Contents

Chapter 1

Describing Information Systems in Logic

In these lecture notes you will encounter a way of describing information and the processing of information that is precise while remaining independent of any particular computing machine. This chapter starts you describing information in logic, in preparation for subsequent chapters where we use formally described information to solve problems via inference.

1.1 Motivating Problems

Problem 1 *Suppose you have data describing films; their directors; their actors and actresses. Each data item is described using the following expressions:* $film(F)$ *means* F *is a film;* $actor(F, P)$ *means that person* P *acted in film* F*;* $director(F, P)$ *means that person* P *directed film* F*;* $oscar(X)$ *means that* X *(which can be a person or a film) got an Oscar. For example, the film "Mars Attacks" was directed by Tim Burton with Jack Nicholson and Glenn Close (both Oscar winners) being actors in it so*

$$\begin{array}{l} film(\text{'MarsAttacks'}) \\ director(\text{'MarsAttacks'}, \text{'TimBurton'}) \\ actor(\text{'MarsAttacks'}, \text{'JackNicholson'}) \\ actor(\text{'MarsAttacks'}, \text{'GlennClose'}) \\ oscar(\text{'JackNicholson'}) \\ oscar(\text{'GlennClose'}) \end{array}$$

are true.

How could we describe precisely the following questions about our data?

- *Which Oscar-winning films were directed by an actor?*
- *Which Oscar winning actors or actresses have directed themselves?*
- *Which directors have directed more than one film?*

- *Which films have more than one director?*

1.2 Notation

The notation used in these notes - First Order Predicate Logic - is defined precisely in Appendix A. Before reading that, here is a guide to writing in this sort of notation. In what follows (and in the rest of these notes) I use the word "expression" to refer to something written formally in this style.

The basic element of an expression is a predicate. A predicate has a name and zero or more arguments, written $P(A_1, \ldots, A_n)$ where P is the predicate name and each A is an argument. Predicates can either be true or false in whatever model of the world we describe. Predicate names are constants - that is, they refer to a specific thing in our model of the world. Arguments may be constants or variables. A variable is a way of referring to a thing without saying which specific thing it is. So if $eats(A, B)$ means (in general) that A eats B then by writing $eats(dave, nemo)$ we say that the thing called $dave$ eats the thing called $nemo$. What if we want to say that "everything eats something"? Then we need variables so that we can refer to things without committing to specific things but we also need to use the variables in two different ways: the "everything" refers to all things while the "something" is looking for only one (at least). To differentiate these two uses of a variable we use quantifiers: a universal quantifier (written $\forall A$ where A is the quantified variable) for the former case and an existential quantifier (written $\exists B$ where B is the quantified variable) for the latter case. So "everything eats something" could be written as $\forall A.\exists B.eats(A, B)$.

In addition to describing individual predicates it is useful also to be able to describe conjunctions of expressions (written $E_1\ and\ E_2$); disjunctions of expressions (written $E_1\ or\ E_2$); implications (written $E_1 \rightarrow E_2$) and negations (written $not(E)$). Any of the Es in these expressions can themselves be expressions so we can write things like $((a\ and\ b)\ or\ c) \rightarrow d$. To make these easier to read we agree that some operators dominate others. This is referred to as establishing the precedence of operators. In the absence of explicit bracketing, the operator with the highest precedence will be the principal operator in an expression.

We take the precedence ordering to be the following:

- $\rightarrow$ and $\leftrightarrow$ have highest precedence.
- and and or have next highest precedence.
- not has lowest precedence.
- In cases where two operators of equal precedence appear in an expression and there is no bracketing to impose an explicit ordering on the operators, then the operator furthest to the right is taken as the principal operator.

Given this convention we can write $((a\ and\ b)\ or\ c) \rightarrow d$ as $a\ and\ b\ or\ c \rightarrow d$ which is easier on the eye.

1.3 Why There are Many Ways to Describe the Same Concept

Once you start writing formal descriptions you will find that there are many ways of saying essentially the same thing. Let's consider the reasons for this.

1.3.1 Ambiguity in Understanding the World

When we observe something it is normally difficult for us to be precise about what we have observed. Suppose, for example, that you observe someone (call him Dave) with hair thinning on top of his head. How would you describe this? You might simply say he is bald but perhaps that is too crude a statement. Maybe it would be more accurate to say he is "going bald" or that he is "partially bald". Perhaps you might even have a quantititative measure of "hair follicle density" and rate him on that scale. None of these are absolutely right or wrong. In the end you will choose your formal description based on your experience in writing and using such descriptions.

1.3.2 Boundary Choices

There is always a choice about what to represent explicitly in our formal model and what to treat as a primitive, indivisible concept. For instance, in the film example of Problem 1 we used the predicate $oscar(X)$ to denote that some person or film won an Oscar. This is the right level of detail if our concern is only to identify oscar winning people in some absolute sense but it does not represent which specific films gave each person their Oscars - for that we would need a predicate like $oscar\_winner(P, F)$ which relates a person P to a film F in which they won an Oscar. Does the date at which they won the oscar matter? If so, we might have a more detailed predicate, $oscar\_win(P, F, T)$, which is as before but with T being the time at which the Oscar was awarded. We could continue to expand the detail of our description and, with this, expand the interrelationships we make between the elements of our description, such as:

$$oscar\_win(P, F, T) \rightarrow oscar\_winner(P, F)\ and\ oscar(P)$$

At some stage, however, we must choose a level of description adequate for the task we wish to undertake and consider other information to be outside the boundary of our model.

1.3.3 Equivalences Between Formal Expressions

Even when we believe we have a resolution to the issues of ambiguity and boundary we still have choices to make because the same thing may be said formally in many different ways. How we choose which way to say something depends partly on elegance of expression - the ability to say exactly what is needed with the minimum of representational effort - and partly on the sorts of inference we wish to perform. For example it is true that $a \rightarrow b$ is equivalent to $not(a)\ or\ b$ (if you don't believe this yet then by the end of the next chapter you will be able to prove it). Logically there is no reason to prefer one over the other but pragmatically there may be strong preferences either way - for example $a \rightarrow b$ might feel more "natural" in human terms or, alternatively, $not(a)\ or\ b$ might better suit certain forms of inference (such as Resolution which you meet in a later chapter).

1.3.4 Differences in Logical Systems

Connected with the variant of formal expression we use is the choice of which system of logic to employ. Here there often is a tension between, at one extreme, using a logic with a simple representation for which inference may be extensive versus, at the other extreme, using a logic that allows more complex forms of representation but less extensive inference. For instance, in Chapter 2 you will encounter basic Propositional Logic in which no variables are allowed in expressions. The representation is therefore simplified and it is always possible in this logic to decide automatically the truth or falsity of any given expression. Once we move to First Order Logic, in Chapters 3 and beyond, we are able to describe things in a more sophisticated way but we may not always be able automatically to decide the truth or falsity of each expression we can write. To ameliorate this problem (and to make certain kinds of sophisticated representation easier to do) there are many variants of logic tuned to particular types of problem. In Chapter 6 you will encounter an example of a logic intended for problems where temporal change to the state of a system is important. All of the variants of logic you will see here are essentially the same - they can all be understood as predicate logic - but they have pragmatic differences according to their intended use. Experts in applied logic are good at making the right pragmatic choices.

1.4 On the Similarity Between Expressions

Section 1.3.3 raised the issue that we can in logic have many different ways of expressing the same concept. We now explore this issue a little further but, this time, with the aim of using dualities between forms of expression to deepen our understanding of what the expressions mean.

1.4.1 Implication Versus Conjunction With Negation

What do we mean when we say that "A implies B". In human conversation we sometimes mean radically different things but in the logics e consider here we mean only one thing: that B is true whenever A is true. Stated without using implication, this is the same as saying that either B is true or A is not true (since if A is true then implication means that B must be true as well). Formally:

$$a \rightarrow b \quad \text{is equivalent to} \quad not(a)\ or\ b$$

This helps us understand why implication (in this logical sense) is not the same as causality. Saying that A implies B in this formal way is not the same as saying that A causes B to be true. All we are saying is that whenever we observe A we also will observe B, although the actual causes of B might be something different from A.

1.4.2 Conjunction Versus Disjunction

Negation allows us to interchange conjunctions with disjunctions. The principle is that if we say A and B are both true then this is the same as saying neither is false. Similarly if we say A or B is true then this is the same as saying that they cannot both be false. Formally:

$$\begin{array}{rll} a\ and\ b & \text{is equivalent to} & not(not(a)\ or\ not(b)) \\ a\ or\ b & \text{is equivalent to} & not(not(a)\ and\ not(b)) \end{array}$$

1.4.3 Universal Quantification Versus Conjunction

When we universally quantify a variable we are saying that any object could match that variable, so for example if there are only two objects in our model of the world, call them x and y then $\forall X.p(X)$ would in this circumstance be the same as saying $p(x)\ and\ p(y)$. Expressing this more generally:

$$\begin{array}{l} \text{If all instances of } X \text{ are } a_1, a_2, \ldots a_n \text{ then} \\ \forall X.p(X) \quad \text{is equivalent to} \quad p(a_1)\ and\ p(a_2)\ and\ \ldots p(a_n) \end{array}$$

1.4.4 Existential Quantification Versus Disjunction

When we existentially quantify a variable we are saying that some object should match that variable, so for example if there are only two objects in our model of the world, call them x and y then $\exists X.p(X)$ would in this circumstance be the same as saying $p(x)\ or\ p(y)$. Expressing this more generally:

If all instances of X are $a_1, a_2, \ldots a_n$ then
$\exists X.p(X)$ is equivalent to $p(a_1)$ *or* $p(a_2)$ *or* $\ldots p(a_n)$

1.5 Solving The Problems of Section 1.1

1.5.1 Solving Problem 1: Describing Data

Each of the questions of Problem 1 can be described as a conjunctive ("and-ed") expression where the conjunction relates variables and satisfying the expression requires us to find an instance of the variables satisfying the whole conjunctive expression. Here are the appropriate expressions:

- Which Oscar-winning films were directed by an actor?
  $\exists F, P.oscar(F)\ and\ director(F, P)\ and\ \exists F1.actor(F1, P)$
- Which Oscar winning actors or actresses have directed themselves?
  $\exists P, F.oscar(P)\ and\ actor(F, P)\ and\ director(F, P)$
- Which directors have directed more than one film?
  $\exists F1, F2, P.director(F1, P)\ and\ director(F2, P)\ and\ not(F1 = F2)$
- Which films have more than one director?
  $\exists F, P1, P2.director(F, P1)\ and\ director(F, P2)\ and\ not(P1 = P2)$

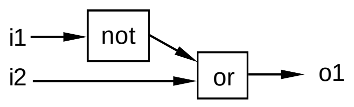
Chapter 2

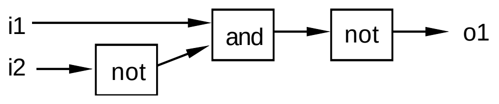
Basics of Propositional Logic

In this chapter we discuss a basic method for determining the truth or falsity of logical expressions that do not contain variables.

2.1 Motivating Problems

Problem 2 *Each of the two logic circuits below takes two inputs (i1 and i2) and produces an output (o1). Inputs and outputs are either 0 or 1. The circuits are built using three type of logic gate: a "not" gate takes an input and produces the opposite output (input 1 gives output 0; input 0 gives output 1); an "and" gate gives output 1 if both inputs are 1 but gives output of 0 otherwise; an "or" gate gives output 1 if either input is 1 but gives output of 0 otherwise. Would the two circuits always do the same thing?*





Problem 3 *Imagine a society in which husbands must remain faithful to their wives, otherwise their wives will shoot them (without fail) on the day they find out. One day a group of three married women is brought to the police station and told that some of their husbands are cheating on them. Each wife knows whether or not the others' husbands are faithful but does not know whether her own husband is faithful. On this first day no husbands are shot. The next day the same group of women is brought together again and informed that, so far, no husband has been shot. Again, no husbands are shot. On the third day, however, the group is re-convened and that day there is shooting. How many husbands were unfaithful?*

P	Q	$not(P)$	$P\ and\ Q$	$P\ or\ Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
t	t	f	t	t	t	t
t	f	f	f	t	f	f
f	t	t	f	t	t	f
f	f	t	f	f	t	t

Figure 2.1: A Truth Table

2.2 Truth Tables

Recall that the truth–functional connectives are so called because they evaluate to either true or false, depending on the truth values of the sub–expressions that they connect. We can enumerate all the possible results of this evaluation for each of the connectives, as shown in the table in Figure 2.1. This table is normally referred to as a *truth table*. To form this table, we write down (in the first two columns) all possible combinations of truth values for expressions P and Q. We then make a column for each connective applied to P and/or Q and enter the appropriate truth value for each row. We use the letter 't' to represent truth and 'f' to denote falsity.

Since we can now allocate truth values to all the connectives of the propositional logic, we are able to determine the truth or falsity of any expression in the logic, given the truth values of all the atomic propositions contained in it. This is done by progressively evaluating the truth of each sub–expression, starting with the connectives joining atomic propositions and propagating the truth values "higher up" in the structure of the expression. Using this method we can distinguish several useful categories of expression:

- **Tautologies**: where the expression is always true regardless of the truth values of the propositions that it contains. A simplest form of tautology is *P or not P*. An example expression $(P\ and\ (P \rightarrow Q)) \rightarrow Q$ is also a tautology, as we shall demonstrate later.
- **Contradiction or Inconsistent expressions**: where the expression is always false regardless of the truth values of the propositions that it contains. For instance, the expression $P\ and\ \ not\ P$ is always inconsistent.
- **Contingent expressions**: where the expression is sometimes true and sometimes false, depending on the truth values of the propositions that it contains. The expression $P\ and\ Q \rightarrow R$ is an example.

Figure 2.2 contains a detailed example of this method of establishing truth values. The goal is to prove that $(P\ and\ (P \rightarrow Q)) \rightarrow Q$ is a tautology. To do this, we enumerate each possible combination of truth values for P and Q (giving 4 combinations in all) and, for each combination, propagate the truth values up through the structure of the expression. The curly braces show the truth values

Goal: to prove that $(P\ and\ (P \rightarrow Q)) \rightarrow Q$ is a tautology. Consider every possible combination of truth values for P and Q:

If P is true and Q is true then the expression is true:

$$\underbrace{\underbrace{(\underbrace{P}_{t}\ and\ \underbrace{(\underbrace{P}_{t} \rightarrow \underbrace{Q}_{t})}_{t})}_{t} \rightarrow \underbrace{Q}_{t}}_{t}$$

If P is true and Q is false then the expression is true:

$$\underbrace{\underbrace{(\underbrace{P}_{t}\ and\ \underbrace{(\underbrace{P}_{t} \rightarrow \underbrace{Q}_{f})}_{f})}_{f} \rightarrow \underbrace{Q}_{f}}_{t}$$

If P is false and Q is true then the expression is true:

$$\underbrace{\underbrace{(\underbrace{P}_{f}\ and\ \underbrace{(\underbrace{P}_{f} \rightarrow \underbrace{Q}_{t})}_{t})}_{f} \rightarrow \underbrace{Q}_{t}}_{t}$$

If P is false and Q is false then the expression is true:

$$\underbrace{\underbrace{(\underbrace{P}_{f}\ and\ \underbrace{(\underbrace{P}_{f} \rightarrow \underbrace{Q}_{f})}_{t})}_{f} \rightarrow \underbrace{Q}_{f}}_{t}$$

Since the expression is true for all truth values of P and Q, it is a tautology.

Figure 2.2: Proving that $(P\ and\ (P \rightarrow Q)) \rightarrow Q$ is a tautology

Goal: to prove that $P\ and\ not(P)$ is inconsistent. Consider every possible assignment of truth values for P:

If P is true then the expression is false:

$$\underbrace{\underbrace{P}_{t} \wedge \underbrace{not(\underbrace{P}_{t})}_{f}}_{f}$$

If P is false then the expression is false:

$$\underbrace{\underbrace{P}_{f}\ \ and\ \ \underbrace{not(\underbrace{P}_{f})}_{t}}_{f}$$

Since the expression is false for all truth values of P it is a contradiction.

Figure 2.3: Proving that $P\ and\ not(P)$ is inconsistent

assigned to each part of an expression during this process. The final truth value assigned to each of the 4 expressions is 'true' so this expression is a tautology.

Proving that an expression is inconsistent is also straightforward. An example appears in Figure 2.3. Again, we establish the truth value for the entire expression, given each truth value for P. Whatever truth value we assign to P the expression is false so it is inconsistent.

2.3 Solving The Problems of Section 2.1

2.3.1 Solving Problem 2: Proving Equivalence

We can represent each of the logic circuits of Problem 2 as a logical expression. The circuit on the left is the expression $not(i1)\ or\ i2$ and the circuit on the right is the expression $not(i1\ and\ not(i2))$. The output of each circuit ($o1$) then corresponds to the truth value of its corresponding logical expression, given the truth values of the inputs ($i1$ and $i2$). Now we can draw truth tables for the expressions, as shown below.

$i1$	$i2$	$not(i1)$	$not(i2)$	$not(i1)\ or\ i2$	$i1\ and\ not(i2)$	$not(i1\ and\ not(i2))$
t	t	f	f	t	f	t
t	f	f	t	f	t	f
f	t	t	f	t	f	t
f	f	t	t	t	f	t

If you compare the columns of truth values for $not(i1)\ or\ i2$ and $not(i1\ and\ not(i2))$ these are identical so the two expressions (and the circuits they represent) give identical behaviour. Therefore, as far as their logic is concerned, each is as good as the other. There may, however, be good engineering reasons for preferring one over the other: compactness; types of logic gate used; *etc.* Things that are logically eqivalent need not physically be the same.

2.3.2 Solving Problem 3: Attaining Common Knowledge

Problem 3 might seem fanciful but it demonstrates a critical problem for large, distributed systems (like the Internet): how do groups of information processors achieve common knowledge?

Let's use the letters "a", "b" and "c" to identify each of the potentially unfaithful husbands, so the proposition a means that husband "a" is unfaithful and $not(a)$ means that "a" is faithful. We can then build the truth table shown below for all the possible situations that could apply for all truth values for a, b and c.

a	b	c	$a\ and\ b\ and\ c$	$a\ and\ b\ and\ not(c)$	$a\ and\ not(b)\ and\ c$	$a\ and\ not(b)\ and\ not(c)$	$not(a)\ and\ b\ and\ c$	$not(a)\ and\ b\ and\ not(c)$	$not(a)\ and\ not(b)\ and\ c$	$not(a)\ and\ not(b)\ and\ not(c)$
t	t	t	t	f	f	f	f	f	f	f
t	t	f	f	t	f	f	f	f	f	f
t	f	t	f	f	t	f	f	f	f	f
t	f	f	f	f	f	t	f	f	f	f
f	t	t	f	f	f	f	t	f	f	f
f	t	f	f	f	f	f	f	t	f	f
f	f	t	f	f	f	f	f	f	t	f
f	f	f	f	f	f	f	f	f	f	t

Now let's consider what the women know at each stage. When they first get together each woman knows that some husband is unfaithful, so this excludes the last row of our truth table (where a, b and c are false). The next day no husbands have been shot and this allows each woman to exclude the rows of the truth table in which only one husband is unfaithful, since if this were the case his guilt would have been revealed to the woman concerned (who would know that the others were faithful and thus identified her own husband as unfaithful). On the second day there still has been no shooting so each woman can exclude the rows of the truth table in which exactly two husbands are unfaithful, since if this had been the case (knowing from the previous day that more than one husband is unfaithful) the woman concerned would have had to see that only one of the other husbands was unfaithful and thus identify her own husband as unfaithful. This leaves us with only one row in the truth table, in which a, b and c are true and, sure enough, all of the husbands are shot on the third day.

So it took three cycles to attain shared common knowledge in this simple information system. Now imagine that instead of women we have computer processes in their millions distributed across the world and these need to attain common knowledge. Do you believe this is possible? Most people that I know believe, in an absolute sense, it is impossible yet a large amount of common knowledge is shared. How does that work?[1]

[1]You need to attend the lectures to find out.

Chapter 3

Proof With More Complex Expressions: Sequent Calculus

The truth tables of Chapter 2 are effective where it is practical to enumerate all the combinations of truth values for our logical expressions but they are cumbersome for complex problems. An alternative (and more common) method is to view proof as the application of trusted proof rules to a set of axioms (describing the problem) in order to satisfy some goal expression. This chapter introduces you to an example of this form of proof.

3.1 Motivating Problems

Problem 4 *A common operation in database systems is to construct a new table of data by joining two existing tables but choosing only those combined rows that satisfy given constraints. Suppose, for example, that we have the following two tables of data:*

lecturer		
name	*sex*	*age*
dave	*male*	*42*
mary	*female*	*21*
phil	*male*	*64*

employee	
name	*ID*
dave	*2345*
ken	*9324*
mary	*6782*
phil	*7934*

We now want to construct a joined table containing (joined) the rows of the original tables only for those rows in which the lecturer and employee names are the same and for which the age of the lecturer is less than 50. Define this in such *a way that the specification of the join would work for any size of tables in the given form.*

Problem 5 *In the English language we can differentiate grammatically well formed from grammatically ill formed sentences. A context free grammar describes permitted sequences of words by showing how these are composed from permitted sub-sequences of words. Each of these permitted compositions is described by a grammar rule. For example:*

$$\begin{array}{rcl} sentence & \Rightarrow & nounphrase, verbphrase \\ nounphrase & \Rightarrow & noun \mid determiner, noun \\ verbphrase & \Rightarrow & verb \mid verb, nounphrase \end{array}$$

is a collection of three grammar rules. The first says that the sequence of words in any English sentence must be composed from a nounphrase sequence followed by a verbphrase sequence. The second rule says that a nounphrase sequence must be composed from a noun or from a determiner followed by a noun. The third rule says that a verbphrase must be composed from a verb or a verb followed by a nounphrase. How would we define grammars like the one above as theories in logic that we could use to generate grammatically valid sequences of words or test any sequence of words for conformance to the given grammar?

3.2 Proofs From Assumptions

The proof method used in this chapter is a basic sequent calculus. A proof rule provides a strategy for establishing the truth of an expression, given the truth of other expressions. For example, we know that $A\ and\ B$ is true given some set of assumptions, S, if we can prove that A is true, given S and that B is true, given S. We write the sequent $S \vdash C$ to denote that the expression, C, can be proved from the list of assumptions, S. If S is empty, then C is a theorem (*i.e.* its truth depends on no assumptions).

Some examples of valid sequents, some of which are expressed as theorems, are shown in Figure 3.1. Some of these will be referred to in later sections – particularly when discussing conversion to normal forms in Section 5.2.

The proof rules which we shall use in this section are shown in the table in Figure 4.1. Each row of this table corresponds to a particular proof rule, consisting of: a name, for easy reference; a sequent for which the rule provides proof; and the supporting proofs which are necessary in order to establish the truth of this sequent. Below, we restate each of the proof rules in English.

- immediate: provides a proof of an expression A from some assumptions if A is one of the assumptions.

Commutativity of '*and*'	$[] \vdash (A\ and\ B) \leftrightarrow (B\ and\ A)$
Associativity of '*and*'	$[] \vdash (A\ and\ (B\ and\ C)) \leftrightarrow ((A\ and\ B)\ and\ C)$
Transitivity of '$\rightarrow$'	$[(A \rightarrow B)\ and\ (B \rightarrow C)] \vdash A \rightarrow C$
Equivalence $\rightarrow$, *or*	$[] \vdash (P \rightarrow Q) \leftrightarrow (not(P)\ or\ Q)$
Equivalence *and*, *or*	$[] \vdash not(P\ and\ Q) \leftrightarrow (not(P)\ or\ not(Q))$
Equivalence *or*, *and*	$[] \vdash not(P\ or\ Q) \leftrightarrow (not(P)\ and\ not(Q))$
Equivalence $\leftrightarrow$, $\rightarrow$	$[] \vdash (P \leftrightarrow Q) \leftrightarrow (P \rightarrow Q)\ and\ (Q \rightarrow P)$
Distribution of *or* over *and*	$[] \vdash (P\ or\ (Q\ and\ R)) \leftrightarrow ((P\ or\ Q)\ and\ (P\ or\ R))$

Figure 3.1: Some useful sequents

- and_intro: states that we can know $A\ and\ B$, given some assumptions if we can prove A from those assumptions and also prove B from those assumptions.
- and_elim: allows us to prove an expression, C, from some assumptions if we can find a conjunction, $A\ and\ B$, among those assumptions and obtain a proof of C from the assumptions with the addition of A and of B.
- or_intro_left: gives a proof of an expression, $A\ or\ B$, from some assumptions if A can be proved from those assumptions.
- or_intro_right: gives a proof of an expression, $A\ or\ B$, from some assumptions if B can be proved from those assumptions.
- or_elim: allows us to prove an expression, C, from some assumptions if we can find a disjunction, $A\ or\ B$, among those assumptions and obtain a proof of C from the assumptions with the addition of A, and then obtain a proof of C from the assumptions with the addition of B.
- imp_intro: states that we can know $A \rightarrow B$, given some assumptions if we can prove B from those assumptions with A added.
- imp_elim: allows us to prove an expression, C, from some assumptions F if we can find an implication, $A \rightarrow B$, among those assumptions. We then obtain a proof of A from the assumptions F; then find a proof of C from the assumptions with B added.

To understand how the rules of Figure 3.2 can be used, consider the following example. The proof is of the sequent: $[b \rightarrow c] \vdash (a\ or\ b) \rightarrow (a\ or\ c)$. Intuitively, this seems valid because if c follows from b then if some other proposition, a, is true then $(a\ or\ c)$ is true; on the other hand if b is true then c is true and so $(a\ or\ c)$ is true. To prove this formally it is first necessary to apply the *imp_intro* rule to insert $(a\ or\ b)$ among the set of assumptions. We then apply the *or_elim* rule which allows us to prove $(a\ or\ c)$ given the presence of $(a\ or\ b)$ in the set of assumptions and independent proofs of $(a\ or\ c)$ from a and from b.

The full proof is shown, using a tree diagram, in Figure 3.3. At the top of the diagram is shown the initial sequent. The branches below it show the application

Rule name	Sequent	Supporting proofs
immediate	$\mathcal{F} \vdash A$	$A \in \mathcal{F}$
and_intro	$\mathcal{F} \vdash A \text{ and } B$	$\mathcal{F} \vdash A,\ \mathcal{F} \vdash B$
or_intro_left	$\mathcal{F} \vdash A \text{ or } B$	$\mathcal{F} \vdash A$
or_intro_right	$\mathcal{F} \vdash A \text{ or } B$	$\mathcal{F} \vdash B$
or_elim	$\mathcal{F} \vdash C$	$A \text{ or } B \in \mathcal{F},\ [A\|\mathcal{F}] \vdash C,\ [B\|\mathcal{F}] \vdash C$
imp_elim	$\mathcal{F} \vdash B$	$A \rightarrow B \in \mathcal{F},\ \mathcal{F} \vdash A$
imp_intro	$\mathcal{F} \vdash A \rightarrow B$	$[A\|\mathcal{F}] \vdash B$

Where: $\mathcal{F}$ is some list of assumptions.
A and B are well formed expressions.
$X \in Y$ denotes that X is an element of set Y.

Figure 3.2: A set of proof rules without negation

of the proof rules. This form of tree is called an "and tree" because if there is a set of branches $[B_1, B2, \cdots, B_N]$ arising from a given sequent then it is necessary to satisfy B_1 *and* $B2$ *and* $\cdots$ *and* B_N. Using this diagram, one can reconstruct the temporal sequence of steps taken to perform the proof by starting at the top; moving downwards; and always exploring left hand branches before those to the right. This form of search is known as depth–first, left–to–right search because all branches are explored out to their tips, starting with those farthest to the left.

It often is useful to have some strategy for choosing which rule to apply at any given stage in a proof. One of the many strategies is given in Figure 3.4.

Let us now consider how the search strategy of Figure 3.4 is applied to a particular problem. Suppose that we want to establish the sequent:
$[a \leftarrow b, a \leftarrow (d \text{ or } e), e \leftarrow (f \text{ and } g), f, g] \vdash a$
The sequence in which our strategy would search for a proof is shown in Figure 3.5, where the sequence of application of rules is from top to bottom of the page and indentation indicates which proofs are sub–proofs of others.

3.3 Solving The Problems of Section 3.1

3.3.1 Solving Problem 4: Joining Relational Data

First let's represent the two tables of data as follows:

$$lecturer(dave, male, 42)$$
$$lecturer(mary, female, 21)$$
$$lecturer(phil, male, 64)$$

$$\begin{tikzcd}
 & {[b \to c] \vdash (a \ or \ b) \to (a \ or \ c)} \arrow[d, "\mathit{imp\_intro}"] & \\
 & {[a \ or \ b, b \to c] \vdash a \ or \ c} \arrow[dl] \arrow[d, "\mathit{or\_elim}"] \arrow[dr] & \\
\boxed{(a \ or \ b) \in [a \ or \ b, b \to c]} & {[a, a \ or \ b, b \to c] \vdash a \ or \ c} \arrow[d, swap, "\mathit{or\_intro\_left}"] & {[b, a \ or \ b, b \to c] \vdash a \ or \ c} \arrow[d, "\mathit{or\_intro\_right}"] \\
 & {[a, a \ or \ b, b \to c] \vdash a} \arrow[d, swap, "\mathit{immediate}"] & {[b, a \ or \ b, b \to c] \vdash c} \arrow[dl] \arrow[d, "\mathit{imp\_elim}"] \\
 & \boxed{a \in [a, a \ or \ b, b \to c]} \quad \boxed{(b \to c) \in [b, a \ or \ b, b \to c]} & {[b, a \ or \ b, b \to c] \vdash b} \arrow[d, "\mathit{immediate}"] \\
 & & \boxed{b \in [b, a \ or \ b, b \to c]}
\end{tikzcd}$$

Figure 3.3: Proof tree for $[b \to c] \vdash (a \ or \ b) \to (a \ or \ c)$

Given some sequent to be proved, of the form $\mathcal{F} \vdash P$:

- If P follows directly from the *immediate* rule then it is proved.
- If P is of the form A *and* B then use *and_intro*.
- If P is of the form A *or* B then first try to prove it using *or_intro_left* but, if that fails or you need another proof, then try using *or_intro_right*.
- If P is of the form $A \rightarrow B$ then use *imp_intro*.
- Otherwise, apply the *imp_elim* rule with the first member of $\mathcal{F}$ which is of form $P \leftarrow C_1$. If no proof can be found using this implication statement then take the next statement, $P \leftarrow C_2$, and apply the *implication* rule again. Repeat this procedure until either a proof is found or all the implication statements have been used.

Figure 3.4: Example proof strategy using rules of Figure 3.2

```
Problem : $[a \leftarrow b, a \leftarrow d \text{ or } e, e \leftarrow f \text{ and } g, f, g] \vdash a$
Agree to represent the set of assumptions using the symbol $\mathcal{F}$

Goal: $\mathcal{F} \vdash a$
    Apply implication given $a \leftarrow b \in \mathcal{F}$
    New subgoal: $\mathcal{F} \vdash b$
        No proof rule can be applied to this goal.
    So re–apply implication given $a \leftarrow d \text{ or } e \in \mathcal{F}$
    New subgoal: $\mathcal{F} \vdash d \text{ or } e$
        Apply or_intro_left
        New subgoal: $\mathcal{F} \vdash d$
            No proof rule can be applied to this goal.
        So apply or_intro_right
        New subgoal: $\mathcal{F} \vdash e$
            Apply implication given $e \leftarrow f \text{ and } g \in \mathcal{F}$
            New subgoal: $\mathcal{F} \vdash f \text{ and } g$
                Apply and_intro
                First new subgoal: $\mathcal{F} \vdash f$
                    Apply immediate given $f \in \mathcal{F}$
                Second new subgoal: $\mathcal{F} \vdash g$
                    Apply immediate given $g \in \mathcal{F}$
```

Figure 3.5: A proof using the strategy of Figure 3.4

$$\begin{array}{l} employee(dave, 2345) \\ employee(ken, 9324) \\ employee(mary, 6782) \\ employee(phil, 7934) \end{array}$$

We can also define axioms describing the type of data value for each column of our two data tables:

$$\begin{array}{rcl} lecturer(N, S, A) & \rightarrow & name(N) \ and \ sex(S) \ and \ age(A) \\ employee(N, I) & \rightarrow & name(N) \ and \ id(I) \end{array}$$

We no define the predicate $new(N1, S, A, N2, I)$ which is true when $lecturer(N1, S, A)$ and $employee(N2, I)$ are true and when $N1 = N2$ and $A < 50$ (the join condition).

$$lecturer(N1, S, A) \ and \ employee(N2, I) \ and \ N1 = N2 \ and \ A < 50 \ \rightarrow \ new(N1, S, A, N2, I)$$

The new data table is then the set of instances satisfying:

$$\exists N1, S, A, N2, I.new(N1, S, A, N2, I)$$

which is:

$$\begin{array}{l} lecturer(dave, male, 42, dave, 2345) \\ lecturer(mary, female, 21, mary, 6782) \end{array}$$

This simply constructed data set is not ideal - for one thing it contains duplicate entries for names. In the data component of Informatics 1 you will learn about more sophisticated operations on databases.

3.3.2 Solving Problem 5: Grammars as Logic

Let's consider sentence generation first. In another part of Informatics 1 you are learning about functional programming so one way to approach this problem is to describe a function to generate sentences. Let's describe this function in English (though still precisely) as follows, where the name of the function is on the left of each "=" and its computation (in terms of word sequences and other functions) is on the right. Concatenation is a function that joins two sequences in order.

sentence	=	concatenation of nounphrase with verbphrase
nounphrase	=	noun or concatenation of determiner with noun
verbphrase	=	verb or concatenation of verb with nounphrase

These correspond directly to the three grammar rules of Problem 5. In order to generate valid sequences of words, however, we need also to supply the functions giving us the appropriate noun, verb and determiner sequences.

noun	=	[dave] or [dust]
verb	=	[bites]
determiner	=	[the]

Applying the sentence function will now generate for us grammatically valid sequences like [dave,bites,the,dust][1].

We want to do this in predicate logic so the model of inference is different but we can still follow the same pattern of design. The basic difference is that instead of composing functions to generate valid sequences we will define predicates with a single argument for the "output" sequence and define the conditions for satisfiability of these sequences analogously to the computations of our functions. The logic versions of our grammar rules then are as shown below:

$$\begin{array}{rcl} \mathit{nounphrase}(S1) \ \mathit{and} \ \mathit{verbphrase}(S2) \ \mathit{and} \ \mathit{concatenate}(S1,S2,S) & \rightarrow & \mathit{sentence}(S1) \\ \mathit{noun}(S) \ \mathit{or} \ (\mathit{determiner}(S1) \ \mathit{and} \ \mathit{noun}(S2) \ \mathit{and} \ \mathit{concatenate}(S1,S2,S)) & \rightarrow & \mathit{nounphrase}(S) \\ \mathit{verb}(S) \ \mathit{or} \ (\mathit{verb}(S1) \ \mathit{and} \ \mathit{nounphrase}(S2) \ \mathit{and} \ \mathit{concatenate}(S1,S2,S)) & \rightarrow & \mathit{verbphrase}(S) \end{array}$$

Where $concatenate(S1, S2, S)$ is true if sequence $S1$ joined to sequence $S2$ in order is sequence S. Now we also need the specific noun, verb and determiner sequences as before:

$$\begin{array}{l} noun([dave]) \\ noun([dust]) \\ verb([bites]) \\ determiner([the]) \end{array}$$

We now can use our logical grammar to generate sentences (as we could with our functional definition). For example, if we attempt to satisfy:

$$\exists X.sentence(X)$$

then this is true for instances of X such as [dave,bites,the,dust] and [dave,bites,dust]. Unlike our functional definition, however, we can also test given sentences for their grammatical correctness (that is, we can parse as well as generate sentences) so we could satisfy goals such as:

$$sentence([dave, bites, the, dust])$$

This might seems strange given the functional description with which we started, since parsing a sentence would be like computing our functions "backwards" from a given output. Predicates, however, do not necessarily differentiate input from output - they only relate arguments - so in this case we can use the same logical axioms for both generation and parsing.

[1]It also generates grammatically valid but implausible sentences like [dust,bites,the,dave] - avoiding those is a deeper issue

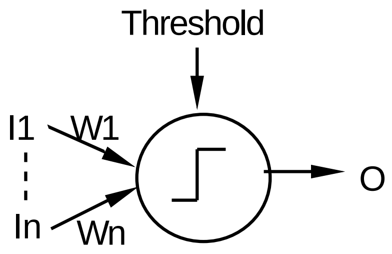
Chapter 4

Negation and Contradiction

This chapter extends the range of proofs we can perform to those involving proofs by contradiction and "proofs" in which negations of expressions are assumed when a proof fails.

4.1 Motivating Problems

Problem 6 *An artificial neural network is a system composed of a large, interconnected set of simple processing units. The neural network is trained by supplying it with stimuli and reinforcing responses that are appropriate and/or suppressing those which are inappropriate. In this way networks may learn to exhibit certain kinds of behaviours. A classic sort of processing unit in such networks is the perceptron, described in the diagram below:*



which shows a perceptron with a set of inputs $\{I1, \ldots, In\}$*; a threshold input; and an output,* O*. Inputs always have a value of* 0 *or* 1*. The output is calculated by the equation:*

$$\begin{array}{l} O = Th(W1 * I1 + \ldots + Wn * In - Threshold) \\ \text{where}: \;\; Th(E) = 0 \text{ if } E \leq 0 \\ \qquad\quad\;\;\; Th(E) = 1 \text{ if } E > 0 \end{array}$$

To what extent would it be possible to train a perceptron to reason logically? Could we train it to give the logically correct evaluation of $A \ and \ B$ *for all truth* *values of A and B for example? What about more complex expressions such as "exclusive or" ($(A\ or\ B)\ and\ not(A\ and\ B)$)?*

4.2 A Proof System Including Negation

To allow us to reason with negative information we extend the rules from Figure 3.2 with the following additional rules:

- contradiction: says that any expression C can be proved from some assumptions if we can prove that the assumptions are inconsistent (*i.e.* if we can prove $false$ from them).
- neg_intro: provides a proof of $not(A)$ from some assumptions if we can prove $false$ from those assumptions with the addition of A.
- neg_elim: says that any expression C can be proved from some assumptions if $not(A)$ appears in those assumptions and we can also prove A from those assumptions. Since the ability to deduce both $not(A)$ and A indicates that the assumptions are inconsistent, this is similar to the $contradiction$ rule.
- double_neg: provides a proof of an expression A from some assumptions if those assumptions yield a proof of $not(not(A))$ (*i.e.* that the negation of A is false).

The full set of rules is given in Figure 4.1.

An example of a proof by contradiction, using the rules from Figure 4.1, is shown in Figure 4.2. We want to prove the sequent: $[a, not(a\ and\ b)] \vdash not(b)$ and do this by adding b to the set of assumptions and proving that this is inconsistent (using the neg_intro rule). Therefore, since the expression, a, is inconsistent, $not(a)$ must be true.

4.3 The Closed World Assumption

The proof rules of Figure 4.1 allow us to perform proofs with negative information. These proofs are conservative in the sense that they allow us to conclude that a proposition is false only if it can be proved to be false. An alternative, often simpler, method is to accept that a proposition is false if it cannot be proved to be true. If we follow this route we may simplify our proof rules to those given in Figure 4.3. Here we have only one rule (cw_neg) to establish that a proposition is false and it succeeds if all attempts to prove the proposition fail. The integrity of this rule relies on the assumption that we have included in our axiomatisation of the problem all of the information relevant to the proposition - we have

Rule name	Sequent	Supporting proofs
equiv_elim	$\mathcal{F} \vdash A \leftrightarrow B$	$\mathcal{F} \vdash A \rightarrow B,\ \mathcal{F} \vdash B \rightarrow A$
and_intro	$\mathcal{F} \vdash A\ and\ B$	$\mathcal{F} \vdash A,\ \mathcal{F} \vdash B$
and_elim	$\mathcal{F} \vdash C$	$A\ and\ B \in \mathcal{F},\ [A, B \vert \mathcal{F}] \vdash C$
or_intro_left	$\mathcal{F} \vdash A\ or\ B$	$\mathcal{F} \vdash A$
or_intro_right	$\mathcal{F} \vdash A\ or\ B$	$\mathcal{F} \vdash B$
or_elim	$\mathcal{F} \vdash C$	$A\ or\ B \in \mathcal{F},\ [A \vert \mathcal{F}] \vdash C,\ [B \vert \mathcal{F}] \vdash C$
imp_intro	$\mathcal{F} \vdash A \rightarrow B$	$[A \vert \mathcal{F}] \vdash B$
imp_elim	$\mathcal{F} \vdash B$	$A \rightarrow B \in \mathcal{F},\ \mathcal{F} \vdash A$
contradiction	$\mathcal{F} \vdash C$	$\mathcal{F} \vdash false$
neg_intro	$\mathcal{F} \vdash not(A)$	$[A \vert \mathcal{F}] \vdash false$
neg_elim	$\mathcal{F} \vdash C$	$not(A) \in \mathcal{F},\ \mathcal{F} \vdash A$
double_neg	$\mathcal{F} \vdash A$	$\mathcal{F} \vdash not(not(A))$
immediate	$\mathcal{F} \vdash A$	$A \in \mathcal{F}$

Where: $\mathcal{F}$ is some list of assumptions.
A, B and C are well formed expressions.
$X \in Y$ denotes that X is an element of set Y.
$[X|Y]$ is a list with first element X and remaining elements Y.

Figure 4.1: Proof rules of Figure 3.2 extended with negation

$$\begin{tikzcd}
 & {[a, not(a\ and\ b)] \vdash not(b)} \arrow[d, "neg\_intro"] & \\
 & {[b, a, not(a\ and\ b)] \vdash false} \arrow[dl, "neg\_elim"'] \arrow[dr] & \\
\boxed{not(a\ and\ b) \in [b, a, not(a\ and\ b)]} & & {[b, a, not(a\ and\ b)] \vdash a\ and\ b} \arrow[dll] \arrow[d, "and\_intro"] \\
{[b, a, not(a\ and\ b)] \vdash a} \arrow[d, "immediate"] & & {[b, a, not(a\ and\ b)] \vdash b} \arrow[d, "immediate"] \\
\boxed{a \in [b, a, not(a\ and\ b)]} & & \boxed{b \in [b, a, not(a\ and\ b)]}
\end{tikzcd}$$

Figure 4.2: Proof tree for $[a, not(a\ and\ b)] \vdash not(b)$

Rule name	Sequent	Supporting proofs
immediate	$\mathcal{F} \vdash A$	$A \in \mathcal{F}$
and_intro	$\mathcal{F} \vdash A \text{ and } B$	$\mathcal{F} \vdash A,\ \mathcal{F} \vdash B$
or_intro_left	$\mathcal{F} \vdash A \text{ or } B$	$\mathcal{F} \vdash A$
or_intro_right	$\mathcal{F} \vdash A \text{ or } B$	$\mathcal{F} \vdash B$
or_elim	$\mathcal{F} \vdash C$	$A \text{ or } B \in \mathcal{F},\ [A\vert\mathcal{F}] \vdash C,\ [B\vert\mathcal{F}] \vdash C$
imp_elim	$\mathcal{F} \vdash B$	$A \rightarrow B \in \mathcal{F},\ \mathcal{F} \vdash A$
imp_intro	$\mathcal{F} \vdash A \rightarrow B$	$[A\vert\mathcal{F}] \vdash B$
cw_neg	$\mathcal{F} \vdash not(A)$	$\mathcal{F} \nvdash A$

Where: $\mathcal{F}$ is some list of assumptions.
A and B are well formed expressions.
$X \in Y$ denotes that X is an element of set Y.

Figure 4.3: A set of proof rules with closed world negation

described all we need to know about "the world", hence the term "closed world assumption".

Often it is effective to make the closed world assumption and many practical inference systems rely on it. It must, however, be handled with care. Suppose, for example, that we have the sequent $[p(a), q(b)] \vdash p(X) \text{ and } not(q(X))$. First we try to prove $p(X)$ and this is satisfied with $p(a)$ so we then satisfy $not(q(a))$ using the *cw_neg* rule. But suppose we chose to prove $not(q(X))$ first. This would fail (since $q(b)$ is provable) so in this case our sequent would be judged false. There are ways around such problems but we won't discuss these here.

4.4 Solving The Problems of Section 4.1

4.4.1 Solving Problem 6: Limits of Perceptrons

First let's define the behaviour of a perceptron using the equation supplied in Problem 6:

$$sum\_weights(In, S) \text{ and } E = S - T \text{ and } \left(\begin{array}{c} (E \leq 0 \text{ and } Out = 0) \\ or \\ (E > 0 \text{ and } Out = 1) \end{array} \right) \rightarrow perceptron(In, T, Out) \tag{4.1}$$

where $sum\_weights(In, S)$ sums the weighted inputs in the In set.

Let's now assume that we have defined a representative set of weights and thresholds. For example:

$$\begin{array}{cc} weight(-2) & threshold(-2) \\ weight(-1.9) & threshold(-1.9) \\ \vdots & \vdots \\ weight(1.9) & threshold(1.9) \\ weight(2) & threshold(2) \end{array} \tag{4.2}$$

We also can define the permitted values for any input:

$$input(0) \quad input(1) \tag{4.3}$$

Next we define what it means for the behaviour of a perceptron to be valid. We write $valid\_perceptron(Type, W1, W2, T)$ whenever a 2-input perceptron of a given type, P, gives a valid behaviour with the inputs' weights set to $W1$ and $W2$ and its threshold set to T. This is true if the weights and threshold are such that there is no combination of inputs for which the perceptron's behaviour does not give the correct answer for that type of behaviour.

$$\left(\begin{array}{l} weight(W1)\ and\ weight(W2)\ and \\ threshold(T)\ and \\ not \left(\begin{array}{l} input(I1)\ and\ input(I2)\ and \\ perceptron([(I1, W1), (I2, W2)], T, O)\ and \\ not(required(P, [(I1, W1), (I2, W2)], O)) \end{array} \right) \end{array} \right) \rightarrow valid\_perceptron(P, W1, W2, T) \tag{4.4}$$

The last step is to define the required behaviours. These are as follows:

$$\begin{array}{c} required(and, [(1, W1), (1, W2)], 1) \\ required(and, [(1, W1), (0, W2)], 0) \\ required(and, [(0, W1), (1, W2)], 0) \\ required(and, [(0, W1), (0, W2)], 0) \end{array} \tag{4.5}$$

$$\begin{array}{c} required(or, [(1, W1), (1, W2)], 1) \\ required(or, [(1, W1), (0, W2)], 1) \\ required(or, [(0, W1), (1, W2)], 1) \\ required(or, [(0, W1), (0, W2)], 0) \end{array} \tag{4.6}$$

$$\begin{array}{c} required(exor, [(1, W1), (1, W2)], 0) \\ required(exor, [(1, W1), (0, W2)], 1) \\ required(exor, [(0, W1), (1, W2)], 1) \\ required(exor, [(0, W1), (0, W2)], 0) \end{array} \tag{4.7}$$

This completes the definition of all the assumptions we wish to make about the problem. Let's call this set $\mathcal{S}$ and it contains all the expressions from 4.1 to 4.7 above. We obtain answers to the questions posed in Problem 6 by checking whether or not the following sequents are satisfiable:

$$\mathcal{S} \vdash valid\_perceptron(and, W1, W2, T)$$

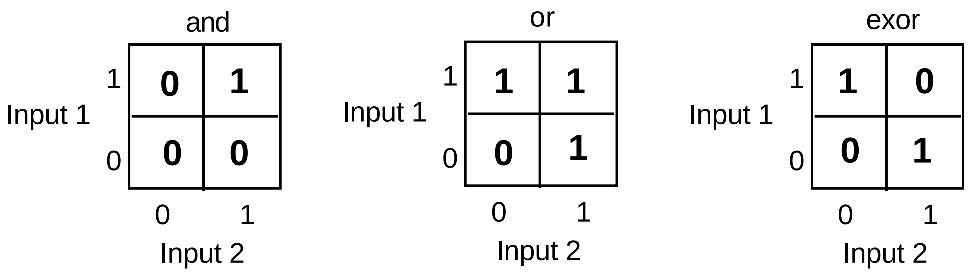
is satisfiable with many combinations of $W1$, $W2$ and T. For example there is a solution with $W1 = 0.2$, $W2 = 0.2$ and $T = 0.4$.

$$\mathcal{S} \vdash valid\_perceptron(or, W1, W2, T)$$

also is satisfiable with many combinations of $W1$, $W2$ and T. For example there is a solution with $W1 = 0.3$, $W2 = 0.3$ and $T = 0.1$.

$$\mathcal{S} \vdash valid\_perceptron(exor, W1, W2, T)$$

is not satisfiable. Does this, by itself, lead you to believe conclusively that a perceptron cannot represent exclusive or ($exor$)? Remember that our "proof" relied on searching for a satisfiable solution using the permitted weights and thresholds given in expression 4.2. These aren't all the possible weights and thresholds - for instance 0.15 isn't given although 0.1 and 0.2 are. We need, therefore, a more complete analysis of perceptron behaviour. One simple way to do this is to visualise the space of responses to input of the different perceptron types. The diagrams below depict the required response of each of the three types of perceptron (as defined in logic in expression 4.7 above). Each square corresponds to a different input combination and the number in a box is the appropriate output.



Now try to draw a straight line on each diagram that separates all its "0" squares from all its "1" squares. You can do this for the "and" and "or" diagrams but not for "exor". If you study the thesholding equation used in the perceptron relies on this linear separation.

Notice that what we have done in this example is to describe our problem in a particular way relying on specific assumptions. We used a from of proof based on these assumptions to explore the questions we had in mind but the proofs we had, although revealing, were not enough because we didn't entirely trust our assumptions. We then strengthened those assumptions by exploring the problem in a different style which relied on visual insight and examination of the thresholding equation. Combining different styles of formal analysis like this is common in complex problems. Experts in applied logic have reliable intuitions about which styles of representation and proof to apply to different problems.

Chapter 5

Proofs Using One Proof Rule: Resolution

In this chapter, we describe a technique for reducing the number of proof rules to just one: the Resolution rule of inference. In order for this rule to work, it is necessary also to simplify the expressions which we are dealing with. We first describe this simplification process and then explain how resolution works.

5.1 Motivating Problems

Problem 7 *It would be useful if we could have a very close relationship between logic and computer programs so that we could (among other things) use logic to describe precisely what we want a program to do and use proof to verify that it does those things. However, programmers using conventional programming language expect those languages to have a number of features that are not obvious in traditional logic:*

- *Programs written in the language determine sequences of tasks performed by the computer.*
- *The strategy used to execute programs in the language chooses tasks in a simple and predictable way that allows programmers to envisage the behaviour of the program before running it.*
- *Programs written in the language can be run efficiently.*
- *Programs written in the language can interact with other programs (such as databases and user interfaces).*

Could we find a route that takes us from some traditional logic to something that has the features of a traditional programming language? For example, could we apply logic to describing the proof strategy that we gave informally in Figure 3.4.

5.2 Normal Forms

In making various statements in logic we have made use of a variety of operators – namely not, and, or, $\rightarrow$ and $\leftrightarrow$. We know that all of these operators are not strictly necessary – for example, the $\rightarrow$ operator can always be "rephrased" in terms of the 'not' and 'or' operators by employing the equivalence: $(P \rightarrow Q) \leftrightarrow (not(P)\ or\ Q)$. By progressively rewriting an expression in this way, it turns out to be possible to represent any expression using *only* the 'and', 'or' and 'not' operators. Furthermore, it is possible to further simplify the expression by breaking it up at each 'and' symbol; converting the resulting propositions (which we shall call literals) into sets; and finally placing these sets themselves into a set. The result is therefore a set of (implicitly "and"ed) sets of (implicitly "or"ed) literals. The expression is then said to be in clausal form. The conversion procedure is described in Appendix B below[1]. To illustrate how this procedure works, suppose that we have the following expression:

$$(a\ and\ not(b) \rightarrow c)\ \ and\ \ a\ \ and\ \ not(c)$$

We can convert this expression into clausal form by applying the sequence of transformations shown in Figure 5.1. We end up with the final normalised expression:

$$[[not(a), b, c],\ [a],\ [not(c)]]$$

which comprises a set of three subsets. These subsets are implicitly conjoined (*i.e.* all three sets are must be true). The elements within each subset (where there is more than one element in a set) are implicitly disjoined – *e.g.* the first subset represents the expression $not(a)\ or\ b\ or\ c$. This example highlights some important points:

- The ordering of literals within each subset is unimportant, since they are all "or"ed together and any one would be sufficient to prove the truth of a subset.
- The ordering of subsets is unimportant because they are all "and"ed together and all must be proved to prove the truth of the expression.
- It would not be possible to reconstruct the original expression, with all the logical operators back in place, just by looking at its clausal form. The same clausal form could have been produced from many different combinations of logical operators.
- The clausal form is quite difficult for a human to read. By imposing uniformity on our representation we have sacrificed some of its intelligibility.

[1]The full conversion procedure is slightly more complex. We have simplified it in order to bring out the important features

Initial expression:
$(a\ and\ not(b) \rightarrow c)\ \ and\ \ a\ \ and\ \ not(c)$
Eliminate '$\rightarrow$' operator
$not(a\ and\ not(b)\ or\ c)\ \ and\ \ a\ \ and\ \ not(c)$
Drive negation in using $not(P\ and\ Q) \leftrightarrow (not(P)\ or\ not(Q))$
$(not(a)\ or\ not(not(b))\ or\ c)\ \ and\ \ a\ \ and\ \ not(c)$
Remove double negations using $not(not(P)) \leftrightarrow P$
$(not(a)\ or\ b\ or\ c)\ \ and\ \ a\ \ and\ \ not(c)$
Convert to sets
$[[not(a), b, c],\ [a],\ [not(c)]]$

Figure 5.1: A conversion to clausal form

5.3 Resolution

Resolution provides us with a simple way of establishing the truth of an expression in clausal form, given a set of assumptions also in clausal form. It relies upon a single proof rule, called the *resolution rule of inference*. A definition of this rule appears below.

The resolution rule of inference, permits the following procedure:

- If we have two sets of literals, R and S which are implicit disjunctions in clausal form.
- and if we can extract from R an element, P, leaving the remaining elements R'.
- and if we can extract from S an element, $not(Q)$, leaving the remaining elements S'.
- and if Q matches with P.
- then we can derive the new clause obtained by merging R' and S'.

As an illustration, consider the clauses $[not(a), b, c]$ and $[not(c)]$ from our running example. Applying the resolution rule to these two clauses allows us to "cancel out" c and $not(c)$ (in the process, substituting $dave$ for X), leaving us with a clause consisting of the remaining elements from both sets – namely: $[not(a), b]$. Proving the truth of an expression by resolution employs a technique similar to the "proof by absurdity" approach described in Section 4. A (simplified) restatement of this rule is that the conjunction $P\ and\ not(P)$ is inconsistent. If we convert this into clausal form then we can say that $[[P], [not(P)]]$ is inconsistent. Furthermore, we know that the resolution rule allows us to resolve $[P]$ with $[not(P)]$ to obtain the empty clause, $[]$. Therefore, if we can resolve any of the sets in our set of clauses to obtain the empty clause we have proved that our clauses are inconsistent. This allows us to prove the truth of a clause, given some set of assumptions, by *negating* it; proving that the negated clause, when combined with the set of assumptions, is inconsistent by resolving until an empty clause is obtained; and thus concluding that since the negation of the clause is inconsistent (and therefore false) the original, non–negated clause must follow from the truth of the set of assumptions.

An expression, F, in clausal form is true, given a set, A, of assumptions in clausal form if its negation, F', can be proved inconsistent with A by resolution. F' is inconsistent with A when an empty set of literals can be found by some sequence of applications of the resolution rule to the set formed by adding F' to A.

Figure 5.2 gives an example of a resolution proof applied to our running example. We set out to prove the proposition b. To do this we negate b; add it to the set of assumptions; obtain an empty clause using resolution – thus establishing that negating b causes a contradiction; and so conclude that b is true.

5.4 Solving The Problems of Section 5.1

5.4.1 Solving Problem 7: Logic Programs

Resolution eliminates the need to choose proof rules but this does not, of itself, make proof less complex or closer to traditional computation. We can, however, further simplify the way we write expressions. Recall that each expression in clausal form is a disjunctive set of propositions, so the set $[not(a), not(b), c]$ is equivalent to the expression $not(a)\ or\ not(b)\ or\ c$. Suppose that we restrict our attention to those expressions containing at most one non-negated proposition. We can convert these into implications with as single proposition as a conclusion by the following translation:

- Apply the equivalence $(not(X)\ or\ not(Y)) \leftrightarrow not(X\ and\ Y)$ to all the negated propositions in the clause. For example $not(a)\ or\ not(b)\ or\ c$ translates to $not(a\ and\ b)\ or\ c$.
- Apply the equivalence $(not(X)\ or\ Y) \leftrightarrow (X \rightarrow Y)$ to introduce an implication. For example, $not(a\ and\ b)\ or\ c$ translates to $(a\ and\ b) \rightarrow c$

Let's now write down the proof strategy of Figure 3.4 using clauses in this form. The trick is to invent a predicate that we shall call $satisfy(X)$ that is true when a theorem X is satisfiable using the proof rules of Figure 3.2 and the strategy of Figure 3.4. We then define a clause for this predicate for each case of the strategy, using the appropriate supporting proofs from Figure 3.2 as preconditions to the corresponding strategy clause. The resulting set of clauses is:

Given assumptions:

$$[[not(a), b, c],\ [a],\ [not(c)]]$$

To prove that b is true:
First, negate the goal to form the clause $[not(b)]$.
Then add the negation to the set of assumptions to form the following set of clauses (numbered for subsequent reference):

1. $[not(b)]$ (the negation of our goal)
2. $[not(a), b, c]$
3. $[a]$
4. $[not(c)]$

Now apply the resolution rule until an empty set is obtained.

$$\begin{tikzcd}
(1)\ [\underline{not(b)}] \arrow[dr] & & (2)\ [not(a), \underline{b}, c] \arrow[dl] & & \\
& [\underline{not(a)}, c] \arrow[dr] & & (3)\ [\underline{a}] \arrow[dl] & \\
& & [\underline{c}] \arrow[dr] & & (4)\ [\underline{not(c)}] \arrow[dl] \\
& & & [\,] &
\end{tikzcd}$$

We have now established that $not(b)$ is inconsistent with our original assumptions so we conclude that b is true.

Figure 5.2: An example of resolution

$$\begin{array}{rcl}
A \in \mathcal{F} & \rightarrow & satisfy(\mathcal{F} \vdash A) \\
satisfy(\mathcal{F} \vdash A)\ and\ satisfy(\mathcal{F} \vdash B) & \rightarrow & satisfy(\mathcal{F} \vdash A\ and\ B) \\
satisfy(\mathcal{F} \vdash A) & \rightarrow & satisfy(\mathcal{F} \vdash A\ or\ B) \\
satisfy(\mathcal{F} \vdash B) & \rightarrow & satisfy(\mathcal{F} \vdash A\ or\ B) \\
satisfy(\{A\} \cup \mathcal{F} \vdash B) & \rightarrow & satisfy(\mathcal{F} \vdash A\ \rightarrow\ B) \\
A \rightarrow B \in \mathcal{F}\ and\ satisfy(\mathcal{F} \vdash A) & \rightarrow & satisfy(\mathcal{F} \vdash B)
\end{array}$$

There is a simple way of using these clauses to apply the strategy they describe:

- Given some goal, find a clause with a conclusion that matches it (backtracking to choose another if this one doesn't yield a solution).
- If the clause has no precondition then the goal is true.
- If the clause has a precondition then take each of its constituent propositions and apply the strategy to it.

This strategy resembles the sort of problem decomposition done by programming languages. In fact it is essentially the strategy (and representation) used in the Prolog programming language.

Chapter 6

Proofs and State Change: Modal Logic

In Chapter 1 we explained how varieties of logic develop to tackle particular forms of representation and inference. We now give a basic example of one of these: a modal logic dealing with state change over time.

6.1 Motivating Problems

Problem 8 *The following is a communication protocol for a simple bank transaction system. There are three types of agents in this system: customers, branches and banks. The behaviour of each type, T, is constrained by the definition $a(T, X) :: D$ where X is an identifier for a specific agent of type T and D defines its behaviour. Behaviours are defined using message passing. $M \Rightarrow A$ means that message M is sent to agent A. $M \Leftarrow A$ means that message M is received from agent A. When an agent must perform two behaviours, $B1$ and $B2$, in sequence we write $B1$ then $B2$. When an agent can choose between two behaviours, $B1$ and $B2$, we write $B1$ or $B2$.*

$$\begin{array}{ll} a(customer, C) :: & buy \Rightarrow a(branch, B) \ \ then \\ & sell \Leftarrow a(branch, B) \ \ then \\ & a(customer, C) \end{array}$$

$$\begin{array}{ll} a(branch, B) :: & buy \Leftarrow a(customer, C) \ \ then \\ & check\_funds(C) \Rightarrow a(bank, X) \ \ then \\ & has\_funds(C) \Leftarrow a(bank, X) \ \ then \\ & sell \Rightarrow a(customer, C) \ \ then \\ & reconcile(C) \Rightarrow a(bank, X) \ \ then \\ & a(branch, B) \end{array}$$

$$\begin{array}{ll} a(bank, X) :: & ((check\_funds(C) \Leftarrow a(branch, B) \ then \\ & \quad has\_funds(C) \Rightarrow a(branch, B)) \ or \\ & reconcile(C) \Leftarrow a(branch, B)) \ then \\ & a(bank, X) \end{array}$$

Is this a good protocol from the banks point of view? Can you see any flaws in it?

6.2 A Basic Temporal Logic

Many varieties of temporal logic exist. The one chosen here has a style similar to the sequent based systems you saw earlier and it also operates directly on sequences of states. The proof rules for the logic are in Figure 6.1. These are similar in style to the other proof rules you have seen, except for the way in which the assumptions are given. Instead of a set of assumptions we base our proofs of a sequence of states, $\mathcal{S}$ and an integer index, J, that identifies the current state. If $J = 0$ it refers to the first state; if $J = 1$ it refers to the second state, and so on.

This allows us to talk about the truth of propositions relative to specific states, since $(\mathcal{S}, J) \vdash A$ means that A is true at state J of sequence $\mathcal{S}$. Once we can identify specific points then we can also make use of the temporal ordering of the sequence to talk about the truth of propositions forward or backward from a given point. $(\mathcal{S}, J) \vdash next(A)$ means that A is true in the state immediately following state J. $(\mathcal{S}, J) \vdash e\_future(A)$ means that A is true in some state following state J. $(\mathcal{S}, J) \vdash a\_future(A)$ means that A is true in all states following state J. We can reference past states similarly.

6.3 Solving The Problems of Section 6.1

6.3.1 Solving Problem 8: Protocol Verification

Suppose that we have a means of generating from the protocol of Problem 8 sequences of messages permitted by the protocol. An example of one such sequence (where $c1$ is a customer; $b1$ is a branch and $x1$ is a bank) is:

Rule name	Sequent	Supporting proofs
	$(\mathcal{S},J) \vdash A$	$access(J,\mathcal{S},\mathcal{F}),\ \mathcal{F} \vdash A$
	$(\mathcal{S},J) \vdash not(A)$	$(\mathcal{S},J) \not\vdash A$
	$(\mathcal{S},J) \vdash A\ and\ B$	$(\mathcal{S},J) \vdash A,\ (\mathcal{S},J) \vdash B$
	$(\mathcal{S},J) \vdash A\ or\ B$	$(\mathcal{S},J) \vdash A$
	$(\mathcal{S},J) \vdash A\ or\ B$	$(\mathcal{S},J) \vdash B$
	$(\mathcal{S},J) \vdash next(A)$	$(\mathcal{S},J+1) \vdash A$
	$(\mathcal{S},J) \vdash prev(A)$	$J > 0,\ (\mathcal{S},J-1) \vdash A$
	$(\mathcal{S},J) \vdash e\_future(A)$	$(\mathcal{S},K) \vdash A$ for some $K > J$
	$(\mathcal{S},J) \vdash e\_past(A)$	$(\mathcal{S},K) \vdash A$ for some $K < J$
	$(\mathcal{S},J) \vdash a\_future(A)$	$(\mathcal{S},K) \vdash A$ for all $K > J$
	$(\mathcal{S},J) \vdash a\_past(A)$	$(\mathcal{S},K) \vdash A$ for some $K < J$

Where: $\mathcal{S}$ is a set of states.
Where: $\mathcal{F}$ is a state (an element of $\mathcal{S}$).
A and B are well formed expressions.
$access(J,\mathcal{S},\mathcal{F})$ extracts the Jth state from $\mathcal{S}$.

Figure 6.1: A set of temporal proof rules

$$
\begin{array}{l}
message(a(customer,c1),\ buy\ \Rightarrow\ a(branch,b1))\\
message(a(branch,b1),\ buy\ \Leftarrow\ a(customer,c1))\\
message(a(branch,b1),\ check\_funds(c1)\ \Rightarrow\ a(bank,x1))\\
message(a(bank,x1),\ check\_funds(c1)\ \Leftarrow\ a(branch,b1))\\
message(a(bank,x1),\ has\_funds(c1)\ \Rightarrow\ a(branch,b1))\\
message(a(branch,b1),\ has\_funds(c1)\ \Leftarrow\ a(bank,x1))\\
message(a(branch,b1),\ sell\ \Rightarrow\ a(customer,c1))\\
message(a(customer,c1),\ sell\ \Leftarrow\ a(branch,b1))\\
message(a(branch,b1),\ reconcile(c1)\ \Rightarrow\ a(bank,x1))
\end{array}
$$

We could view message sequences as sequences of states, where each state corresponds to the sending of a message. Then we could use our temporal logic from Section 6.2 to test if given pathological properties hold of any possible state sequence. For example, one thing the bank might want to avoid is selling money to the same customer through two different branches before those branches had reconciled the transactions with the bank. This property expressed in our temporal logic (where $b1$ and $b2$ are two different branches) is:

$$
\begin{array}{lll}
(\mathcal{S},J) \vdash & message(a(customer,c1), sell <= a(branch,b1))\ and & \\
& e\_future(\ message(a(customer,c1), sell <= a(branch,b2))\ and & \\
& \qquad e\_future(\ message(a(branch,b1), reconcile(C) => a(bank,x1))\ and & \\
& \qquad\qquad next(message(a(branch,b2), reconcile(C) => a(bank,x1))))) &
\end{array}
$$

There are message sequences satisfying this property that can be generated from the protocol. One such sequence is:

$$\begin{array}{l}
message(a(customer, c1),\ buy\ \Rightarrow\ a(branch, b1))\\
message(a(branch, b1),\ buy\ \Leftarrow\ a(customer, c1))\\
message(a(branch, b1),\ check\_funds(c1)\ \Rightarrow\ a(bank, x1))\\
message(a(bank, x1),\ check\_funds(c1)\ \Leftarrow\ a(branch, b1))\\
message(a(bank, x1),\ has\_funds(c1)\ \Rightarrow\ a(branch, b1))\\
message(a(branch, b1),\ has\_funds(c1)\ \Leftarrow\ a(bank, x1))\\
message(a(branch, b1),\ sell\ \Rightarrow\ a(customer, c1))\\
message(a(customer, c1),\ sell\ \Leftarrow\ a(branch, b1))\\
message(a(customer, c1),\ buy\ \Rightarrow\ a(branch, b2))\\
message(a(branch, b2),\ buy\ \Leftarrow\ a(customer, c1))\\
message(a(branch, b2),\ check\_funds(c1)\ \Rightarrow\ a(bank, x1))\\
message(a(bank, x1),\ check\_funds(c1)\ \Leftarrow\ a(branch, b2))\\
message(a(bank, x1),\ has\_funds(c1)\ \Rightarrow\ a(branch, b2))\\
message(a(branch, b2),\ has\_funds(c1)\ \Leftarrow\ a(bank, x1))\\
message(a(branch, b2),\ sell\ \Rightarrow\ a(customer, c1))\\
message(a(customer, c1),\ sell\ \Leftarrow\ a(branch, b2))\\
message(a(branch, b1),\ reconcile(c1)\ \Rightarrow\ a(bank, x1))\\
message(a(branch, b2),\ reconcile(c1)\ \Rightarrow\ a(bank, x1))
\end{array}$$

Probably you found it difficult to guess simply by reading the protocol that it contains this potential flaw. Even if you guessed, it would have taken you a lot of effort to produce precisely an example of the problem. Normally protocols are more complex so this sort of task is daunting without mechanical systems to check them quickly.

Appendix A

Notation: Well Formed Expressions

The definition used in this course for a well formed expression in First Order Predicate Logic is given below.

Definition 1 *A well formed expression in the First Order Predicate Logic is defined as follows:*

1. *A constant can be any number or any unbroken sequence of symbols beginning with a lower–case letter.*
2. *A variable is any unbroken sequence of symbols beginning with an upper case letter.*
3. *A predicate is a term consisting of a functor, the predicate name, and an ordered set of 0 or more arguments. Predicates with 1 or more arguments are written in the style: $F(A_1, \cdots, A_N)$, where F is the functor and N is the number of arguments (or arity) of the predicate.*
4. *Predicate names must be constants.*
5. *Arguments may be either constants or variables. Variables may be quantified using either Universal or Existential Quantifiers, i.e. $\forall$ or $\exists$, respectively.*
6. *If P and Q are expressions, then the following are also expressions:*
   - $not\ P$
   - $not\ Q$
   - $P\ and\ Q$
   - $P\ or\ Q$
   - $P \rightarrow Q$
   - $P \leftrightarrow Q$
7. *If P is a well formed term then $\forall X\ \ P$ and $\exists X\ \ P$ are terms quantified over X. Any variables not quantified using either $\forall$ or $\exists$ are referred to as free variables in P.*

Appendix B

Conversion to Clausal Form

The algorithm for conversion of a FOPL expression to clausal form is as follows:

- Eliminate any $\leftrightarrow$ and $\rightarrow$ operators by rewriting it using the following equivalences (where the expression on the left side of the '$\leftrightarrow$' can be rewritten as the statement on the right side):
  - $(P \leftrightarrow Q) \leftrightarrow ((P \rightarrow Q)\ and\ (Q \rightarrow P))$
  - $(P \rightarrow Q) \leftrightarrow (not(P)\ or\ Q)$
- Convert to prenex form by moving all quantifiers to the left hand side of the expression, using the following equivalences (where X is a quantified variable and A and B are sub–expressions in the expression):
  - $not(\forall X\ A) \leftrightarrow (\exists X\ not(A))$
  - $not(\exists X\ A) \leftrightarrow (\forall X\ not(A))$
  - $((\forall X\ A)\ and\ B) \leftrightarrow (\forall X\ A\ and\ B)$
  - $((\exists X\ A)\ and\ B) \leftrightarrow (\exists X\ A\ and\ B)$
  - $(A\ and\ (\forall X\ A)) \leftrightarrow (\forall X\ A\ and\ B)$
  - $(A\ and\ (\exists X\ A)) \leftrightarrow (\exists X\ A\ and\ B)$
  - $((\forall X\ A)\ or\ B) \leftrightarrow (\forall X\ A\ or\ B)$
  - $((\exists X\ A)\ or\ B) \leftrightarrow (\exists X\ A\ or\ B)$
  - $(A\ or\ (\forall X\ A)) \leftrightarrow (\forall X\ A\ or\ B)$
  - $(A\ or\ (\exists X\ A)) \leftrightarrow (\exists X\ A\ or\ B)$
- Eliminate all existential quantifiers as follows:
  - Those outside the scope of any universal quantifier are replaced with Skolem constants (arbitrary names which don't appear anywhere else). For example, the expression: $\exists X\ happy(X)$ might be converted into $happy(someone)$, where '$someone$' is a Skolem constant representing some arbitrarily selected person.

  - Those inside the scope of any universal quantifier are replaced with Skolem functions, whose arguments are the universally quantified variables within whose scope the existential occurs. For instance, the expression: $\forall X\ \exists Y\ hates(X, Y)$ might be converted into $hates(X, enemy\_of(X))$ where $enemy\_of(X)$ is a Skolem function which obtains some enemy for any X.

- Remove all universal quantifiers, on the understanding that all the variables are implicitly universally quantified. The expression is now free of quantifier symbols.

- Drive negation in to the individual predicates, using the equivalences:

  - $not(P\ and\ Q) \leftrightarrow (not(P)\ or\ not(Q))$
  - $not(P\ or\ Q) \leftrightarrow (not(P)\ and\ not(Q))$
  - $not(not(P)) \leftrightarrow P$

- Distribute disjunction over conjunction, using the equivalence:

  - $(P\ or\ (Q\ and\ R)) \leftrightarrow ((P\ or\ Q)\ and\ (P\ or\ R))$

  The expression is now said to be in conjunctive normal form.

- Convert each group of disjunctions into a set of atomic expressions and place each of these sets into an implicitly conjoined set of disjunctions.

- Rename all variables so that the same variable name doesn't appear in different disjunctive sets.