenex CNF is by rewriting the formula using known equivalences in first-order logic as follows:

1. rename bound variables so that no variable appears in the scope of more than one quantifier;
2. rewrite all connectives using $\wedge$ and $\vee$;
3. move the negation connective $\neg$ inward as far as it will go;
4. move all quantifiers out to the front by repeatedly choosing a quantifier in the matrix which is not in the scope of another quantifier still in the matrix and move it to the front (but still inside the other quantifiers at the front). That is, if a quantifier Q occurs in the scope of another Q' then the outermost one Q' must be moved out first;
5. put the matrix into CNF using distributive laws as in the propositional case.

The justifications for the above steps are as follows:

1. $\forall y \; F(y) =\!\mid\models \forall z \; F(z)$ and $\exists y \; F(y) =\!\mid\models \exists z \; F(z)$;
2. $A \to G =\!\mid\models \neg F \vee G$;
3. $\neg\neg F =\!\mid\models F$ and $\neg(F \wedge G) =\!\mid\models \neg F \vee \neg G$ and $\neg(F \vee G) =\!\mid\models \neg F \wedge \neg G$ and $\neg \forall x F(x) =\!\mid\models \exists x \neg F(x)$ and $\neg \exists x F(x) =\!\mid\models \forall x \neg F(x)$;
4. $F \operatorname{op} QxG(x) =\!\mid\models Qx(F \operatorname{op} G(x))$ (provided x is not free in F) and $QxF(x) \operatorname{op} G =\!\mid\models Qx(F(x) \operatorname{op} G)$ (provided x is not free in G) where Q is a quantifier and op is either $\vee$ or $\wedge$;
5. the distributive laws of propositional logic suffice since there are no quantifiers in the matrix.

For example consider the formula $\forall x(p(x) \to q(x)) \to (\forall x p(x) \to \forall x q(x))$:

- renaming all (bound) variables gives: $\forall x(p(x) \to q(x)) \to (\forall y p(y) \to \forall z q(z))$;
- rewriting $F \to G$ as $\neg F \vee G$ gives: $\neg \forall x(\neg p(x) \vee q(x)) \vee (\neg \forall y p(y) \vee \forall z q(z))$;
- rewriting $\neg\forall$ as $\exists\neg$ gives: $\exists x \neg(\neg p(x) \vee q(x)) \vee (\exists y \neg p(y) \vee \forall z q(z))$;
- pushing negations inwards gives: $\exists x(p(x) \wedge \neg q(x)) \vee (\exists y \neg p(y) \vee \forall z q(z))$;
- extracting the $\exists x$ gives: $\exists x((p(x) \wedge \neg q(x)) \vee (\exists y \neg p(y) \vee \forall z q(z)))$;
- extracting the $\exists y$ gives: $\exists x \exists y((p(x) \wedge \neg q(x)) \vee (\neg p(y) \vee \forall z q(z)))$;
- extracting the $\forall z$ gives: $\exists x \exists y \forall z((p(x) \wedge \neg q(x)) \vee (\neg p(y) \vee q(z)))$;
- using $(F \wedge G) \vee G' =\!\mid\models (F \vee G') \wedge (G \vee G')$ gives: $\exists x \exists y \forall z((p(x) \vee \neg p(y) \vee q(z)) \wedge (\neg q(x) \vee \neg p(y) \vee q(z)))$;

The conversion from prenex clausal form into preclausal form involves the elimination of all existential quantifiers from the prefix $Q_1x_1 \cdots Q_kx_k$:

1. starting from the leftmost quantifier choose the first existential quantifier $\exists x$;
2. if $\forall y_1 \cdots \forall y_n$ are all the quantifiers to the left of $\exists x$ then create a new n-ary function symbol f—that is, one which does not appear elsewhere in the formula;
3. delete the existential quantifier $\exists x$ from the current formula;
4. replace all remaining occurrences of x by $f(y_1, y_2, \cdots, y_n)$;
5. if $\exists x$ was the first quantifier then $n = 0$ so replace all occurrences of x by some new constant c;

For example:

- replacing $\exists x$ by a new 0-ary Skolem function a gives:
  $\exists y \forall z(p(a) \vee \neg p(y) \vee q(z)) \wedge (\neg q(a) \vee \neg p(y) \vee q(z));$
- replacing $\exists y$ by a new 0-ary Skolem function b gives:
  $\forall z(p(a) \vee \neg p(b) \vee q(z)) \wedge (\neg q(a) \vee \neg p(b) \vee q(z));$
- dropping $\forall z$ gives: $(p(a) \vee \neg p(b) \vee q(z)) \wedge (\neg q(a) \vee \neg p(b) \vee q(z)).$

The steps 1-5 are known as Skolemisation and are justified by the following argument: $\forall x \exists y F(x,y)$ indicates that "for all values of y there exists a value of x such that $F(x,y)$ is true". Thus, the value of y may depend on the value of x. The skolemised formula $\forall x F(x, f(x))$ just gives a new function name to this dependency while eliminating the variable y at the same time. Notice that by introducing the function symbol f, we are extending the language over which we are working in just the same way as we did by introducing new constant symbols for first-order tableaux. If we have a model for the skolemised formula we can obtain a model for the original formula by simply forgetting about the Skolem functions.

The above method is probably the simplest to apply in the general case. To minimise the arity of the Skolem functions it is sometimes better to use a slightly different method which moves all quantifiers inwards, skolemise and then moves the remaining universal quantifiers outside. For example, the given procedure turns the formula:

- $\exists x \forall y p(x,y) \rightarrow \forall y \exists x p(x,y);$
- into $\forall x \exists y \forall w \exists z(\neg p(x,y) \vee p(z,w));$
- which skolemises to $\forall x \forall w(\neg p(x, f(x)) \vee p(g(x,w),w));$
- whereas the different method gives $\forall x \exists y(\neg p(x,y)) \vee \forall w \exists z p(z,w)$
- which skolemises to $\forall x \forall w(\neg p(x, f(x)) \vee p(g(w),w))$ where g has a lower arity.

8.3 First Order Binary Resolution

We now turn to the resolution rule for first-order clauses.

Two clauses C_1 and C_2 **clash** if C_1 contains a literal L_1 and C_2 contains a literal L_2 such that $\overline{L_1}$ and L_2 have an MGU θ. Note the use of the complement; that is, the two literals must have opposite signs so that when we take the complement of one the mismatch in signs becomes irrelevant and unification can proceed. The clauses are said to clash on the literals L_1 and L_2.

For example, $[p(x); r(b)]$ and $[\neg p(a); q(y)]$ clash on the literals $p(x)$ and $\neg p(a)$ because $\overline{p(x)} = \neg p(x)$ and $\neg p(a)$ have an MGU $[x/a]$.

Let C_1 and C_2 be two clauses. If C_1 and C_2 have any variables in common, then rename the variables in one of them, say C_1, so that the new clause C_1' and C_2 have no common variables. Suppose C_1' and C_2 clash on the literals L_1 and L_2 with MGU θ. Then the clause

$$(C_1'\theta - \{L_1\theta\}) \cup (C_2\theta - \{L_2\theta\})$$

is a **binary resolvent** of C_1 and C_2 (notice there is no prime on C_1).

That is, a binary resolvent is obtained by

1. creating two new clauses $C_1'\theta$ and $C_2\theta$;
2. creating two new literals $L_1\theta$ and $L_2\theta$;
3. deleting $L_1\theta$ from $C_1'\theta$ giving C_1'';
4. deleting $L_2\theta$ from $C_2\theta$ giving C_2';
5. and then forming the union of C_1'' and C_2';
6. note that the parent clauses C_1 and C_2 remain unchanged.

The process of renaming the variables in one of the clauses is known as standardising variables apart and is vital to the completeness of the resolution procedure.

In first-order logic, unlike propositional logic, there may be more than one way to usefully resolve two clauses, since they may clash on more than one pair of literals with different unifiers. For example $[p(x); \neg q(b); r(x,y)]$ and $[\neg p(a); q(z)]$ have the binary resolvents $[\neg q(b); q(z); r(a,y)]$ and $[p(x); \neg p(a); r(x,y)]$.

Another difference with first-order logic is that, in the propositional case, the resolution rule gave us a complete proof procedure. This is not the case for binary resolution in predicate logic. The additional ingredient we need for completeness is known as *factoring*.

Suppose we are given a clause C and that there is a subset $\{L_1, L_2, \cdots, L_n\}$ of C such that $\{L_1, L_2, \cdots, L_n\}$ has an MGU θ. Then the clause $C\theta$ is known as a **factor** of C.

For example,

$[p(x); p(a); q(b,x)]$ has a factor $[p(a); q(b,a)]$

since $\{p(x), p(a)\}$ is a subset of literals from C with MGU $\theta = [x/a]$ and

$[p(x);p(a);q(b,x)][x/a] = [p(a);p(a);q(b,a)] = [p(a);q(b,a)]$.

Note that the clause $[p(x);\neg p(a);q(b,x)]$ has no factors since $p(a)$ and $p(x)$ appear with *different* signs. Thus in factoring we look for two or more literals in *one* clause with the same sign!

A **resolvent** of two clauses C_1 and C_2 is any one of the following:

1. a binary resolvent of C_1 and C_2; or
2. a binary resolvent of C_1 and C_2' where C_2' is a factor of C_2; or
3. a binary resolvent of C_1' and C_2 where C_1' is a factor of C_1; or
4. a binary resolvent of C_1' and C_2' where C_1' is a factor of C_1 and C_2' is a factor of C_2.

Given a set of clauses S_0 the *resolution procedure* is then as before for the propositional case:

1. beginning with $i = 0$ repeat
2. assume that we have formed S_i;
3. choose a pair of clashing clauses C_1 and C_2 from S_i (standardising variables apart if necessary);
4. let C be a resolvent of C_1 and C_2;
5. if $C = []$ then terminate with failure (S_0 is not satisfiable);
6. else add C to S_i to give S_{i+1};
7. until $S_i = S_{i+1}$ for all ways of choosing C_1 and C_2 and the clashing literal
8. terminate with success (S_0 is satisfiable).

(We have built factoring in by defining resolvents in terms of binary resolvents. An alternative way is to simply let the resolution procedure above add a factor of any clause in S_i to give S_{i+1}. Then options 2-4 from the definition of resolvent can be omitted so that a resolvent is just a binary resolvent)

For example,

1. the two clauses $[\underline{p(f(x),g(y))};q(x,y)]$ and $[\underline{\neg p(f(f(a)),g(z))};q(f(a),g(z))]$ clash on the underlined literals ;
2. that is, once the $\neg$ sign is dropped, they have an MGU $[x/f(a),y/z]$;
3. a binary resolvent of these clauses is: $[q(f(a),z);q(f(a),g(z))]$;
4. no factoring was involved (since it was not possible).

For example,

1. the clause $[p(a);p(x);q(x)]$ has a factor $[p(a);q(a)]$;

2. the two clauses $[p(a);p(x);q(x)]$ and $[\neg p(z);q(z)]$ have a binary resolvent $[p(x);q(x);q(a)]$;
3. and another binary resolvent $[p(a);q(z)]$;
4. but they also have a resolvent $[q(a)]$ via the factor.

For example,

1. the two clauses $[p(f(x))]$ and $[\neg p(x)]$ contain a common variable x;
2. but these clauses represent the formula $\forall x p(f(x)) \wedge \forall x \neg p(x)$ which is clearly not satisfiable;
3. standardising apart gives $[p(f(x_1))]$ and $[\neg p(x_2)]$;
4. these two have a resolvent $[]$ as desired.

Note that

1. standardisation is not allowed on the clause $[p(f(x);\neg p(x)]$ since the variable x appears in the *same* clause;
2. factoring is not allowed on the clause $[p(f(a);\neg p(x)]$ because the literals have *opposite* signs i.e. they are not unifiable;
3. factoring is not allowed on the clause $[p(f(x);p(x)]$ because they are not unifiable due to the occurs check.

8.4 Soundness and Completeness of First Order Resolution

The resolution procedure (with factoring) is sound and complete.

Soundness says that if $X \vdash_{\mathcal{R}} F$ (i.e. there is a refutation of $X \cup \{\neg F\}$) then $X \models F$.

Completeness says that if $X \models F$ then $X \vdash_{\mathcal{R}} F$; which is the same as saying if $X \not\vdash_{\mathcal{R}} F$ then $X \not\models F$. In words, if there is no refutation of $X \cup \neg F$ then $X \cup \neg F$ is satisfiable.

If we drop factoring then binary resolution is not complete for first order logic! That is if we drop items 2-4 from the definition of resolvent then the resolution procedure above is not complete.

8.5 Resolution as a (non-)Decision Procedure

Soundness guarantees that if there is a refutation of the initial set then that set is unsatisfiable. Completeness guarantees that if the initial set is unsatisfiable then there is a refutation for it. Thus, if a set is unsatisfiable we will be able to detect it.

Soundness also guarantees that if the set is satisfiable there is no refutation for the set. Completeness also guarantees that if there is no refutation for the set then the set is satisfiable. This would give us a test for satisfiability if we could guarantee that resolution terminated on all inputs.

Unfortunately, resolution is not guaranteed to terminate on all inputs. For example,

- consider the clause $C_1 = [\neg p(x); p(f(x))]$;
- a renamed copy of this clause is $C_2 = [\neg p(y); p(f(y))]$;
- C_1 and C_2 have a resolvent $[\neg p(y); p(f(f(y)))]$, which we rename as $C_3 = [\neg p(y_1); p(f(f(y_1)))]$;
- no matter what pair we chose the resolvent is always more complicated and contains two literals of opposite sign so there is no room for factoring;
- thus resolution will never terminate;
- however the clauses represent $\forall x(p(x) \rightarrow p(f(x)))$ which has an interpretation

  $$\{\mathcal{N}, \{p \mapsto \text{non-zero}\}, \{f \mapsto \text{successor}\}, \emptyset\}$$

  (among many others);
- that is, the clauses are satisfiable but resolution will never detect it because it will not terminate;

Thus resolution is a semi-decision procedure that will detect unsatisfiable sets only. That is, if $X \models F$ then we will be able to detect this. But if $X \not\models F$ then resolution may run for ever. But we already know that this is the best that is possible for first order logic since it is only semi-decidable so this is not the fault of resolution itself. However, with resolution at least we may be lucky some of the time and terminate without finding a refutation. In this case we can definitely conclude that the initial set of clauses was satisfiable.

8.6 General Resolution

Factoring can be omitted if we allow the clashing items to be sets. That is, let C_1 contain a subset C_1' of literals all with the same sign. Let C_2 contain a subset C_2' of literals all with the same sign but of opposite sign to those in C_1'. Suppose that the set $\overline{C_1'} \cup C_2'$ has an MGU θ. Then the general resolvent of C_1 and C_2 is obtained by creating two new clauses $C_1\theta$ and $C_2\theta$; creating two new sets of literals $C_1'\theta$ and $C_2'\theta$; deleting $C_1'\theta$ from $C_1\theta$ giving C_1'; deleting $C_2'\theta$ from $C_2\theta$ giving C_2'; and then forming the union of C_1' and C_2'.

8.7 Input, Linear and Set of Support Resolution

The definitions of input resolution, linear resolution and set of support are the same except that we must allow for the presence of factors in forming the resolvent.

Again, linear binary resolution with factoring is complete but input resolution is not complete. Set of support is also complete.

Linear-input resolution is another resolution strategy. The linear part means that we choose a centre clause and then always use the resolvent as a parent clause. The input part means that the other parent must be an input clause. Linear-input binary resolution is not complete for first order logic. However, linear-input binary resolution without the factoring rule *is* complete for a subset of first order logic known as Horn clauses. Horn clauses and linear-input binary resolution form the basis of Logic Programming.

8.8 Exercises

Convert the following first order formulae into clausal form:

1. $\forall x(\neg p(x) \rightarrow \forall y p(y)) \wedge \neg(\exists x p(x) \wedge \forall z q(z))$
2. $\forall x(p(x) \rightarrow q(x)) \wedge \neg((\forall x p(x)) \rightarrow (\exists x q(x)))$

Use binary resolution to prove the following, if possible:

1. $$\begin{array}{l} \exists x(p(x) \wedge (\exists y(q(y) \rightarrow r(x,y))) \\ \forall x(p(x) \rightarrow (\forall y(s(y) \rightarrow \neg r(x,y)))) \\ \models \forall x(q(x) \rightarrow \neg s(x)) \end{array}$$
2. $(\forall x(p(x) \rightarrow q(x))) \models (\exists x p(x) \rightarrow \exists x q(x))$
3. $\forall x \exists y p(x,y) \models \exists y \forall x p(x,y)$
4. $\forall x \exists y p(x,y) \models \forall z \exists x p(z,x)$
5. Is there a resolution refutation of

   $$\forall x(\neg p(x) \rightarrow \forall y p(y)) \models \exists x p(x) \wedge \forall z q(z)$$

   If so, give a refutation; otherwise say why there isn't.
6. Give a resolution refutation for $\forall x(p(x) \rightarrow q(x)) \models (\forall x p(x)) \rightarrow (\exists x q(x))$
7. $\forall x((\exists y(p(x,y) \wedge q(y))) \rightarrow (\exists y(r(y) \wedge s(x,y)))) \models \neg(\exists x r(x)) \rightarrow (\forall x \forall y(p(x,y) \rightarrow \neg q(y)))$

9 Logic Programming

We mentioned in §4 that input resolution does not provide a complete proof system even for propositional logic. This is a pity as input resolution can be implemented quite efficiently due to the reduced search space when we are looking for clashing clauses. It turns out, however, that input resolution (in fact *linear* input resolution, without factoring) is complete for sets of a particular class of clauses called Horn clauses. Furthermore, there is a natural interpretation of a set of Horn clauses has a "database" of facts and rules for deriving new facts, against which we can make queries. These observations form the basis of logic programming, which we have seen in section 3. This section will first describe resolution on Horn clauses and will therefore give the background to the operation of Prolog.

9.1 Horn clauses and linear input resolution

A **positive** literal is an atomic formula, i.e. $p(t_1,t_2,\ldots,t_n)$ where p is an n-ary predicate symbol and $t_1,\ldots,t_n$ are terms. A **negative** literal is a negated atomic formula i.e. $\neg p(t_1,t_2,\ldots,t_n)$.

A **Horn clause** is a clause that contains at most one positive literal. Thus each of $[p(a);\neg r(b);\neg q(x,c)]$ and $[\neg r(x)]$ and $[q(x)]$ are Horn clauses but $[p(a);p(b)]$ is not a Horn clause.

A **definite clause** is a Horn clause that contains exactly one positive literal, for example $[p(a);\neg r(b);\neg q(x,c)]$.

A **fact** is a definite clause that contains no negative literals e.g $[q(x)]$.

A definite clause can be written as $[H;\neg B_1;\ldots;\neg B_n]$ where H is the lone positive literal (called the head) and the $\neg B_i$ are the negative literals (called the body). A fact is $[H]$ is just a definite clause with an empty body ($n=0$).

A **goal clause** is a Horn clause that contains no positive literals. Thus a goal clause can be written as $[\neg G_1;\neg G_2;\ldots;\neg G_n]$ where each G_i is an atomic formula. Facts and goal clauses are exactly the same as those we described in section 3 for Prolog.

Suppose that we are given a set of Horn clauses X and a goal clause $\mathcal{G}$. Suppose that we use linear input resolution on this set choosing $\mathcal{G}$ as the first center clause:

1. since there is only one positive literal in each member of X the only possibility of a clash is between some negative literal $\neg G_i$ from $\mathcal{G}$ and the head H^1 of some clause $C^1 = [H^1;\neg B^1_1;\ldots;\neg B^1_{n_1}]$ from X;
2. if G_i and H^1 have MGU θ_1 then we get a new centre clause
$$\mathcal{G}' = (\mathcal{G}\theta_1 - \neg G_i\theta_1) \cup [\neg B^1_1\theta_1;\ldots;\neg B^1_{n_1}\theta_1];$$
3. again the only possibility of a clash is between some negative literal $\neg G_j$ from $(\mathcal{G}\theta_1 - \neg G_i\theta_1)$ or from $[\neg B^1_1\theta_1;\ldots;\neg B^1_{n_1}\theta_1]$, and the head H^2 of some clause $C^2 = [H^2;\neg B^2_1;\ldots;\neg B^2_{n_2}]$ from X;
4. if G_j and H^2 have MGU θ_2 then we get a new centre clause
$$\mathcal{G}'' = (\mathcal{G}'\theta_2 - \neg G_j\theta_2) \cup [B^2_1\theta_2;\ldots;B^2_{m_2}\theta_2];$$

5. continuing in this way we get a linear sequence of goals $\mathcal{G}$, $\mathcal{G}'$, $\mathcal{G}''\ldots$ and an associated set of unifiers $\theta_1,\theta_2,\ldots$ until hopefully we derive the empty clause.

Notice that we left open the choice of which negative literal $\neg G_i$ to choose from the center clause and also which head to choose if more than one clash was possible.

Furthermore, if at step 3 we found that no head clashed with $\neg G_j$ then we can always return (backtrack) to step 1 and look for another head that clashes with $\neg G_j$. A different head may result in a different unifier and a totally different new goal clause in step 2.

If we find one sequence of choices that derives the empty clause then we know that $X \cup \{\mathcal{G}\}$ is unsatisfiable.

Notice the following:

1. if X contains a goal clause then this goal clause will never be used since we always look for a positive literal from X. Thus having more than one goal clause present is useless;
2. we throw away the substitutions $\theta_1,\theta_2,\ldots,\theta_n$.

The information from these observations can be used by restricting X to be a set of definite clauses (so that it contains no goal clauses); restricting our attention to just one goal clause $\mathcal{G}$ and using the freedom of choice given to us by resolution in a systematic manner.

Specifically, when it is necessary to standardise variables apart, we always rename the variables in the input parent rather than renaming variables in the centre clause; and when unifying some G_i from the centre clause with the head H of an input clause, if the disagreement pair is (x,y) and x comes from G_i and y comes from H, then always choose the substitution $\{x/y\}$ rather than $\{y/x\}$. Since the choice of which variable to substitute for which in the unification procedure was arbitrary, we still obtain an MGU so this doesn't affect soundness or completeness.

For example, suppose we are given the set of definite clauses
$$X = \{[p(a)],[\neg p(x);p(s(x))]\}$$
Suppose we are given a goal clause $\mathcal{G} = [\neg p(y)]$:

1. there is an obvious clash between the goal $\neg p(y)$ and the input clause $p(a)$;
2. this gives an MGU $\theta_1 = \{y/a\}$ and a resolvent goal which is the empty clause $[]$;
3. now, the clause $[\neg p(y)]$ represents the formula $\forall y \neg p(y)$ thus we can conclude that $X \models \neg\forall y \neg p(y)$;
4. but this is equivalent to saying $X \models \exists y p(y)$;
5. but note that if we apply θ_1 to G_1 we get $G_1\theta_1 = p(y)\{y/a\} = p(a)$ thus giving us an actual instance of $\exists y p(y)$ which follows from the clause set X. Note that we applied the substitution to G_1, not $\mathcal{G}$.

But also notice that if the above choice had failed to derive the empty clause, or if we decided to reject this solution as uninteresting, then there is another possibility if we now backtrack to step 1:

1. there is another clash between the goal $\neg p(y)$ and the input clause $[\neg p(x); p(s(x))]\}$;
2. this gives an MGU $\theta_1 = \{y/s(x)\}$ and a resolvent goal
$$\mathcal{G}' = [\neg p(x)]\{y/s(x)\} = [\neg p(x)];$$
3. by keeping our attention on $\mathcal{G}'$ we now seek another clash by looking for a clashing head in X. If we find a head and need to standardise variables we rename the variables in the head rather than in $\mathcal{G}'$;
4. we find an obvious clash between the goal $\mathcal{G}' = \neg p(x)$ and the input clause $p(a)$;
5. this gives an MGU $\theta_2 = \{x/a\}$ and a resolvent goal which is the empty clause $[]$;
6. now, the clause $[\neg p(y)]$ represents the formula $\forall y \neg p(y)$ thus we can again conclude that $X \models \neg\forall y \neg p(y)$;
7. that is, again, $X \models \exists y p(y)$;
8. but note that if we apply $\theta_1 \circ \theta_2$ to $\neg\mathcal{G}$ we get $(p(y)\{y/s(x)\})\{x/a\}) = p(s(a))$ thus giving us another instance of $\exists y p(y)$ which follows from the clause set X.
9. to see this notice that X represents the formula $p(a) \wedge \forall x(p(x) \rightarrow p(s(x)))$;
10. notice also that the connection between θ_1 and θ_2 is x which would be lost if we were allowed to rename the variables in the goal clause.

If we find a refutation and the accumulated MGUs are $\theta_1, \theta_2, \ldots, \theta_m$, let σ be the composition $\theta_1 \circ \theta_2 \circ \ldots \circ \theta_m$. Suppose the variables in the goal $\mathcal{G}$ were $x_1, \ldots, x_k$. Then the substitution $\{x_1/x_1\sigma, \ldots, x_k/x_k\sigma\}$ is known as an **answer substitution**. The reason for the slightly convoluted definition is that we do not care what σ does to variables not appearing in $\mathcal{G}$ so we can save ourselves some work by only computing the effect of σ on $x_1, \ldots, x_k$.

Thus we have a mechanism for asking queries of the form $X \models A$ as long as X contains only definite clauses and the clause form of A is a goal clause. The easiest way to ensure the second condition is to restrict A to be a conjunction of positive literals.

That is, if $A = G_1 \wedge G_2 \wedge \ldots \wedge G_n$ where each G_i is a positive literal, then $\neg A$ is automatically represented by a goal clause $\mathcal{G} = [\neg G_1; \neg G_2; \ldots; \neg G_n]$.

Then we can use linear-input resolution in the above manner on $X \cup \{\mathcal{G}\}$ to compute an answer substitution μ which when applied to $\neg\mathcal{G}$ will give more than just a yes/no answer by giving terms to substitute for the variables in A such that, when these substitutions are made, the universal closure of the resulting formula is a logical consequence of X.

9.2 Syntax of Logic Programming

As you have no doubt found this continual switching from $X \models F$ to $X \cup \{\neg F\}$ and back again is rather confusing. To remedy this, Logic Programming uses a notation which hides all these negations. We can therefore now see how the syntax of Prolog relates to that of Horn clauses.

A Horn clause $[H_1; \neg B_2; \ldots; \neg B_n]$ is written as $H :- B_1, B_2, \ldots, B_m$. where H_1 is the lone positive literal. This use of $:-$ has nothing to do with its use in substitutions. Note there is a full stop at the end of the body.

A goal clause $[\neg G_1; \neg G_2; \ldots; \neg G_n]$ is written as $:- G_1, G_2, \ldots, G_m$. and called a **goal** where each G_i is an atomic formula. Note there is a full stop at the end of the body.

Notice that all the negation signs have been eliminated by the notation, but if we read $:-$ as an implication from right to left and read each comma in the body as $\wedge$ then rewriting $H :- B_1, B_2, \ldots, B_m$. into $H \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_m$ gives us a clausal formula.

The idea is to read a definite clause as "the head is true if the body can be made true". That is, as a *rule* by which we can conclude that the head is true whenever each component of the body is true. A fact $H :- .$ is read as "H is true" and is sometimes written as $H :- true.$. A goal $:- G_1, G_2, \ldots, G_m$. is interpreted as a question "under what circumstances (that is, for what values of the variables in the goal) can we conclude that each of the G_i is true?". The reduction of a goal to $:- .$ produces an answer substitution which provides just that—an answer to the question.

Suppose that we are given a set of Horn clauses X and a goal clause
$$\mathcal{G} = \ :- G_1, G_2, \ldots, G_n.$$

Then the following is another way of performing linear-input resolution on this set choosing $\mathcal{G}$ as the first center clause:

1. try to match some G_i with some head H^1 directly (i.e. not a clash but a match) where H^1 is from X and obtain an MGU θ_1;
2. pick up the body $B^1_1, B^1_2, \ldots, B^1_{m_1}$ of H^1, apply θ_1 to it and to the other members of $:- \mathcal{G}$. and replace G_i by this modified body giving the new goal
$$:- \mathcal{G}' = \ :- G_1\theta_1, G_2\theta_1, \ldots, G_{i-1}\theta_1, B^1_1\theta_1, B^1_2\theta_1, \ldots, B^1_{m_1}\theta_1 G_{i+1}\theta_1, G_{i+2}\theta_1, \ldots, G_n\theta_1.$$
3. look for a match (not a clash) between some G_j from the body of $:- \mathcal{G}'$. and the head H^2 of some clause $H^2 :- B^2_1, \ldots, B^2_{n_2}$. from X;
4. if G_j and H^2 have MGU θ_2 then form a new centre clause $:- \mathcal{G}''$. by replacing G_j by $B^2_1\theta_2, \ldots, B^2_{n_2}\theta_2$ and applying θ_2 to the other remaining members of $:- \mathcal{G}'$.;
5. continuing in this way we get a linear sequence of goals $:- \mathcal{G}.$, $:- \mathcal{G}'.$, $:- \mathcal{G}''.\ldots$ and an associated set of MGUs $\theta_1, \theta_2, \ldots$ until hopefully we derive the empty clause.

If we find one sequence of choices that derives the empty clause then the substitution $\sigma = \theta_1 \circ \theta_2 \circ \ldots \circ \theta_n$ gives us a way of instantiating the variables in the initial centre clause $\mathcal{G}$ so as to derive the empty clause. That is, it gives us an instance of $\mathcal{G}$ such that $X \models (\mathcal{G}\sigma)$.[3] But no negation signs are necessary because they are built into the notation. If $\mathcal{G}\sigma$ contains any variables, they should be interpreted as being universally quantified. For example, if $G\sigma = F(x)$ then the interpretation is that $X \models \forall x\, F(x)$.

[3] We drop the arrow at this point.

Notice again that we left open the choice of which G_i to choose from the center clause and also which head to choose if more than one match was possible. Different choices give different search strategies.

Furthermore, if at step 3 we found that no head matched with G_j then we can always return (backtrack) to step 1 and look for another head that matches with G_i. A different head may result in a different MGU in step 2 and a totally different new goal clause in step 2.

9.3 Prolog Revisited

We can now complete the rest of the connections with Prolog.

- a **logic program** P is a set of *definite clauses*, and the clauses are written underneath each other in some order;
- the elements of each clause are also written in a particular order which we can think of as going from left to right. So clauses and programs are really thought of as *sequences*, rather than sets.
- we are allowed to ask a goal (query) $:-\ G_1, G_2, \ldots, G_n.$.
- recall from section 3 that variables are written as identifiers beginning with upper case letters (so X is Prolog notation for x). Function and predicate symbols are identifiers not beginning with upper case letters.
- the Prolog engine chooses to attack the goal from left to right by looking for a head in P to match G_1;
- the search for the head starts at the first clause of P (remember they are ordered) and continues downwards;
- if a matching head is found and the MGU of G_1 and H is θ_1 then the new goal is

  $$:-\ B_1\theta_1, B_2\theta_1, \ldots B_m\theta_1, G_2\theta_1, \ldots, G_n\theta_1.$$

  where $B_1, B_2, \ldots B_m$ is the body of the definite clause with head H. Note the order in which the new goal is written. You might like to compare this ordering strategy with the use of stacks to implement recursion in imperative programming languages;
- the next search for a head again begins at the first clause of P but now looks for a match with $B_1\theta_1$ (unless the clause with head H was a fact, i.e. it had an empty body, in which case we proceed to $G_2\theta_1$. Notice that this is the only way in which the new goal can be shorter than the old);
- if at any stage no matching head can be found for the current goal then the Prolog engine backtracks to the last point where it found a match for the previous goal and continues the search for that previous match from that point. If no match is found, Prolog will backtrack to the goal before that, and so on;
- thus if no matching head was found for $B_1\theta_1$ the Prolog engine would backtrack and search for another matching head for G_1 since it may give a different θ_1 and a different body which would allow the new body's $B_1\theta_1$ to find a match;

- the search ends when we obtain an empty goal $:-\ .$ or when all possible matches and backtrackings have been tried.

To aid efficiency Prolog engines also usually omit the occurs check in performing unification (SWI-Prolog does this). Thus they sometimes perform substitutions like $\{X/f(X)\}$. This may lead to the interpreter going into an infinite loop trying to build an infinitely large term.

Most Prolog engines (SWI-Prolog included), also allow goals to be negated; thus $\mathcal{G}$ is allowed to contain negative literals of the form $\neg G_1$. This is known as negation as failure and is beyond the scope of these lectures.

As a detailed example of the evaluation of a logic program, consider the following: P :

$$natnum(\textit{zero}) :-\ .$$
$$natnum(succ(X)) :-\ natnum(X).$$

Suppose we are given a goal clause $:-\ natnum(Y).$:

1. there is a match between $natnum(Y)$ and the head of the fact $natnum(\textit{zero}) :-\ .$;
2. this gives an MGU $\theta_1 = \{Y/\textit{zero}\}$ and a resolvent goal $:-\ .$ since there is no associated body for the fact;
3. if we apply θ_1 to the original goal we get $:-\ natnum(Y).\{Y/\textit{zero}\}$ giving $:-\ natnum(\textit{zero})$. The answer substitution is $\{Y/\textit{zero}\}$.

If this choice of head had failed or if we asked for another solution we would backtrack to step 1:

1. there is an alternate match between $natnum(Y)$ and the head of $natnum(succ(X)) :-\ natnum(X).$;
2. this gives an MGU $\theta_1 = \{Y/succ(X)\}$ and a resolvent goal $:-\ natnum(X).\{Y/succ(X)\}$ which is $:-\ natnum(X).$;
3. starting at the top, we then search for and find a match for $natnum(X)$ in the head of $natnum(\textit{zero}) :-\ .$;
4. this gives an MGU $\theta_2 = \{X/\textit{zero}\}$ and a resolvent goal which is the empty goal $:-\ .$;
5. if we apply $\theta_1 \circ \theta_2$ to the original goal we get $(:-\ natnum(Y).)\{Y/succ(X)\} \circ \{X/\textit{zero}\} = (:-\ natnum(succ(\textit{zero})))$. The new answer substitution is $\{Y/succ(\textit{zero})\}$.
6. the intended meaning of $natnum$ is “natural-number” and the program is supposed to say that 0 is a natural number and that the successor of X is a natural number if X is a natural number;
7. thus the answers are respectively that 0 is a natural number and that $succ(0)$ i.e. 1 is a natural number;

9.4 SLD-Resolution and SLD-Trees

The search strategy of Prolog can be represented by a tree structure known as an *SLD-Tree*, because Prolog uses *SLD resolution*.

Each Prolog program and goal will produce a different SLD-tree. The idea is that the choice order, back-tracking and so on, can be represented by traversing the various paths of the SLD-tree in a particular way.

First we define an SLD-Tree for goal G_0, with respect to a program $P = R_1 \dots R_k$, i.e. with rules R_i, $i = 1 \dots k$. Each internal node in the SLD-Tree is labelled by a goal clause. The children of a node are given an order. Leaf nodes are either labelled 'fail' or by the empty clause, i.e. meaning that a successful resolution has been achieved via the path from the root to that leaf node. A child is reached from its parent by performing a resolution step. It is convenient to label to edge with the rule used and the MGU substitution applied.

- The root node is labelled with the initial goal clause G_0
- Suppose a parent node i is labelled with goal clause $G_i = G_{i1}, \dots, G_{im}$, having m literals. Each successive child node of i is formed by resolving G_{i1} with the head H_j of the next matching rule R_j:

  $$R_j = H_j : -B_{j1}, \dots, B_{jn}$$

  i.e. having n negative literals. The resolvent uses MGU θ_j.

  The new child is labelled with the resulting goal clause:

  $$[B_{j1}\theta_j, \dots, B_{jn}\theta_j, G_{i2}\theta_j, \dots, G_{im}\theta_j].$$

  i.e. the new goal clause is formed by removing the first literal G_{i1} from the literals in the parent goal clause, appending these to the negative literals from the rule R_j and applying the MGU θ_j to all the literals in the new sequence. Note that the order of the literals is retained and that the remaining literals from the original goal clause are *appended* to the body of rule R_j.

  The edge going from the parent node to the new child node is labelled with (R_j, θ_j), so that the MGUs can be retrieved later to compute the answer substitution.
- A branch is (finitely) **failed** if there is no resolvent and the leaf node can be labelled 'fail'.
- A branch is **closed** if the resolvent is empty
- Note that a branch may be **infinite** (what would happen if Prolog started to explore such a branch?)

The complete SLD-Tree represents the complete behaviour of P and initial goal clause G_0. The Prolog search strategy performs a *depth-first search* on the tree, exploring branches from a node in a *left-to-right order*. In other words, given the current goal clause, it takes the first literal from the clause and matches it with the first rule it finds in program P. It then continues performing resolution with the resulting (centre) goal clause. Back-tracking is performed by traversing back up the SLD-Tree until a node is found with branches which have yet to be explored — these are visited in sequence.

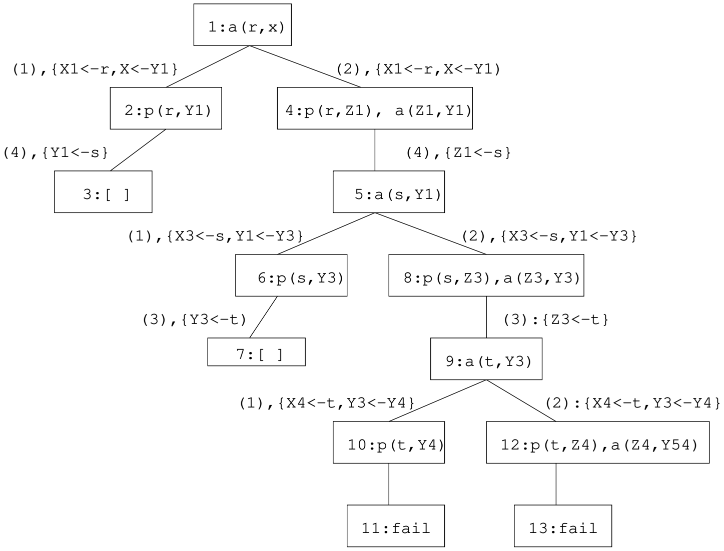
Every instantiation of a rule is given 'fresh' variables. Where a substitution involves two variables then the variable in the parent goal is renamed.

For each **closed** branch, representing a successful resolution, then the MGUs on that branch can be composed in sequence and the answer substitution obtained by restricting the result to variables in the initial goal G_0.

For example, consider the following Prolog program P:

```
Rule (1)    a(X,Y) :- p(X,Y).
Rule (2)    a(X,Y) :- p(X,Z), a(Z,Y).
Rule (3)    p(s,t).
Rule (4)    p(r,s).
```

taken with goal clause G = `a(r,X)`. The resulting SLD-Tree is:



The label for each node is boxed, so Node 1 is labelled '`1:a(r,X)`'.

Node 2 is produced from Node 1, using Rule 1 (with fresh variables $X1, Y1$), and MGU $\{X1/r, X/Y1\}$. Note that X becomes $Y1$ rather than the other way round. The resulting goal clause has a single literal $p(r, Y1)$ which labels the child Node 2. The edge from Node 1 to Node to is labelled '`(1),`$\{$`X1<-r,X<-Y1`$\}$', meaning that Rule (1) was used in this step with MGU '$\{$`X1<-r,X<-Y1`$\}$'.

In the next step taken, the Node 3 is the result where the empty clause is produced by resolving $p(r, Y1)$ with Rule 4, i.e. $p(r, s)$, with MGU $\{Y1/s\}$.

The two MGUS can be composed: $\{X1/r, X/Y1\} \circ \{Y1/s\}$ to produce $\{X1/r, X/s, Y1/s\}$ and the answer substitution is obtained by restricting the domain to X, giving $\{X/s\}$.

Prolog now backtracks to Node 2, but fails to find a further path to explore (i.e. no new rules are matching), so backtracks to Node 1, where it can match with Rule 2 to form Node 4. It forms Node 5 via resolution wih fact 4. Node 6 and Node 7 yield the answer substitution $\{X/t\}$. Backtracking produces no choice at Node 6, but a second branch at Node 5 via Rule 2. Further exploration leads to two failed branches, because literals $p(t, Y4)$ and $p(t, Z4)$ fail to match a rule.

References

[1] J. Kelly. *The Essence of Logic.* Prentice Hall, 1997.

[2] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT, 1986.

[3] Clocksin and Mellish. *Programming in Prolog.* Springer-Verlag, 1994.

[4] U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog.* Wiley, 1990.

All of these references should be available either in the John Rylands Library or in the Departmental Research Library.