Formal systems, logic and semantics

Daniel Richardson,
Department of Computer Science, University of Bath.
email : masdr@bath.ac.uk

September 27, 2006

Contents

1 Formal Systems **5**
1.1 Introduction 5
1.2 Problem Solving 5
1.3 Design principles of formal systems 8
1.4 Summary 9
1.5 Related ideas:Confluence and termination 9

2 String Rewriting Systems **11**
2.1 Rules and derivations 12
2.2 Language generation: Grammars 13
2.3 Problems (which may be discussed in tutorials) 14
2.4 Applications 16
2.5 Other uses of string rewriting systems 16
2.6 Summary 16

3 Term languages, substitution and unification **17**
3.1 Function Spaces 17
3.2 Term languages 17
3.3 Interpretation of a term language 19
3.4 Parse Trees 20
3.5 Substitution 22
3.6 Valuations 23
3.7 Properties of term languages 25
3.8 Unification 26
3.9 The Unification Algorithm 27
3.10 Problems, which may be discussed in problems class 30
3.11 Applications 30
3.12 Summary 31

4 Predicate Logic **33**
4.1 Introduction 33
4.2 Truth values and predicates 33
4.3 Variables and types 34
4.4 Translation from informal to formal language 35
4.5 First order languages 35
4.6 Syntax, substitution 37
4.7 Free and bound variables 37
4.8 Semantics for first order languages 38
4.9 Examples : L_N and L_R 41
4.10 Uses of these ideas in computing 42
4.11 More Substitution 43
4.12 Other kinds of language and other logics 44
4.13 Logic Games 45
4.14 Normal Forms 45
4.14.1 CNF and DNF 46
4.14.2 Prenex Normal form 48
4.14.3 Skolem form 49
4.14.4 Clausal form 51
4.15 Problems 53
4.16 Summary 54
4.17 Revision Problems. Clausal form 55
4.18 Solutions 55
4.19 Examples and Solutions 56

5 Semantic Tableaux **59**
5.1 Introduction 60
5.2 Semantic tableau rules 63
5.3 Some advice about the use of the rules 66
5.4 How much use is this system? 67
5.4.1 Gödel completeness theorem 67
5.4.2 Unsolvability and intractability 70
5.5 Problems 70
5.6 Satisfaction Games 72
5.7 Revision Problems 72
5.8 Solutions 73
5.9 Summary 74

6 Other formal deductive systems for first order logic **75**
6.1 Deduction by Resolution and Unification 75
6.2 The Sequent Calculus 76
6.3 Other deductive systems 78

7 A Formalisation of Mathematics: ZF set theory 79

8 Gödel Incompleteness theorems 85

9 Logic programming (prolog) 87
9.1 Headed Horn clauses . . . 87
9.2 Predicates, facts, constants, variables in prolog . . . 90
9.3 Programs in prolog . . . 93
9.3.1 Comments in Prolog . . . 96
9.4 How prolog works . . . 96
9.5 Programs with just facts . . . 97
9.6 Backtracking . . . 99
9.7 Warning: problems with prolog . . . 100
9.8 How to write simple prolog . . . 101
9.9 Examples . . . 102
9.10 Circular definitions and Recursion . . . 105
9.11 The Cut, and negation . . . 106
9.11.1 The Cut and fail combination . . . 109
9.12 Family Tree . . . 111
9.13 Lists, Member(X,Y), Append(X,Y,Z) . . . 111
9.14 Sorting . . . 113
9.15 Equivalence relations; and how to find your way out of a labyrinth . . . 114
9.15.1 First Attempt . . . 115
9.15.2 Second attempt . . . 116
9.15.3 Third attempt . . . 117
9.15.4 Fourth attempt . . . 117
9.15.5 Fifth attempt . . . 118
9.15.6 Sixth Attempt . . . 119
9.15.7 Testing and Proof . . . 119
9.16 Labyrinth Program . . . 119
9.17 Language Recognisers in Prolog . . . 120
9.18 kitchen table propositional theorem prover . . . 121
9.19 Problems . . . 122
9.20 Input and Output in Prolog . . . 124
9.20.1 write(X) . . . 125
9.20.2 read(X) . . . 125
9.20.3 Debugging and tracing . . . 125
9.20.4 Assertions and Retractions . . . 126
9.21 Arithmetic in Prolog . . . 126
9.22 Typed Prolog . . . 128
9.23 Summary of this chapter . . . 129

10 Multisorted, higher order languages, and non classical logics 131

10.1 Notation for types . . . 132
10.2 Problems with types . . . 132
10.2.1 Solutions to Types . . . 133
10.3 Non classical logics . . . 133
10.4 Proof Assistants . . . 134

11 Semantics and Specification for Programs 135

11.1 Programming Language Semantics . . . 135
11.2 Denotational Semantics . . . 136
11.3 Correctness and Completeness of software . . . 138
11.4 Formal Methods . . . 138

12 References 139

Chapter 1

Formal Systems

1.1 Introduction

There is a long tradition, going back at least to the axiomatic geometry of the ancient Greeks, of trying to model human thought processes by formal systems. Computers, allowing global communications and requiring development of standard languages, encourage standard formal representations of human thinking, reckoning, scheming, calculating and cogitation. A great deal of work is currently being done in this area, which includes and exceeds the bounds of the subject usually called artificial intelligence. This book attempts to describe one of the most successful of the formalisms: formal deductive systems.

1.2 Problem Solving

When we are absorbed in the business of trying to solve a problem, we are usually too occupied to observe patterns in what we are doing, or to make generalisations about how the process works. However, since we wish to cooperate with others in problem solving and also to use computers to help us, and to represent our insights computationally, some generalisation is called for.

Consider the following list of problem solving situations.

1. A person trying to solve the Rubic cube puzzle.
2. Two people playing chess.
3. A mouse, apparently trying to find its way out of a maze.

4. A child learning to speak.
5. A mathematician trying to prove a theorem, which he or she is intuitively sure is correct.
6. A computer programmer developing a program.

Perhaps you will agree that there are some common features in this list. It seems that in some ways we can take the mouse in the maze as typical of problem solving agents. Problem solving seems to be a process of exploration within some definite space of possibilities. Within this space of possibilities, some moves are allowed and some are prohibited. What kind of general representation can we make for this situation?

We will call a position in the space of possibilities a configuration. We hope to be able to write down the configuration at each moment. The configuration at a moment in a game of chess, for example, might be the arrangement of pieces on the board, together with an indication of who should move next. We think of each configuration as a point, or a position in the space of possibilities, and the moves are transitions from one configuration to another.

Assume that we represent each configuration by a data structure in some set D of data structures, appropriate to the situation we are considering. (Here, a data structure could be a number or a string of characters, or a vector or a matrix, or a tree labelled with data structures, or a directed graph labelled with data structures, for example.) To say that D is a set of data structures does not commit us to very much. But we do want to insist that the items in D are unambiguous and finite and can somehow be written down. The items in D are syntactic; they are pieces of syntax, arrangements of symbols.

We must also say how the moves within D are constrained. What transitions are allowed, and what forbidden? We will suppose that we have some collection of rules, which we will denote R, which determine which transitions in D are allowed. If X and Y are in D, we will write

$X \Rightarrow_R Y$

to mean that the rules R allow a transition from X to Y in one step.

Definition 1.1 *A formal system is (D, R) where D is a set of data structures, and R is a set of rules which determine which transitions between objects in D are allowed.*

Definition 1.2 *If (D, R) is a formal system, let $X \Rightarrow_R Y$ mean that X and Y are in D and that the transition from X to Y is allowed by the rules R.*

Definition 1.3 *If (D, R) is a formal system, let*

$X \Rightarrow_R^* Y$

mean that there exists a finite sequence $X_1, ..., X_n$ of items in D so that $X = X_1 \Rightarrow_R X_2 \Rightarrow_R X_3 \Rightarrow_R ... \Rightarrow_R X_n = Y$.

$X \Rightarrow_R^* Y$ means that it is possible to get from X to Y by a finite sequence of steps which are allowed by the rules R. A sequence of this kind,

$X_1 \Rightarrow_R X_2 \Rightarrow ... \Rightarrow_R X_n$

is called a derivation, of length $n - 1$, in the formal system (D, R).

So $X \Rightarrow_R^* Y$ means that there exists a derivation from X to Y. Notice that derivations of length 0 are allowed. So $X \Rightarrow_R^* X$ is always true.

If the set of rules can be understood from the context, we will leave off the subscript R.

We require that $X \Rightarrow_R Y$ be decidable. That is, given X and Y in D, we should be able to determine whether or not $X \Rightarrow_R Y$. (Otherwise, what good are the rules?) Moreover, we expect that it should be *easy* to decide $X \Rightarrow_R Y$. The decision as to whether or not $X \Rightarrow_R Y$ must only depend on X and Y as syntactic objects. That is, R is not allowed to refer to any meaning associated with the data structures in D. For example, we would not allow a rule which said that $X \Rightarrow Y$ if and only if " a move from X to Y is a good idea". The reason for this is that R is required to be unambiguous. In this sense, R is formal.

Although we insist that $X \Rightarrow_R Y$ should be easy, we expect that $X \Rightarrow_R^* Y$ may be relatively hard. Our claim is that the core of problem solving has to do with this relation $X \Rightarrow_R^* Y$.

People have probably always invented and puzzled over formal systems, as described above. Part of their charm is the contrast between the typical simplicity of $X \Rightarrow Y$ and the typical complexity of $X \Rightarrow^* Y$. Even when D is finite, $X \Rightarrow^* Y$ is often extremely interesting, as in chess, for example.

There are two reasons for the development of formal systems (D, R). One reason is their inherent fascination and beauty; and the other is that a formal system may have quite simple rules, and yet may capture the combinatorial features of an infinite and surprising reality. We are trying to do something which is so ambitious that it is almost preposterous: the hope is to capture some aspects of our own thought processes, by modelling them with formal systems. We are trying to make language mechanisms which behave like thought. The design and study of such formal systems is the primary motivation of the field of study known as artificial intelligence. We are trying to catch some aspects of the action of intelligence within formal systems.

Remarks. If you like geometric pictures, you may visualise a formal system as a collection of points, some of which are connected by arrows. The points are called configurations and each one is identified by a finite piece of syntax. In this sense, a formal system may be considered to be a certain kind of labelled, possibly infinite, graph. If you know what a Turing machine is, you will see that a Turing machine is a special kind of formal system. If you know what a (mathematical) *category* is, you will see that if (D, R) is a formal system, then D together with $\Rightarrow^*$, is a certain kind of category.

Examples: Chess is a formal system, with configuration being chess positions. In general two person games of strategy can usually be modelled as formal systems. Interaction between a computer system and an environment can also be modelled in this way. We may consider logical argument or disputation as a formal system, a sort of game in which only certain moves are allowed. A non deterministic finite automaton is a formal system; so is a pushdown automaton, or a deterministic or non deterministic Turing machine; the configurations specify the complete situation of the system at a moment of time. The lambda calculus is a formal system; the data structures are the lambda terms, the transitions are alpha and beta reductions. In linear algebra, we may take matrices as configurations, and row operations as allowed transitions; then $A \Rightarrow^* B$ means that matrix A can be changed into matrix B by a sequence of row operations.

We may also model populations of agents by formal systems. A configuration gives the internal state of each agent, and any other appropriate information such as, for example, position. We might, for example if we were trying to model group behaviour of animals or humans, allow each agent to observe its immediate surroundings, and to perform certain actions in relation to other agents, for example offering to shake hands, bowing, following, avoiding, bluffing, mating, killing, eating, etc. $X \Rightarrow Y$ means that the rules allow configuration X to change to configuration Y in one step. Of course, we may also consider independently acting processes in computer systems to be agents, whose behaviour is constrained by rules.

1.3 Design principles of formal systems

A specification for a formal system is a statement, hopefully a precise statement, of what we want the formal system to do.

There are accepted criteria for judging how well a formal system (D, R) meets its specification.

1. Completeness. (D, R) does everything which the specification requires.

2. Correctness. (D, R) does nothing which the specification prohibits.
3. Simplicity. This is also called the Occam's razor condition. One statement of this very old condition of good intellectual design is: *Thou shalt not multiply entities unnecessarily* . The formal system should fulfil its specification, or come close to fulfilling its specification with minimal complexity.
4. Naturalness. The formal system should be in accord with intuition.

All of these conditions are important. In many cases, in most interesting cases, they may also, in part, contradict one another. Design of a good formal system usually involves a trade off between these conditions.

It will not have escaped the thoughtful reader that the relationship between a formal system and its specification is like the relationship between a scientific theory and the phenomena which it is designed to explain. One difference is that science has traditionally believed that its theories are physically true. Although, when a formal system is extremely good, as with the formal grammars explained in the next chapter, it is hard not to suspect that there could be some physical basis for it, most people think that formal systems are inventions rather than discoveries.

1.4 Summary

A formal system is a set of data structures together with a set of transformations which act on them.

Important examples are formal grammars for generating formal languages, formal proof systems in predicate logic, the lambda calculus which formalises symbolic computation, and populations of autonomous agents. Also all games.

1.5 Related ideas:Confluence and termination

We will say that a formal system (D, R) is terminating if there is no infinite derivation in it. This means that no matter what choices are made, every derivation must eventually stop.

A formal system (D, R) is *confluent* if whenever $A \Rightarrow^* B$ and $A \Rightarrow^* C$ there exists D so that $B \Rightarrow^* D$ and $C \Rightarrow^* D$. This means that if we started at configuration A and computed to find C, and computed in another way to find B, there is some way to bring the two computations back together to get D.

A formal system (D, R) is *locally confluent* if whenever $A \Rightarrow B$ and $A \Rightarrow C$ there exists D so that $B \Rightarrow^* D$ and $C \Rightarrow^* D$.

For formal systems used for computation (such as the lambda calculus) confluence is a very useful property.

Theorem 1.1 *A formal system is confluent if it is locally confluent and terminating.*

Chapter 2

String Rewriting Systems

Suppose we start with a language, such as English or Urdu or C or L_{ZF}. Each utterance in the language is a finite string of symbols. Some strings are in the language and some are not. We may think of the language as a set of strings of symbols. It seems that if people speak or recognise a language this means that they somehow know the definition of this possibly infinite set of strings of symbols. How can these complicated sets be defined? The best contemporary answer to this puzzle is that although languages may be infinite they can be generated by finite sets of rules.

String rewriting systems are formal systems whose data structures are strings of symbols, and whose transformations are rewriting rules, as defined below.

Let Σ be a finite alphabet of symbols. Let Σ^* be the set of all finite length strings which can be formed using symbols from Σ. The elements of Σ^* will be called words on Σ.

A Σ rewriting rule is an expression of the form

$$\alpha \leadsto \beta$$

where α and β are in Σ^*. Note that Σ^* contains the empty word.

A rule $\alpha \leadsto \beta$ means that *it is allowed* to replace α by β in any context. Note that here and in the following we are using Greek letters as linguistic variables for strings of words in Σ^*.

Definition 2.1 *A Σ rewriting system is a finite list of Σ rewriting rules:*

$\alpha_1 \leadsto \beta_1$

.

.

.

$\alpha_n \rightsquigarrow \beta_n$

A Σ rewriting system is meant to be a set of rules which defines a computational process on strings of symbols from Σ. It turns out that any computational process can be seen in this way.

2.1 Rules and derivations

Definition 2.2 *Let* $P =$
$\alpha_1 \rightsquigarrow \beta_1$
.
.
.
$\alpha_n \rightsquigarrow \beta_n$
be a Σ *rewriting system, and let* A *and* B *be words in* Σ^**.*

We will say that B *can be obtained from* A *in* P *in one step, and write this as*

$$A \Rightarrow_P B$$

if A *is of the form* $W_1\alpha_iW_2$*, and* B *is of the form* $W_1\beta_iW_2$*, where* $\alpha_i \rightsquigarrow \beta_i$ *is in* P*. (Note that* W_1 *and/ or* W_2 *can be empty.)*

When the rewriting system P is understood, we will leave off the subscript P.

As with all formal systems, once we have got a clear notion of one step derivation, we can iterate this to obtain derivations of any finite length.

Definition 2.3 *A derivation in rewriting system* P *is a list of words* $A_1, A_2, ..., A_k$ *so that* $A_1 \Rightarrow A_2 \Rightarrow ... \Rightarrow A_k$*.*

Definition 2.4 *We will say that there is a derivation which goes from* A *to* B*, and write this as* $A \Rightarrow^* B$ *if there exists* $A_1, ..., A_n$ *so that*

$$A = A_1 \Rightarrow A_2 \Rightarrow ... \Rightarrow A_n \Rightarrow B$$

We allow derivations of length zero. So we will always have $A \Rightarrow^* A$.

Example 2.1 *Suppose* $\Sigma = \{a, b\}$*, and* P *is*
$aba \rightsquigarrow b$
$aaa \rightsquigarrow$
Let W *be the word* $aaaababbbbababab aaaababa$*.*

In order to apply P *to* W*, we attempt to match left hand sides of the rules of* P *with parts of* W*. There are 6 matches with* aba *and four matches with* aaa*. So there are ten words* B *with* $W \Rightarrow B$*.*

Problem 2.1 *Start with W, as given in the example above. Apply the given rules in any order until you terminate. Why did you eventually terminate? Do you get the same result at termination, no matter in which order you apply the rules?*

2.2 Language generation: Grammars

Definition 2.5 *We will say that a word W is terminal in rewriting system P if there is no word Z so that $W \Rightarrow Z$.*

Definition 2.6 *A grammar is a rewriting system P together with an initial word I. Such a grammar (P, I) generates the language*
$\{W : I \Rightarrow^* W, W terminal\}$.

Example 2.2 *Consider the language, $L = \{ab, aabb, aaabbb, ...\}$. The language consists of the set of words which are formed by a string of a's followed by an equally long string of b's. There are many different grammars which could be used to generate this language. One would be:*
$X \rightsquigarrow ab$
$X \rightsquigarrow aXb$.
The starting word is X. A typical derivation would be:
$X \Rightarrow aXb \Rightarrow aaXbb \Rightarrow aaaXbbb \Rightarrow aaaabbbb$
Note that the string on the end of the derivation is terminal and is in the language L.

Try to prove that the language generated by this grammar is exactly L. Note that there would be two parts to such a proof. We would want to show that every terminal string generated by the grammar is in L; and we would also want to show that every string in L could be generated by this grammar. So even at this simple level there is a completeness and correctness problem.

$\Box$.

2.3 Problems (which may be discussed in tutorials)

Problem 2.2 *Define a variable to be any upper case letter followed by any string of letters and digits. Write a grammar which generates this language.*

Problem 2.3 *Let* $\Sigma = \{X, ', 0\}$.
Describe the Σ *language* L_1 *generated by grammar* (P, X) *where* P *is:*
$X \rightsquigarrow X' \mid 0$
(It is cheating in this case to describe the language as the language generated by the grammar.)

Problem 2.4 *Describe the language* L_2 *generated by the grammar* (P, X) *where* P *is*
$X \rightsquigarrow X' \mid (X + X) \mid 0$
Show that $((0'' + 0''') + 0')'''$ *is in* L_2*, by giving a derivation of it from the initial word* X*. You may, to shorten the amount you have to write, use* $X \equiv> Y$ *to mean that* Y *can be obtained from* X *by doing several non overlapping single steps in parallel.*

Problem 2.5 *Consider languages* L *such that 0 is in* L *and whenever a string* α *is in* L *the string* α' *is also in* L*. Give two examples of such languages. How is* L_1 *special among such languages.*

Problem 2.6 *Give a grammar for the language which consists of strings of b's separated by single a's.*

Problem 2.7 *Here are some samples from a language.*

1. $(((p \vee q) \wedge \neg r) \vee (p \wedge q))$
2. $\neg\neg(p \wedge q)$

Give a grammar to generate this language. Also give a description of the language (i.e. a specification), which does not use the idea of a grammar.

Problem 2.8 *Here are some samples from a language.*

1. (xy)
2. $(x(yz))$
3. $((xy)z)$
4. $(\lambda x.(xz))$

5. $((\lambda x.x)(\lambda y.y))$
6. $((x(\lambda y.(yy)))z)$

Give a grammar, as simple as possible, which generates all of these. Although the answer to this problem is simple once you see it, it is a harder problem than the earlier ones. (Of course, if I have made any typing mistakes in the sample, this problem is more or less impossible.)

Problem 2.9 *Let P1 be a programming language, consisting of statements. A sequence of statements is also a statement. Statements are closed under the usual IF and WHILE constructions. Assignments are statements. Variables are all of type integer. Expressions include the variables and the usual notation for the integers, and are closed under +, -, *. Give a grammar (as simple as possible consistent with this description) for the expressions and also for the statements of P1.*

Example 2.3 *L_{ZF}, the language of Zermelo-Fraenkel set theory, is generated by the grammar (P, S), where P is as follows.*

$S \rightsquigarrow (S \vee S) \mid (S \wedge S) \mid (\neg S) \mid (S \rightarrow S) \mid (S \leftrightarrow S) \mid (\forall Variable)S \mid (\exists Variable)S \mid Variable = Variable \mid Variable \in Variable$

where $Variable$ is as defined above. Almost all of contemporary mathematics can be written in this famous formal language L_{ZF}. This is a very successful example of formalisation. Note that the fact that mathematics can be expressed in L_{ZF} does not imply that we should use L_{ZF} as a working language of mathematical thought. On the other hand, the existence of a formalisation is extremely useful, as a last resort, in case of disagreement about meaning.

As an example of translation from mathematical English into L_{ZF}, consider the statement: For any sets X and Y there exists a set Z so that the elements of Z are either elements of X or elements of Y or both.

In L_{ZF}, this would be:

$(\forall X)(\forall Y)(\exists Z)(\forall W)(W \in Z \leftrightarrow (W \in X \vee W \in Y))$

A *formal language* is one which can be defined by a grammar, as described above.

2.4 Applications

Programming languages are formal languages. For example, the syntacticly correct programs of C constitute a formal language. It is quite important to have a compact and unambiguous description of a programming language, and formal grammars give us exactly that.

Formal languages are also used as a language for specification of programs. This raises the possibility that part of the process of development, from specification to working program, could be formalised and automated.

Formal languages are also used in the foundations of mathematics. Part of the process of formalisation of a field of mathematics is the development of an appropriate formal language.

Some people think also that natural human languages have some common formal core. The idea is that there is something like a formal language which is common to humanity, and that natural languages are obtained from this by a process of transformation.

2.5 Other uses of string rewriting systems

In this chapter we have concentrated on grammars and languages. However, string rewriting can be used also as a basis for computation on strings of symbols. This will be explored later.

2.6 Summary

In string rewriting systems, the data structures are strings of symbols, and the transformations are string rewriting rules.

Given a string rewriting system P, we will say $A \Rightarrow_P B$ if string B can be obtained from string A by one application of the rules in P. We will say $A \Rightarrow^* B$ if strong B can be obtained from string A by a sequence (possibly of length zero) of applications of the rules.

From an initial word I, a language is generated within a string rewriting system P. The language is

$\{W : I \Rightarrow_P W, W/terminal\}$.

The string rewriting system together with the initial word is called a *grammar* for the language.

Chapter 3

Term languages, substitution and unification

3.1 Function Spaces

A *type* is a set together with a family of operations defined on the set. An example is the integers, with the usual operations of addition, subtraction and multiplication.

If A and B are types, $A \to B$ will mean the type of all functions from A to B. So $f : N \to N$ means that f is a function from the natural numbers to the natural numbers.

If D is any set, $f : D^k \to D$ means that f is a function of k arguments from D which takes values in D.

Let D be a set. Let F_D be $\cup_{n=0}^{\infty}(D^n \to D)$. F_D will be called the function space of D. It contains all the functions which apply to arguments in D and give values in D. Since D may be infinite, we may not be able to give names to all the objects in F_D. What we want therefore is a convenient notation for some of these functions.

3.2 Term languages

Term languages occur very often in mathematics and computing. The terms in a term language are generally used to denote functions in some function space.

Programming languages typically have term languages as expressions.

A term language is obtained by taking a collection of function symbols and constants and a set of variables and forming all possible correctly formatted expressions from the variables and the function symbols, allowing arbitrary nesting of expressions.

The collection of function symbols and constants is called the *signature* of the term language.

Function symbols are also called function names, or function identifiers. The function symbols are not functions, they are just names, pieces of syntax.

It is assumed here that we know, for each function symbol in a signature, the number of arguments which it should have. This number is called the *arity* of the function symbol. It is also assumed that we know, for each function symbol, the correct format for writing the application of it to its arguments.

For example $+$ usually has arity 2, and $sin(X)$ has arity 1. A function name $f(X, Y)$ would have arity 2.

We will consider constants as function symbols of arity 0. So a *signature* for a term language is just a set of function symbols.

We can begin with any signature. We do not determine in advance what the function symbols mean.

Example 3.1 *The function names $sin(X)$ and $cos(X)$, together with the variables generates a term language. One of the terms in this language is $sin(sin(cos(sin(cos(sin(Yag))))))$. This term is meant to stand for some function of the variable Yag. If we give sin and cos their usual interpretations, this term is determined as a real valued function of a real variable. But sin and cos could be interpreted in other ways, and the term language is just the set of terms, as syntactic things in themselves, without any special assumptions about the interpretation of the function symbols.*

Term languages are languages with a fairly simple semantics. So this is a reasonable place to try out some ideas.

Example 3.2 *Suppose $f(X, Y)$, and $s(X)$ are function symbols. The term language generated by these function symbols consists of all terms which can be written using any variables, and these two function symbols and composition. So, for example $f(s(s(W)), f(U, f(B, s(W))))$ is a term in this term language. Note that the terms are syntactic things, strings of symbols. We have not yet decided what these terms mean, if anything. Note also that we do have a strict notion of syntactic correctness for these terms, even without being sure what they stand for. For example $f(Z, W($ must be*

a typing mistake; it is not a term. Similarly, $s(A, B, C)$ might be a term in some other term language, but not in this one.

One way to define a term language is by a grammar. For the above example, the grammar would be:

$< term > \rightsquigarrow f(< term >, < term >) \mid g(< term >) \mid < Variable >$

where, $< Variable >$ means an upper case letter followed by a string of letters and digits.

Example 3.3 *Suppose we take signature $[+, *, ', 0]$ with arities 2,2,1,0. An example of a term is $(X+(Y*X)'')$. Another term is $0''''''''''''''''''''''''''''''''$.*

3.3 Interpretation of a term language

Definition 3.1 *Let $\mathcal{A}$ be a term language, with signature $\Sigma = (f_1, ..., f_k)$. An interpretation of $\mathcal{A}$ is $(D, \mathbf{f_1}, ..., \mathbf{f_k})$, where D is a set and $\mathbf{f_1}, ..., \mathbf{f_k}$ are in the function space of D and, for $i = 1, ..., k$, if the arity of f_i is n_i then $\mathbf{f_i} : D^{n_i} \to D$.*

In other words, an interpretation of a term language gives a specific meaning in some function space to each function symbol in the signature.

Given an interpretation, each term is interpreted as a function in the function space F_D. We can regard the term language as a notation for some of the functions in the function space. It would also be correct to say that the functions give a meaning to the term language. Notice that an interpretation is specified by giving a meaning to the function symbols in the signature; the meaning of all the terms is then automatically generated.

D is called the domain of the interpretation. This can be any set. The functions can be any totally defined functions, which take arguments which are tuples of the right arity in D to match the signature, and take values in D.

For each term language, you should imagine a spectrum of all possible interpretations. D could be a finite set, or the natural numbers, or the reals.

D could even be $\mathcal{A}$ itself. That is, the domain could be the terms! Within this domain, we can interpret a function symbol f_i, of arity n_i as a function $\mathbf{f_i} : A^{n_i} \to A$ by letting $\mathbf{f_i}$ map terms $T_1, ..., T_{n_i}$ into the term which is simply obtained by writing f_i applied to $T_1, ..., T_{n_i}$. For example $f(X, Y)$ applied to Z and $g(W)$ is literally $f(Z, g(W))$. In this interpretation a term is regarded both as a syntactic object and also as a definition of a function from terms to terms. We will call this the literal interpretation of a term language.

Example 3.4 *Consider signature* $[f, g, s, a]$ *with arities 2,1,1, 0. One of the terms would be*

$f(s(X), f(Y, f(Y, Z)))$.

Even though we do not yet know what f, g, s *are, we know how to read this notation. This is a compound function which has been put together from basic parts* f, g, s.

Example 3.5 *Consider signature* $[(cons\ X\ Y), (car\ X), (cdr\ X), nil]$ *with arities 2, 1, 1, 0. One of the terms in this language is*

$(cons\ (cdr\ X)\ (cons\ Y\ (cons\ Y\ Z)))$.

Note that no predefined meaning is attached to these function symbols. In fact this term language is essentially the same as the term language in the previous example, even though the function names and the format conventions are dissimilar.

In all the following examples of term languages, variables will begin with upper case letters. Any name which starts with a lower case letter is assumed to be a function name.

So jane is a constant, Jane is a variable, and lub(X,Y) is a term, in which the function lub of arity 2 is applied.

3.4 Parse Trees

Consider the term language with grammar

$< T > \leadsto f(< T >, < T >) \mid g(< T >) \mid < Variable >$,

with $< Variable >$ defined as usual. A typical term in this language is $f(g(X), f(Y, g(X)))$.

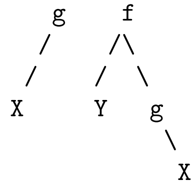
Problem 3.1 *Write down three different derivations of the term above in the grammar of its term language.*

It should be clear that although there are different ways of deriving this term, all the derivations are essentially the same. We could try to make this clear by combining together several parallel steps:

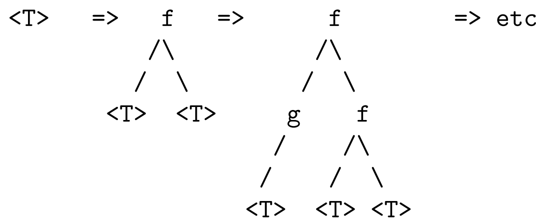
$< T > \equiv> f(< T >, < T >) \equiv> f(g(< T >, f(< T >, < T >))) \equiv> f(g(X), f(Y, g(< T >))) \equiv> f(g(X), f(Y, g(X)))$.

We may also display the derivation as a tree:

```
      f
     /\
    /  \
```



These trees are called parse or expression trees. We can think of them as growing downwards from the initial string.



The parse tree of a term shows how the term is derived.

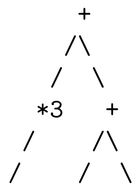
Definition 3.2 *The* outermost operator *of a term is the function name which appears at the top of the parse tree of the term.*

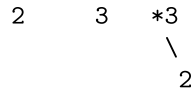
Continuing with this example, suppose we give a semantics to the term language by specifying that $f(X, Y)$ is addition and $g(X)$ is multiplication by 3 over the integers.

It should be clear that each term represents a function, which takes values in the integers. That is, if we assign integer values to the variables, we should get an integer value returned.

We can see how this works on the parse tree:

Suppose we have $X = 2, Y = 3$.





As we evaluate the tree, the numbers percolate upwards, until we get the final value at the top.

In conclusion, grammars generate parse trees from the root downwards; semantics evaluates and interprets a parse tree, starting from the frontier and finishing with the return value at the root.

The outermost operator of a term is the one chosen first in the derivation, and evaluated last.

3.5 Substitution

Substitution of terms for variables inside other terms is a very natural operation. If N and M are terms in a term language and x is a variable, we will use the notation $N[x := M]$ to mean the result of substituting M for all occurrences of x in N. We can also do several substitutions simultaneously.

Definition 3.3 *If N is a term, and x_1,..., x_n are variables, and M_1,..., M_n are terms, then*

$$N[x_1 := M_1, ..., x_n := M_n]$$

means the result of simultaneously *replacing all occurrences of x_1,..., x_n by terms M_1,..., M_n in term N.*

Notice that the insistence on simultaneity is important.

Example 3.6 $(X + Y)[X := (X * Y), Y := X] = ((X * Y) + X)$

Problem 3.2 *Suppose A, B, C are terms. Which of the following are correct?*

1. $(A[x := B])[y := C] = (A[y := C])[x := B]$
2. $A[x := B, y := C] = A[y := C, x := B]$
3. $(A[x := B])[y := C] = A[x := B, y := C]$

Give counterexamples where possible.

A new formal system has popped up here. The data structures are terms. The transformations are the substitutions. It seems that everybody who studies mathematics or computing already knows about substitutions. Are there any hard mathematical problems in this area?

Problem 3.3 *Suppose we start with a term and apply a substitution to it, and then apply another substitution to the result. Can the final result be obtained from the initial term by one substitution? A simple yes or no answer will not suffice here. If the answer is obvious, it should also be easy to give a proof. Try this.*

Definition 3.4 *If σ_1 and σ_2 are substitutions, let $\sigma_1 \circ \sigma_2$ be the result of first applying σ_1 and then applying σ_2. This will be called the composition of σ_1 and σ_2.*

3.6 Valuations

Suppose $\mathcal{A}$ is a term language with signature $(f_1, ..., f_k)$. Let $D, \mathbf{f_1}, ..., \mathbf{f_k})$ be an interpretation of $\mathcal{A}$. A valuation from D is a function which gives values in D to a subset of the variables of $\mathcal{A}$. If $X_1, ..., X_n$ are variables in $\mathcal{A}$, and $a_1, ..., a_n$ are objects in D, we will use the notation

$[X_1 := a_1, ..., X_n := a_n]$

for the valuation which gives value a_1 to variable X_1, ... and value a_n to variable X_n.

The support of a valuation is the set of variables which are given values by it.

Let τ be a term of $\mathcal{A}$ and set v be a valuation whose support includes all the variables which occur in $\mathcal{A}$.

We will use the notation

$\tau(v)$

for the value of τ in D which results from setting the values according to v, and using the given interpretation of the function symbols. If the support of v does not include all the variables of τ, then $\tau(v)$ will be undefined. In general, given an interpretation, terms denote partially defined functions of valuations.

Example 3.7 *Let E be the term language with function symbols $+, -, *$, with arity 2, and sin, cos, exp, abs with arity 1, and $0, 1, \pi$ with arity zero. The natural interpretation of E has the the real numbers as its domain and the usual function and constant meanings. So $exp(X)([X := \pi])$ is the value of $exp(X)$ when $X = \pi$, i.e. e^π. This value is a real number, not an approximation. Note that in this example there is no possibility of writing*

down names for all the objects in the domain. Of course, we can write down names for some of the objects, for example

$exp(1) - exp(3 + exp(2))$, *or* $exp(exp(\pi))$.

Unfortunately, in the present state of mathematics, we do not know how to decide whether or not any two given names of this kind denote the same real number.

We expect that substitution and evaluation will interact in a nice way.

Problem 3.4 *Let τ be a term and α a substitution, and v a valuation whose support includes all the variables which occur in τ or in α. We can first substitute, and then evaluate. This gives us*

$\tau\alpha(v)$

Show that there exists a valuation w which, applied directly to τ, gives the same result.

Definition 3.5 *Let $\mathcal{A}$ be a term language and $I = (D, \mathbf{f_1}, ..., \mathbf{f_k})$ an interpretation of A. If τ_1 and τ_2 are two terms of A, we will write*

$\models_I \tau_1 = \tau_2$

if

$\tau_1(v) = \tau_2(v)$.

for all valuations v from I whose support includes all the variables of τ_1 and τ_2.

$\models_I \tau_1 = \tau_2$ will be read: Under interpretation I, terms τ_1 and τ_2 are semantically equal. For example, under the usual interpretation of multiplication over the reals, $X * Y$ and $Y * X$ are semantically equal. Of course, $X * Y$ and $Y * X$ are not syntactically equal. We can find an interpretation in which $X * Y$ and $Y * X$ have different meanings.

A given interpretation induces a notion of semantic equality (in that interpretation) on a term language.

Problem 3.5 *Show that for any term language, $\mathcal{A}$, there exists an interpretation, I in which the notion of semantic equality in I is the same as literal syntactic identity in $\mathcal{A}$.*

Of course we can decide syntactic equality in a term language. Two terms are syntactically the same only when they are identical. But sometimes semantic equality is difficult to recognise. For example, in the term language E mentioned earlier, interpreted in the reals in the natural way, semantic equality is *undecidable*. That is, there can be no algorithm to decide whether or not two terms of E, interpreted over the reals, have the same meaning.

There are even difficulties, as mentioned earlier, in deciding semantic equality of constants standing for real numbers.

Problem 3.6 *Assume sqrt(x) means the square root of x, defined for non negative reals.*

What should be the value of x after the following?

```
If (sqrt(9 + 4*sqrt(2)) = 1+ 2*sqrt(2))
then x:=1
else x:=0;
```

Problem 3.7 *How could a computer* correctly *solve the previous problem?*

Semantics is supposed to give a standpoint for criticism. The above problem is intended to convince you that an honest semantics should consider interpretations in the real numbers, and not just in an approximation. On the other hand, there are severe difficulties in this area.

Don't worry if you can not see how to solve the problem above. No one else is sure how to do this either. After all, the test could have been much more difficult, e.g.

(sqrt((112+70*sqrt(2)) +(46+34*sqrt(2))sqrt(5))=

(5+4*sqrt(2))+(3+sqrt(2))*sqrt(5) ?

How to make a computational representation of the real numbers which is an improvement on bounded precision floating point numbers is a subject of current research in computing.

3.7 Properties of term languages

A term says how to compute something. Compound terms are built up from simple ones by function application. If we compare a term language with the way in which functions are built up from one another in a typical programming language, such as C, we see that term languages have a beautiful simplicity and clarity. In evaluation of a parse tree, all subtrees are independent, and can be evaluated in parallel. In any expression to be evaluated, we can replace a term by its value without changing the final result. There is a horrible piece of jargon for this. It is called referential transparency. Term languages have referential transparency. As far as I can understand, the original idea here is that we can, so to speak, look right through the term to see its value in the domain without any loss of information; we can suppose that the term and its value are the same. On the other hand, in a C program, a function may depend on the state of the computer memory, and this may in turn be altered by the act of evaluating

the function. C functions have side effects. So we do not have referential transparency for C functions in general.

Because of referential transparency, computation in a term language is naturally parallelisable. It is also relatively feasible to prove properties of such computation. The main idea of functional programming is to try to get some of these beneficial properties for programming languages.

3.8 Unification

Definition 3.6 *Suppose N and M are terms and σ is a substitution. We will say that σ unifies M and N if $N\sigma = M\sigma$.*

Example 3.8 $(X \wedge (Y \vee Z))$ *and* $((\neg W) \wedge V)$ *are unified by substitution* $[X := \neg horsefeathers, W := horsefeathers, V := (Y \vee Z)]$ *.*

Note that there are other ways of unifying these two terms. In fact the substitution given has an annoying excess specificness. Where did the $horsefeathers$ come from? We can say that although this substitution is a unifier, it is not the most general unifier, since a constant $horsefeathers$ has been unnecessarily used.

We are already starting to use the concept of most general unifier, but we do not yet have a definition of it. We will have to return to this question later. We do need to clarify it. You may be able to do this yourself at this point if you think very hard about it. A unifier is a substitution which unifies two terms. What is a most general unifier?

There is another problem here. Given two terms, how do we decide whether or not they have a unifier, or a most general unifier? If there is such, how can we find it? We will also return to this later.

For the moment, here is a useful image. Suppose we have two terms, and we are looking for a unifier. A good way to begin is to superimpose one parse tree over the other. If they contradict each other, there is no unifier. If they do not contradict each other, we should be able to see the minimal substitutions which will make the two trees the same.

Example 3.9 *Consider the following list of terms:*

1. $f(g(X), f(Y, Z))$
2. W
3. $f(W, f(g(U), V))$
4. $g(T)$

5. $f(Y, Z)$

The first two are unified with the substitution $[W := f(g(X), f(Y, Z))$. *The first and the third can also be unified. To see this, superimpose the parse trees. The* f *matches at the top. So now we need to unify* W *and* $g(X)$ *to get the left hand subtree, and having done that we will need to unify the right hand subtrees. You can see now why we need a most general unifier. We wish to unify and also leave ourselves as much room to manoeuvre as possible. (We still do not have a definition!) But you can see what to do, namely* $[W := g(X)]$. *Fortunately there are no* W *occurrences in the right hand subtree. We need now to unify* $f(Y, Z)$ *and* $f(g(U), V)$. *Once again, the* f *at the top matches. Possibly you can finish this part of the example yourself?*

Note that the attempt at unification fails if you get non matching function names at the top of the parse tree, as in items 4 and 5 above. There is another way in which unification can fail. Consider terms 1 and 5. We get $[Y := g(X)]$. *We are then confronted with the alarming problem of trying to unify* Z *and* $f(g(X), Z)$. *Since terms are finite, we observe that no term can ever be a proper subterm of itself; so there is no way that* Z *and* $f(g(X), Z)$ *can be the same.*

Definition 3.7 *A most general unifier of two terms* T *and* S *is a substitution* α *which is a unifier, and which also has the property that any other unifier* β *can be obtained from* α *by a further substitution. That is, if* α *is a most general unifier and* $T\beta = S\beta$, *then there must exist a substitution* γ *so that*

$(T\alpha)\gamma = T\beta$

and

$(S\alpha)\gamma = S\beta$

Example 3.10 *Suppose we wish to unify* $f(X, g(U, h(U, s), U))$ *and* $f(g(U, V, s), g(W, Y, t))$

We see that X *must be the same as* $g(U, V, s), U)$, *and* $g(U, h(U, s), U)$ *must be the same as* $g(W, Y, t)$. *Thus* W *and* U *must be the same, and* Y *must be* $h(U, s)$, *and* U *must be* t. *So we end up with*

$\alpha = (X := g(t, V, s), Y := h(U, s), W := t, U := t)$

3.9 The Unification Algorithm

This will apply to two lists of terms $(S_1, ..., S_n)$ and $(T_1, ..., T_m)$. The algorithm will either produce a most general unifier of these two lists, or it will return FAIL. It should fail if and only if no unifier is possible. If we wish

to unify two individual terms, we will just put these two terms in two lists of length one and apply the algorithm to this pair of lists.

The algorithm will be described recursively.

Basis. This is a collection of simple cases.

1. (Simple Case 1)

   $unify((S_1, ..., S_n), (T_1, ..., T_m)) = FAIL$

   if $n \neq m$,(since, of course, a substitution can not alter the number of terms in a list).

2. (Simple Case 2)

   Another easy case occurs when $(S_1, ..., S_n)$ and $(T_1, ...T_m)$ are already equal. In this case no unifier is necessary. We let

   $unify((S_1, ..., S_n), (S_1, ..., S_n)) = []$, the empty substitution.

3. (Simple Case 3)

   Another simple case occurs when $n = m = 1$, and one of S_1 or T_1 is a variable, say X.

   We can assume $unify((S_1), (T_1)) = unify((T_1), (S_1))$, so swapping of arguments is allowed.

   Swap arguments if necessary to that $S_1 = X$. We have already dealt with the earlier simple cases, so we can assume that T_1 is not X. We check to see if X occurs inside T_1. If it does than no unification is possible, since, for any substitution α, $(S_1)\alpha$ is strictly smaller than $(T_1)_\alpha$. In this case,

   $unify((S_1), (T_1)) = FAIL$.

   The other possibility is that X does not occur in T_1. In this case

   $unify((S_1), (T_1)) = [X := T_1]$.

Recursive Step .
There are several cases in the recursive step.

1. (Recursive Case 1)

   First suppose that $n = m$ and n and m are greater than one, and none of the simple cases apply.

   We are trying to unify

   $(S_1, ..., S_n)$

   and

$(T_1, ..., T_n)$

We first find $unify((S_1), (T_1))$. If this fails, then $unify((S_1, ..., S_n), (T_1, ..., T_n))$ also fails. Suppose, however that unification of S_1 and T_1 succeeds, with most general unifier α. Thus $(S_1)\alpha = (T_1)\alpha$.

We apply α to the rest of the list and continue recursively. That is, we find

$$unify((S_2, ..., S_n)\alpha, (T_2, ..., T_n)\alpha)$$

If this fails, we return $FAIL$. If this succeeds with most general unifier β, then

$unify((S_1, ..., S_n), (T_1, ..., T_n)) = \alpha \circ \beta$, the composition of β and α.

2. (Recursive Case 2)

   Finally, suppose $n = m = 1$, and none of the earlier cases apply. In this case, we want to unify S_1 and T_1. We can suppose that neither is a variable, since this case was dealt with earlier, and we can also suppose that they are not already equal. If one of them is a constant, then return $FAIL$. Suppose that neither is a constant.

   It must happen that both S_1 and T_1 are compound terms obtained by application of a function symbol to argument lists.

   $S_1 = f(A_1, ..., A_j)$

   $T_1 = g(B_1, ..., B_k)$

   for some function symbols f and g.

   If f and g are not the same, then unification fails, and we return $FAIL$. If f and g are the same, then we call the unification algorithm recursively on the argument lists. That is,

   $unify((S_1), (T_1)) = unify((A_1, ..., A_j), (B_1, ..., B_k))$

□
It is claimed that:
Theorem
1) The unification algorithm, as given above, always eventually terminates.
2) If the algorithm terminates with a unifier this unifier is a most general unifier.
3) If the algorithm terminates and returns FAIL, then there is no unifier.

It is not *obvious* that any of these claims are true. The reader is invited to try the algorithm on a few cases to see if it works. This may or may not inspire confidence.

Exercise. Try to prove the first claim above, that the algorithm always terminates. We can start to think about this by a dialogue between a defender and a critic of the algorithm. Defender: the algorithm always terminates. Critic: No it does not. Defender: OK, then give me a simplest problem in which it does not terminate. Critic: Define simplest. Defender: Minimum number of variables, and subject to that the minimum length. *Now, whatever the critic responds, the defender has a strategy to win the argument. Consider each case in turn, using the assumed minimality of the critic's response....*

The proofs of the other two parts can be done in the same way. For example, for part 3) the critic would have to claim that there was a case in which the algorithm did terminate and returned fail, but that there was a unifier. The defender makes the critic give a minimal explicit case, and then the defender demolishes the critic. Advice to the reader: you ought to think about this proof. You ought to be able to prove the correctness of the algorithm in specific problems.

3.10 Problems, which may be discussed in problems class

Problem 3.8 bish bash buf, bish bash bish bash buf buf, bish bash bish bash bish bash buf buf buf *: three utterances in some language. Specify the language with some exact description. Then give a grammar which you believe generates this language.*

Problem 3.9 *Give a grammar for the smallest term language which contains all the usual variables, and also 0 and 1, and also +, -, *, written in the usual way, with arity 2. In which cases can you leave off some brackets without ambiguity? Give the parse tree for* $((X+W)*(Y-(Z*(X+1))))$. *Suppose* X,Y,Z,W *have values 2,3,4,5 respectively; use this to attach a value to every node of the parse tree.*

Problem 3.10 *Find the composition of the substitution* $[X := X+Y, Y := Z]$*, followed by* $[Z := Z^2, Y := Z]$

3.11 Applications

Unification is an essential part of prolog, which will be discussed later.

3.12 Summary

Term languages are the sets of expressions which can be built up from an initial signature of function names, and a set of variables. Within programming languages, term languages occur as sets of expressions. An interpretation of a term language is given by giving a domain, D, and by interpreting each function name as a function defined over that domain. For the sake of simplicity, we are, at this stage, assuming that the values returned by our functions are also in the domain D. The domain may be a set, such as the real numbers, which is important in applications, but for which we do not have an adequate notation.

Each expression in a term language has a unique parse tree. Given an interpretation, terms in a term language can be evaluated by working up the associated parse trees.

If I is an interpretation and τ_1 and τ_2 are terms $\models_I \tau_1 = \tau_2$ means that τ_1 and τ_2 have the same meaning (or are semantically equal) in interpretation I.

We have defined substitution as an operation on terms.

We have defined a unifier of two terms to be a substitution which makes them the same. A unifier α of two terms A and B is a most general unifier if any unifier of A and B can be obtained as α followed by some other substitution.

The unification algorithm, given two terms A and B, either produces a most general unifier for A and B, or, in case no unifier exists, fails.

Chapter 4

Predicate Logic

4.1 Introduction

In English and other human natural languages complex sentences and properties are built up by combining simpler ones using certain logical operators. There are surprisingly few of these operators. On the other hand, they are quite ambiguous in natural speech.

In order to begin formalisation of reasoning, we consider, in this chapter, formal versions of the logical operators.

4.2 Truth values and predicates

In the following we will make the classical assumption that there are only two truth values True and False, which we will write as T and F. (Note that in other versions of logic, this assumption is not made.)

Definition 4.1 *A predicate is a function whose codomain is the truth values* $\{T, F\}$.

So predicates are usually represented as statements with variables in them. When the variables are given values the statement should evaluate as either true or false.

So, for example, $uncle(X, Y)$, which says that X is an uncle of Y, defines a predicate. For any particular X and Y, this statement will either be true or false.

The number of variables which occur in a predicate and which can be instantiated in this way is called the *arity* of the predicate. So, for example,

$red(X)$, meaning that X is red, has arity 1; and $between(X, Y, Z)$, meaning that X is between Y and Z, has arity 3.

There are a number of formats for writing predicates. We will often put the predicate name first and follow it by the arguments enclosed in brackets, as in $uncle(X, Y)$. This is called prefix format. On the other hand, many commonly occurring predicates use other formats. For example

$X = Y$, meaning that X and Y are the same

$X < Y$, meaning that X is less than Y.

Predicates of arity 1, such as $even(X)$, are sometimes called *properties*.

Predicates can be combined together using logical operators. We will consider the following list of operators:

1. and, written symbolically as $\wedge$
2. or , written symbolically as $\vee$
3. not, written symbolically as $\neg$
4. implies, written symbolically as $\rightarrow$
5. if and only if, written symbolically as $\leftrightarrow$
6. for all, written symbolically as $\forall$
7. there exists, written symbolically as $\exists$.

In the following, we will assume that you know the truth tables for $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$, and that you have some working familiarity with the quantifiers $\forall$, and $\exists$. If you are not happy with this, you should revise.

Note especially that the truth table for $p \rightarrow q$ is the same as the truth table for $\neg p \vee q$. This is somewhat counterintuitive. (In this case Occam's razor has been given priority over naturalness.) We do assume that the truth value of a compound expression should be determined from the truth values of its constituents, without knowing anything else about possible entanglement of meaning. In other words we assume referential transparency for the logical operators. Since some false propositions imply true ones, we have to agree that $(F \rightarrow T)$ evaluates as T; and similarly $(F \rightarrow F)$ evaluates at T. Continuing with this, we get $p \rightarrow q$ evaluates as true when p is false, or when q is true.

4.3 Variables and types

We can of course give types to variables, and write quantifiers in the form $(\forall X : N)A(X)$ for example. Everything we do below can also be done in

these types languages. However, in order to try to keep the notation fairly simple, we will in this chapter assume that all variables have the same type, which will not be declared. Each variable X ranges over some fixed domain D. The domains are the same for all the variables.

4.4 Translation from informal to formal language

We can now take a piece of text written in an informal language, try to identify what the basic predicates are, and represent the text as a logical combination of the basic predicates.

Example 4.1 *Anyone who loves Jane is brave. There seem to be two predicates at work here* $loves(X, Y)$*, which says that* X *loves* Y*, and* $brave(X)$*, which says that* X *is brave. We want to say that if* X *loves Jane, no matter who or what* X *is, then* X *must be brave.*

$(\forall X)(loves(X, jane) \rightarrow brave(X))$

Note that the variable X is not given a type. It just ranges over some universe. All we know about jane is that jane is also in the same universe.

Example 4.2 Z *is the union of* X *and* Y*. We could of course take this as a basic predicate, or it could be expressed using equality and a function symbol for union. We could also express this in terms of set membership. In this case we want to say that any element of* Z *is either an element of* X *or an element of* Y*, and vica versa.*

$(\forall W)(W \in Z \leftrightarrow (W \in X \vee W \in Y))$

One of the values of formalisation is that it uncovers ambiguities in the original statement. It turns out that people are able to maintain mutual misunderstandings over remarkably long periods of time without realizing it. There is no cure for this, but formalisation is one proven technique for diminishing this type of ambiguity.

It is quite hard to learn to do this type of translation correctly. Try some of the examples below.

4.5 First order languages

We saw earlier how a collection of function symbols generates, in a natural way, a term language. Once we give an interpretation to the basic function symbols, all the terms in the term language take on meaning as functions.

Suppose now that we are given a list of function symbols and a list of predicate symbols. As before the function symbols will generate a term language. We can write down predicates by applying the predicate symbols to the terms in the term language. These are called atomic formulae, since there is no way to break them down logically. The atomic formulae can then be combined together, using the logical operators. The resulting set of expressions are called formulae. A *first order language* is the set of formulae generated from a given list of function symbols and list of predicate symbols.

In first order languages, all the variables have the same type. In the interpretations of these languages there is only one universal domain, over which all variables range. Later on we will see more complicated languages: multisorted languages in which there are several different types of variable, and higher order languages in which we have variables for sets or functions as well as for objects.

The lists of function symbols and predicate symbols which specify a first order language is called its *signature*.

The term language always includes the variables. We can therefore allow the list of function symbols to be empty, since we still get infinitely many terms. But we can not use an empty predicate symbol list, since in that case we would not get any atomic formulae or formulae. Since equality is used so often in mathematics and computing, we will assume that the predicate symbol list always has $=$ in it.

An example of a first order language is L_{ZF}, given earlier. In this case, the term language is just the set of variables. The list of predicate symbols is $[=, \in]$, both with arity 2.

In any first order language, the formulae are generated from the atomic formulae in the same way, shown in the following grammar.

$< S > \rightsquigarrow (< S > \wedge < S >) \mid (< S > \vee < S >) \mid (\neg < S >) \mid (< S > \rightarrow < S >) \mid (< S > \leftrightarrow < S >) \mid (\forall < Variable >) < S > \mid (\exists < Variable >< S > \mid < Atomicformula >$

Example 4.3 *L_N, the first order language of arithmetic, has the signature with function symbols $[+, *, ', 0]$, and predicate symbol $[=]$. Function symbols $+$ and $*$ are written in the usual way, so that for example, $((X+Y)*(X*Z))$ is a term. The intended meaning of X' is $X+1$, the successor of X. The arity zero predicate symbol, 0, is intended to mean zero.*

An example of an atomic formula would be

$(0'' * X) = Y$.

An example of a formula would be

$(\exists Y)(0'' * Y = X)$.

Under the usual interpretation, this means that X is even.

4.6 Syntax, substitution

A first order language consists of a set of terms, a set of atomic formulae and a set of formulae. When such a language is interpreted, the terms return values in some domain D, but the formulae (including the atomic ones) return truth values. When an interpretation is given, the variables are all assumed to range over the common domain of the interpretation. It would make sense to substitute one term inside another one. It would also make sense to substitute a term for a variable inside a formula. For example, in L_N, we could have

$(\exists W)(X + W' = Y)$

and substitute $Z * Y$ for Y to obtain

$(\exists W)(X + W' = Z * Y)$

On the other hand, it would not make sense to try to apply a function symbol to a formula. In general formulae may not be substituted for variables in terms, since the value returned by a formula is a truth value, and the function symbols are defined on domain values, not truth values.

4.7 Free and bound variables

Definition 4.2 *In an expression of the form* $(\forall \mathbf{X})\mathbf{wff}$ *or* $(\exists \mathbf{X})\mathbf{wff}$*, the subexpression,* **wff** *is called the* scope *of the quantifier.*

Definition 4.3 *An occurrence of a variable,* $\mathbf{X}$ *in a formula* **wff** *is* bound *if it occurs in* $(\forall \mathbf{X})$ *or* $(\exists \mathbf{X})$*, or in the scope of a quantifier* $(\forall \mathbf{X})$ *or* $(\exists X)$ *in* **wff**.

Definition 4.4 *An occurrence of a variable is* free *if it is not bound.*

Example 4.4 $(\forall Z)((\exists W)((W + Z) = X) \to (\exists W1)((W1 + Z) = Y))$*. In this formula, all occurrences of* X *and* Y *are free , and all other occurrences of other variables are bound. So this formula represents a predicate of arity two.*

Definition 4.5 *A* sentence *is a formula with no free variables.*

If $A(X_1, ..., X_n)$ is a formula with free variables $X_1, ..., X_n$, the sentence

$(\forall X_1)(\forall X_2)...(\forall X_n)A(X_1, ..., X_n)$

is called the universal closure of $A(X_1, ..., X_n)$.

On the other hand, the sentence

$(\exists X_1)...(\exists X_n)A(X_1, ..., X_n)$

is called the existential closure of $A(X_1, ..., X_n)$.

Problem 4.1 *L_N, the language of arithmetic, with signature $(0,', +, *)$ with arities $(0, 1, 2, 2)$ respectively, is a typical first order language. What are the variables, constants, terms, atomic formulae and formulae of this language? Give a grammar for each, and an example of each.*

4.8 Semantics for first order languages

We do not assume any predefined meaning for the function symbols and predicate symbols which occur in a first order language.

The value of a term in a first order language, or the truth value of a formula is likely to depend upon the interpretation given to the function symbols and the predicate symbols of the language.

Example 4.5 *L_G, the first order language of group theory, has term language generated by the usual variables, 1 (meant to be the identity), $X \circ Y$, meant to be the group operation, and X^{-1} meant to be the group inverse. The only predicate symbol needed in L_G is =, meant to be equality.*

However, in an interpretation of L_G, $\circ$ can be any operation : $D^2 \rightarrow D$, and X^{-1} can be any unary operation. Even though we want to use L_G to talk about groups, we do not impose any of our ideas on the set of allowable interpretations. The groups *are the interpretations which satisfy the axioms.*

We do not even insist that = be interpreted as the usual equality. This means that in this context when we give the group axioms we need to include some statements which characterise equality, as follows.

1. $X = X$
2. $X = Y \rightarrow Y = X$
3. $(X = Y \wedge Y = Z) \rightarrow X = Z$
4. $X = Y \rightarrow X^{-1} = Y^{-1}$
5. $(X = Y \wedge Z = W) \rightarrow X \circ Z = Y \circ W$
6. $X \circ (Y \circ Z) = (X \circ Y) \circ Z$
7. $X \circ 1 = X \wedge 1 \circ X = X$
8. $X \circ X^{-1} = 1 \wedge X^{-1} \circ X = 1$

To give an interpretation for a first order language we have to specify a set D which is called the domain of the interpretation, and we have to interpret each function symbol and predicate symbol in the signature of the language as a function or predicate of the appropriate arity defined over D. If a function symbol f has arity n, we have to interpret f as a function $\mathbf{f} : D^n \to D$. We always assume that such functions are totally defined and that they return values in D. If p is a predicate symbol of arity m, we have to interpret p as a predicate $\mathbf{p} : D^m \to \{T, F\}$.

An interpretation of a first order language, L, is given by a set D called the domain of the interpretation, together with a list of functions and predicates defined over D, corresponding to the list of function symbols and predicate symbols in the signature of L.

Let I be an interpretation of a first order language L.

Note that the interpretations are supposed to be mathematical objects, whereas the signatures and languages are supposed to be syntactic objects.

An interpretation I is also called a *structure*.

If I is an interpretation of a first order language, a valuation from I is a function which maps a subset of the variables of A into values in the domain of I. The support of a valuation is the set of variables to which it gives values. We will write valuations as if they were substitutions, e.g. $[X_1 := d_1, \ldots, X_k := d_k]$.

Definition 4.6 *Suppose I is an interpretation of a first order language L, v is a valuation from I, τ is a term of L and A is a formula of L. Suppose that the variables of τ and the free variables of A are included in the support of v. Then*

$\tau(v)$ is the value in the domain of I of the term τ under valuation v.

$A(v)$ is the truth value of A in interpretation I under valuation v.

Of course a formula may be true in one interpretation and false in others. We are especially interested in formulae which are true in all possible interpretations.

If S is a formula of some first order language, L, and I is an interpretation of L, we will write

$\models_I S$

to mean that S is true in I. If S has free variables,

$\models_I S$ means that S is true for all possible values of the free variables in the domain of I. In other words,

$\models_I S$ means $S(v) = T$ for all valuations from I whose support contains all the free variables of S.

S is true in I if and only if the universal closure of S is true in I

Of course the truth or falsity of S may depend upon how the predicate symbols, function symbols and constants are interpreted in I.

Definition 4.7 *We will say* $\models S$ *if* $\models_I S$ *for all possible interpretations* I*. In this case we will say that* S *is logically valid.*

$\models S$ means that S is logically valid. Intuitively, this is meant to express the idea that S is true by virtue of its logical form, independently of the interpretation of its function symbols and predicate symbols.

Definition 4.8 *We will say that a formula* S *is* satisfiable *if there exists an interpretation* I *and a valuation from the domain of* I *which makes* S *true.*

Intuitively, to say that S is satisfiable means that S is true in some possible world, with some evaluation of the free variables. Note that $\models S$ if and only if $\neg S$ is not satisfiable.

An important open problem in computer science is either to give a practical algorithm to decide whether or not if a quantifier free formula is satisfiable, or to show that no such algorithm exists. (A practical algorithm, in this context, would be one which terminated with the right result in a number of steps which was bounded by some polynomial in the length of the input.) This is a version of the famous $P = NP$ question.

Example 4.6 *A formula can look correct and yet not be logically valid. For example*

$(X < Y \wedge Y < Z) \rightarrow X < Z$.

This is not logically valid, since we are free to interpret $<$ *as any binary relation.*

Definition 4.9 *Suppose* Γ *is a list of formulae in a first order language and* M *is an interpretation of the language. We will say that* M *is a model of* Γ *if every formula of* Γ *is true in* M*.*

M is a model of Γ if and only if M is a model of the universal closure of Γ.

Example 4.7 *Let* $M = (D, \circ, X^{-1}, =, 1)$ *be an interpretation of* L_G *in which* D *is the non zero real numbers,* $\circ$ *means multiplication and* X^{-1} *is interpreted as* $1/X$*, and* $=$ *is the usual equality. This is a model for the axioms of group theory.*

Definition 4.10 *Suppose* Γ *is a list of formulae, and* S *is a formula in some first order language. We will say*

$$\Gamma \models S$$

if S *is true in every model of* Γ*. In this case we will also say that* S *is a logical consequence of* Γ*.*

The usual situation in mathematics is that we have a list Γ of axioms, and we are trying to decide whether or not some formula S is a logical consequence of Γ. We all know that such problems can be very hard.

In mathematics and in computing, when we give a list of specifying axioms, we hope to be completely explicit. The reason for allowing non intuitive interpretations of a first order language is to expose any lack of explicitness in sets of axioms. We do not want to build in behind the scenes assumptions which are not explicitly written.

Note: In $\Gamma \models S$, we can replace all the formulae by their universal closures without altering the situation.

Definition 4.11 *Two formulae A and B are logically equivalent if* $\models (A \leftrightarrow B)$.

4.9 Examples : L_N and L_R

L_N, defined earlier is the first order language of arithmetic. In the standard interpretation of this language, the domain is the natural numbers, including zero, $+$ and $*$ have their usual interpretation, and X' means $X+1$, and 0 and 1 have the usual meaning. This structure will be called $\mathbf{N}$.

Suppose S is a formula of L_N. The notation $\models_{\mathbf{N}} S$ means that S is true in the standard interpretation.

There is a famous set of axioms, called the Peano postulates, which is intended to capture the truths of arithmetic. This includes some simple statements which are essentially recursive definitions of $+$ and $*$,

1. $X' = Y' \rightarrow X = Y$
2. $\neg(0 = X')$
3. $\neg(X = 0) \rightarrow (\exists Y)(X = Y')$
4. $X + 0 = X$
5. $(X + Y') = (X + Y)'$
6. $X * 0 = 0$
7. $X * Y' = X * Y + X$

To this we add some axioms for equality

1. $X = X$
2. $X = Y \rightarrow Y = X$

3. $X = Y \wedge Y = Z \rightarrow X = Z$

4. $X = Y \rightarrow X' = Y'$

5. $X = Y \wedge Z = W \rightarrow (X + Z = Y + W \wedge X * Z = Y * W)$

and also an infinite list of induction axioms:
$((A(0) \wedge (\forall X)(A(X) \rightarrow A(X'))) \rightarrow (\forall X)A(X))$
where $A(X)$ can be any formula of L_N. Almost any known theorem which can be stated in L_N can be proved from the Peano postulates. We will see later, however, that this axiom set does not succeed in capturing all the truths of arithmetic.

As usual with axioms, we assume that all the statements are universally true. That is, $X' = Y' \rightarrow X = Y$ means that $(\forall X)(\forall Y)(X' = Y' \rightarrow X = Y)$. A structure in which all of these axioms are true (i.e. universally true) is called a model of the Peano postulates. We can see that there is at least one such model, $\mathbf{N}$.

Another famous first order language is L_R, the first order language of real algebra and geometry. This has functions symbols $+, -, *$, and predicate symbols $=, <$, all with arity 2, and has constants 0 and 1. The standard interpretation of this is $(\mathcal{R}, +, -, *, < . =, 0, 1)$, that is, the real numbers with the usual meanings of the symbols. In this language, with the standard interpretation, we could say that (X, Y) was inside the unit circle by the atomic formula
$X * X + Y * Y < 1$.

Unlike the situation with the natural numbers, the truths in this standard interpretation do have a complete axiomatisation.

4.10 Uses of these ideas in computing

There are a number of ways in which these ideas are used in computing practice. Just as the designer of a bridge may try to prove that it is unlikely to collapse, designers of computer systems try to prove that they are unlikely to fail, or that the code which they produce has some other desirable properties. We may consider a piece of code as a transformation which changes the values of some variables, i.e. as a transformation on valuations. In order to reason about the properties of code, the usual approach is to consider triples, (A, P, B) consisting of a precondition A, the code P, and a postcondition B. The precondition and postcondition are usually expressed in some formal language. The code is correct relative to this triple if whenever we start the code with a valuation satisfying the precondition, then, if the code terminates, the valuation on termination must

satisfy the postcondition. Such a triple may also be considered as a formal specification for the code.

Usually in this sort of application we are interested in truth in some particular interpretation, rather than in all possible interpretations. For example, if we were dealing with the programming language P1 defined earlier, we would consider the standard interpretation over the integers, with plus, minus, and times having the usual meanings.

4.11 More Substitution

Suppose A us a formula, and $T_1, ..., T_n$ are terms, and $X_1, ..., X_n$ are variables. Let $A[X_1 := T_1, ..., X_n := T_n]$ be the result of substitution every free occurrence of $X_1, ..., X_n$ by $T_1, ..., T_n$ respectively. As before in the term languages, all the substitutions are supposed to be done simultaneously. But the situation is much more complicated now than it was previously, because of the quantifiers.

Example 4.8 *Let A be $(\forall X)(\exists Y)(B(X, Z) \to C(Z, Y))$.*

Then $A[Z := W]$ would be $(\forall X)(\exists Y)(B(X, W) \to C(W, Y))$.

We can also form $A[Z := g(Y)]$. But there is something potentially wrong about this substitution. This is because the variable Y in $g(Y)$ will accidentally fall into the scope of the $(\exists Y)$ quantifier.

Let $(\forall X)A$ be a formula which is true in interpretation I. We might expect this truth to be preserved under substitution. So we might suppose $A[X := Te]$ to be true in I for any term Te. But the example above shows that we may need to be careful about these substitutions. Here is another example:

Example 4.9 *Suppose we consider the language of arithmetic L_N, and take the standard interpretation, in which the domain is the natural numbers, $=$ is equality, and $+, *, 0, 1$ have their usual meaning, and X' means $X + 1$. Call this interpretation* $\mathbf{N}$.

$(\exists Y)(Y = X)$ is true in $\mathbf{N}$. Now substitute Y' for X.

$(\exists Y)(Y = X)[X := Y']$ is $(\exists Y)(Y = Y')$, which is a false sentence in $\mathbf{N}$.

Why do we (incorrectly) expect that if $(\forall X)A$ is true in I then $A[X := \tau]$ should be true in I for any term τ? We think:

If $A[X := \tau]$ is not true in I, then there must exist a valuation v so that

$A[X := \tau](v) = F$. We expect that the result of first substituting and then evaluating should be an evaluation. So this would imply

$A(w) = F$ for some valuation w. But this can not happen if $(\forall X)A$ is true in I.

Obviously our expectations are wrong here. It seems that some substitutions followed by evaluation are not evaluations. Such substitutions are a common source of error in mathematics and computing.

The problem is always caused by the fact that a variable in the substituted term is quantified inside the formula, and accidentally falls into the scope of the quantifier after substitution.

Definition 4.12 *A term t is free for a variable X in a formula $A(X)$ if there is no free occurrence of X in $A(X)$ inside the scope of a quantifier, where the quantified variable also occurs in the term t.*

This very unpleasant definition has the following good consequence. If $\models_M A(X)$ and if term t is free for X in $A(X)$, then $\models_M A(t)$.

4.12 Other kinds of language and other logics

Suppose we have some domain D. We will call the elements of D objects of order zero. Predicates over D and functions which take arguments in D and return values in D will be called objects of order 1. These are the predicates and functions which are represented in first order languages. In general an order $n+1$ predicate is a predicate which speaks about objects of order n; and an order $n+1$ function is a function which takes arguments of order n and returns values of order n. Higher order languages can have quantifiers over higher order objects. For example, a nice way to express induction in the natural numbers would be:

For all predicates P over the natural numbers, if $P(0)$ is true and $(\forall n)(P(n) \to P(n+1)$ then $(\forall n)P(n)$ is true.

This is a second order statement. It is not possible to express this in L_N, the first order language of arithmetic.

Other kinds of logical connectives, and other kinds of logic, are also important in applications. For example, in temporal logic we might want to say that some formula A will become true at some time in the future, or might become true. In the modal logic of interacting agents, we might want to say that agent X believes that formula A is true; or that agent X believes that formula A might possibly be true. Many or the artificial systems which we create are nondeterministic, since they have autonomous independently acting subsystems, and what might happen is as important as what will definitely happen. For all of these ideas there are notations and corresponding notions of logical consequence.

In this course we will work on the simplest case of first order languages and classical logic.

4.13 Logic Games

Let S be a formula in a first order language L, and let M be an interpretation of L. Imagine a game played between two people, P, the proponent , and O, the opponent. P wants S to be true , and O desires falsity of S. The verification game is played using the formula S and the interpretation M. If S is an atomic sentence, P wins immediately if $M \models S$, and otherwise O wins immediately. If there are free variables, the opponent O chooses values for them in M, and the value is substituted into S. If S is $(\forall X)A$ then the opponent chooses a value for X. If S is $(\exists X)A$, the proponent P is allowed to choose a value for X. If S is $A \vee B$ the proponent chooses either A or B, and play continues with this choice. If S is $A \wedge B$, the opponent chooses either A or B and play continues with this choice. If S is $\neg A$, the proponent and the opponent switch roles, and play continues. $A \rightarrow B$ is dealt with as $\neg A \vee B$.

From this we get:

$M \models S$ if and only if the proponent has a winning strategy in this game.

If S is a sentence, then $M \models \neg S$ if and only if the opponent has a winning strategy in this game.

Try the following in the problems class, with one team playing proponent and one playing opponent. Which has a winning strategy? What is it?

Example 4.10 *Let S be* $(\forall X)(\forall Y)(r(X,Y) \vee (\exists Z)(r(X,Z) \wedge f(Z,Y)))$
Let M have domain $\{1,2,3,4\}$. *Assume* $r(3,1), r(1,2), r(2,3), r(4,3), r(4,2)$.

Next let the tutor change M and S and repeat the game.

4.14 Normal Forms

Within a first order language there are many formulae which are logically equivalent to any given formula. Given a formula, we would, of course, like to be able to transform it, preserving logical equivalence, into some simplified standard form.

These standard forms are also called normal forms, and there are quite a few of them: conjunctive normal form (CNF), disjunctive normal form (DNF), Prenex normal form, Skolem form, and clausal form.

4.14.1 CNF and DNF

CNF and DNF are obtained by rearranging propositional operators $\wedge, \vee, \rightarrow, \leftrightarrow, \neg$ between formulae.

We will say that a formula S is a *conjunction* of formulae $A_1, ..., A_n$ if S has the form

$A_1 \wedge A_2 \wedge ... \wedge A_n$.

We will write this as

$\bigwedge_{i=1}^{n} A_i$

We will say that a formula of the form $A_1 \vee A_2 \vee ... \vee A_n$ is a disjunction of formulae $A_1, ..., A_n$, and write this as

$\bigvee_{i=1}^{n} A_i$

Define a literal to be either an atomic formula or the negation of an atomic formula.

Definition 4.13 *A formula is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals. A formula is in disjunctive normal form (DNF) if it is a disjunction of conjunctions of literals.*

So a formula in CNF has the form

$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} A_{ij}$, where each A_{ij} is a literal.

That is, a formula in CNF all the ands on the outside, and all the negations applying directly to atomic formulae.

On the other hand a formula in DNF has the form

$\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} A_{ij}$, where each A_{Ij} is a literal.

Theorem 4.1 *Let S be any quantifier free formula. We can find a formula in CNF which is logically equivalent to S. We can also find a formula in DNF which is logically equivalent to S.*

proof.

We will give an algorithm for this.

Algorithm 4.1 DNF

Input: quantifier free formula S.

Output: $DNF(S)$ logically equivalent to S.

Method: Find the atomic formulae $A_1, ..., A_n$ of S. Construct a truth table, showing how the truth value of S depends on the truth values of these atomic formulae. This truth table will have $n+1$ columns, for the truth values of $A_1, ..., A_n, S$ respectively, and 2^n rows. Let the rows which make S true be $R_1, ..., R_k$. We have

S is true if and only if

R_1 happens or R_2 happens or ... R_k happens.

So return $DNF(S)$ as
$\bigvee happens(R_i)$
and write $happens(R_i)$
$\bigwedge_{j=1}^{n} B_{ij}$
where $B_{ij} = A_i$ if R_i gives A_j value True, and $B_{ij} = (\neg A_i)$ if R_i gives A_j value False.

□

The CNF algorithm does this the other way around.

Algorithm 4.2 CNF
Input: quantifier free formula S.
Output: $CNF(S)$ logically equivalent to S.
Method: Find the atomic formulae $A_1, ..., A_n$ of S. Construct a truth table, showing how the truth value of S depends on the truth values of these atomic formulae. This truth table will have $n + 1$ columns, for the truth values of $A_1, ..., A_n, S$ respectively, and 2^n rows. Let the rows which make S false be $R_1, ..., R_k$. We have
S is true if and only if
R_1 does not happen and R_2 does not happen and ... R_k does not happen.
So return $DNF(S)$ as
$\bigwedge nothappens(R_i)$
and write $nothappens(R_i)$
$\bigvee_{j=1}^{n} B_{ij}$
where $B_{ij} = A_i$ if R_i gives A_j value False, and $B_{ij} = (\neg A_i)$ if R_i gives A_j value True.

□

Example 4.11 *Consider $C \leftrightarrow D$. Construct the truth table. DNF for this is $(C \wedge D) \vee \neg C \wedge \neg D)$. CNF for this is $(C \vee \neg D) \wedge (\neg C \vee D)$.*

Note that the method given above can not be called practical, since the amount of work to be done increases with the number of rows of the truth table and this increases exponentially with the number of atomic formulae in the problem.

There are many other ways of finding CNF and DNF, which are not unique. None of the other known methods are practical, i.e. have complexity bounded by some polynomial in the size of the input.

Exercise: Write specifications for the DNF and the CNF algorithms. What does it mean to say that the algorithms are correct with respect to the specifications? What does it mean to say that the algorithms are complete with respect to the specifications? Do the algorithms always terminate?

4.14.2 Prenex Normal form

A formula is in prenex normal form if all the quantifiers are in the front. This means that the formula has the form

$Q_1 Q_2 ... Q_k M$

where each Q_i has the form $(\exists X_i)$ or $(\forall X_i)$, and X_i is a variable, and M is quantifier free.

Note: you may need to use all the correct bracketing to be sure that a formula is in prenex normal form.

For example,

$(\forall X)(A(X) \to B(X))$ is in prenex normal form. However

$((\forall X)A(X) \to B(X))$ is not in prenex normal form, since $B(X)$ is not in the scope of the quantifier in this case.

To say that a formula is in prenex normal form means that all the outermost operators are quantifiers. In $((\forall X)A(X) \to B(X))$, the outermost operator is $\to$.

Given any formula S, we can find a logically equivalent formula in prenex normal form. This is done by progressively moving the quantifiers outside the propositional operators, as explained below.

In order to do this correctly, it is sometimes necessary to rename bound variables. (This is also called α conversion.) The idea is to replace $(\forall X)A(X)$ by $(\forall Y)A(Y)$, where Y is a new variable, which was not used previously. In this case $(\forall X)A(X)$ has the same meaning (i.e. is logically equivalent to) $(\forall Y)A(Y)$. Similarly $(\exists X)A(X)$ is logically equivalent to $(\exists Y)A(Y)$, provided that Y is a new variable.

For example $(\exists X)(X = W')$ has the same meaning as $(\exists Y)(Y = W')$. We can rename the bound variable X as Y, since Y does not occur in the original formula. On the other hand, great confusion can result by renaming bound variables with variables which are already present. For example $(\exists X)(X = W')$ does not have anything like the same meaning as $(\exists W)(W = W')$.

Algorithm 4.3 *Input: Formula S*

Output: a formula in prenex normal form logically equivalent to S.

Method:

1. *Replace all subformulae using $\leftrightarrow$ using*

   $(A \leftrightarrow B) \rightsquigarrow (A \to B) \land (B \to A)$

2. *Replace all subformulae using $\to$ using*

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

3. *Rename all bound variables so that no variable occurs both bound and free in any subformula, and so that no two occurrences of the same variable are in the scopes of different quantifiers.*

4. *Move negations inside quantifiers with*

   $\neg(\forall X)A(X) \rightsquigarrow (\exists X)(\neg A(X))$

   $\neg(\exists X)A(X) \rightsquigarrow (\forall X)(\neg A(X))$

5. *Move all* $\wedge$ *and* $\vee$ *inside all quantifiers with*

   $((QX)A\ opB) \rightsquigarrow (QX)(A\ opB)$ *and*

   $(A\ op(QX)B) \rightsquigarrow (QX)(AopB)$

□

4.14.3 Skolem form

It is fairly difficult to understand formulae with alternations of quantifiers. A famous example of this is the definition of continuity of a function $f(X)$ at a point $X = a.$:

$(\forall\epsilon)(\exists\delta)(\forall X)(((\epsilon > 0 \rightarrow \delta > 0) \wedge (\mid X - a \mid < \delta) \rightarrow (\mid f(X) - f(a) \mid < \epsilon))$.

Suppose we are asked what this means. We might say that if we were given an $\epsilon > 0$ we could find a number $\delta > 0$ so that, for all X , $\mid f(X) - f(a) \mid < \epsilon$ if $\mid X - a \mid < \delta$. This seems equally hard to understand. How can we simplify this?

An important point is that the number δ depends on the number ϵ, but not on the number X. So we might proceed by introducing a function $\delta(\epsilon)$. We would then have

$(\forall\epsilon)(\forall X)$

$(\epsilon > 0 \rightarrow \delta(\epsilon) > 0\wedge$

$(\mid X - a \mid < \delta(\epsilon) \rightarrow \mid f(X) - f(a) \mid < \epsilon$.

Of course it is still complicated. But using this function seems to improve things a bit. The function $\delta(\epsilon)$ is an example of a Skolem function.

A formula is in Skolem form if it is in prenex normal form and only uses universal quantifiers. In order to put a formula into Skolem form we get rid of existential quantifiers by using new function symbols, such as $\delta(\epsilon)$ in the example above..

Suppose a sentence is true in some interpretation M, and is in Skolem form. In terms of the verification game between Proponent and Opponent mentioned earlier, the Skolem functions give a winning strategy to the Proponent.

Example 4.12 *Consider the formula*
$(\forall X)(\forall Y)(\exists Z)(X \leq Z \wedge Y \leq Z)$
This is in prenex normal form. The Z alleged to exist depends on X and Y. If this statement is true, there must be some function $Z = f(X, Y)$ which makes it true. The Skolem form of the statement is:
$(\forall X)(\forall Y)(X \leq f(X, Y) \wedge Y \leq f(X, Y))$
The existential quantifier has been eliminated by introducing a new function $f(X, Y)$.

It is fairly easy to see how to find a Skolem form for a formula in prenex normal form. Suppose our formula is:
$(\forall X_1)(\forall X_2)...(\forall X_n)(\exists Y)M(W_1, ..., W_k, X_1, ..., X_n, Y)$
where $M(W_1, ..., W_k, X_1, ..., X_n, Y)$ is another formula in prenex normal form, having free variables $W_1, ..., W_k, X_1, ..., X_n, Y$. Suppose we wish to introduce a Skolem function to get rid of the $(\exists Y)$.

The value Y which is asserted to exist by the original formula may depend on $W_1, ..., W_k, X_1, ..., X_n$. We pick a new function symbol for our Skolem function. Not trying to be very original here, suppose we choose f. It will have arity $k + n$, where k is the number of free variables in the original formula and n is the number of universal quantifiers to the left of $(\exists Y)$. Replacing this existential quantifier, we get:
$(\forall X_1)(\forall X_2)...(\forall X_n)M(W_1, ..., W_k, X_1, ..., X_n, f(W_1, ..., W_k, X_1, ..., X_n))$
Inside M there may remain more existential quantifiers. So this process of inventing new function symbols and replacing existential quantifiers is repeated until we get Skolem form.

Example 4.13 *A Skolem form of*
$(\forall X)(\exists Y)(\forall Z)(\forall W)(\exists V)A(X, Y, Z, W, V, U)$
would be
$(\forall X)(\forall Z)(\forall W)A(X, dog(X, U), Z, W, cat(X, Z, W, U), U)$
using cat and dog as new function symbols.

Skolem form is not unique, since we can use any new function symbol we choose. It is essential to use a function symbol which has not been used previously.

The Skolem form of a formula logically implies the original formula. However the converse is false. The original formula does not imply the Skolem formula. The Skolem form is slightly stronger. So we have lost logical equivalence. On the other hand, the difference is small.

Suppose we have an interpretation, I, in which the original formula is true. The Skolem form can not even be interpreted in I as it stands, since we have used new function symbols. But since the original formula is true,

we can extend I, defining the Skolem functions so that the Skolem form is true.

Theorem 4.2 *Suppose $sk(A)$ is a Skolem form of formula A. $sk(A) \models A$. Furthermore, if I is an interpretation in which A is true, then I can be extended, by appropriate definition of the Skolem function symbols, to an interpretation $I*$ so that $sk(A)$ is true in $I*$.*

If we regard a formula A as an axiom which makes some demands on reality, it seems reasonable to say that a Skolem form of A makes the same demands, except for notation. If it is possible to satisfy A, it is also possible to satisfy a Skolem form of A, and vica versa.

Note that a formula is satisfiable if and only if its Skolem form is satisfiable.

Remark: we can keep logical equivalence if we are willing to go up to a higher order language. For example $(\forall X)(\exists Y)a(X,Y)$ has Skolem form $(\forall X)a(X, f(X))$, where $f(X)$ is a Skolem function. If we allowed quantification over function variables we would have
$(\exists f)(\forall X)a(X, f(X))$
and this is logically equivalent to the original statement.

4.14.4 Clausal form

Definition 4.14 *A clause is a formula of the form*
$(A_1 \wedge A_2 \wedge ... \wedge A_r) \rightarrow (B_1 \vee B_2 \vee ... \vee B_s)$, *where* $A_1, ..., A_r, B_1, ..., B_s$ *are atomic formulae.*

In a clause, a conjunction of atomic formulae implies a disjunction of atomic formulae. It allowed for for either the conjunction or the disjunction to be empty. The clause
$(A_1 \wedge A_2 \wedge ... \wedge A_r) \rightarrow (B_1 \vee B_2 \vee ... \vee B_s)$
is logically equivalent to the disjunction of literals
$(\neg A_1 \vee \neg A_2 \vee ... \vee \neg A_r \vee B_1 \vee B_2 \vee ... \vee B_s)$.

There is no mechanical method for putting a sentence into the simplest form. The nearest approach we have to this is clausal form. Given a sentence A, a clausal form is obtained as follows.

1. Find prenex normal form of A
2. Find Skolem form. We now have something of the form

   $(\forall X_1)...(\forall X_n)M(X_1, ..., X_n)$,

   where M is quantifier free.

3. Delete the universal quantifiers. We now have a quantifier free formula $M(X_1, ..., X_n)$
4. Put M into conjunctive normal form. We get $C_1 \wedge C_2 \wedge ... \wedge C_j$, where each C_i is a disjunction of literals.
5. Write the conjunction as a list $\Gamma = [C_1, ..., C_j]$
6. Write each disjunction $\neg A_1 \vee \neg A_2 \vee ... \vee \neg A_r \vee B_1 \vee ... \vee B_s$ in the form of a clause

   $(A_1 \wedge A_2 \wedge ... \wedge A_r) \rightarrow (B_1 \vee ... \vee B_s)$.

   The result is a clausal form of A.

This method only makes sense for sentences. That is, we have to start with A which has no free variables.

We end up with a list of quantifier free clauses. Except for the fact that we may have introduced new function symbols, the clausal form has the same meaning as A. On the other hand, the clausal form is often a lot easier to understand than the original.

The clausal form process may be regarded as a simplification procedure.

Example 4.14 *Consider again the continuity definition:*
$(\forall\epsilon)(\exists\delta)(\forall X)(((\epsilon > 0 \rightarrow \delta > 0) \wedge (\mid X - a \mid < \delta) \rightarrow (\mid f(X) - f(a) \mid < \epsilon))$.
A clausal form for this is:
$\epsilon > 0 \rightarrow \delta(\epsilon) > 0$
$(\mid X - a \mid < \delta(\epsilon)) \rightarrow (\mid f(X) - f(a) \mid < \epsilon))$.

4.15 Problems

Problem 4.2 *(6) Put the following statements into conjunctive normal form, and also disjunctive normal form. Check that each statement has the same truth table as its conjunctive normal form.*

a) $(\neg a \wedge b) \rightarrow \neg b)$

b) $(a \wedge ((\neg p) \leftrightarrow \neg q))$

Problem 4.3 *(4)*

Find clausal form for $((a \vee b) \rightarrow c)$

Problem 4.4 *(45) Translate the following statements into some first order language. Note that in e) and h) you may need to use equality.*

a) Not all toothless animals which have feathers can fly.

b) If some human being is in prison than all human beings are in prison.

c) It is not true that every short person who is not crazy likes some other person who is either not short or crazy.

d) Gorillas are hairier than billiard balls.

e) Everyone is either loved or hated by some other person.

f) There is a town in Spain in which there is a barber who shaves everyone in the town who does not shave themselves.

g) Everyone is happy except for Jane.

h) There is only one thing with wheels in the shed and that is a bicycle with a flat tire.

i) For every X *and* Y *there is a* Z *so that* $X \leq Z$ *and* $Y \leq Z$*, and if* W *is such that* $X \leq W$ *and* $Y \leq W$ *then* $Z \leq W$*.*

note: such a Z *is called the least upper bound of* X *and* Y*. Can you see how to establish that least upper bounds are unique?*

Problem 4.5 *(5) a) Repeat the first two parts of previous problem (i.e. the statements about feathered animals and prisoners) without using any existential quantifier.*

b) (5) Repeat the first two parts of the previous problem without using any universal quantifier.

Problem 4.6 *(20)*

Put the statements a,b,c,d,e above into clausal form.

4.16 Summary

In this chapter we have defined first order languages and given them a semantics. We have defined the notions of logical consequence and logical equivalence.

We have also developed a number of normal forms for formulae in first order languages: conjunctive normal form, disjunctive normal form, prenex normal form, Skolem form, and clausal form.

The most valuable and interesting of the normal forms is clausal form. The clausal form of a sentence is a list of clauses, each clause being in the form:

$\bigwedge A_i \rightarrow \bigvee B_i$

where each A_i and B_i is an atomic formula. Thus clausal form uses no quantifiers, and seems not to use negation. The crucial step in the reduction of a sentence to clausal form is replacement of existential quantifiers by use of Skolem functions.

The clausal form of a sentence is weakly equivalent to the original sentence in this sense: Any interpretation which makes the clausal form true also makes the original sentence true. Any interpretation which makes the original sentence true can be extended, by definition of the Skolem functions, to an interpretation which makes the clausal form true. In particular, the original sentence can be satisfied if and only if the clausal form can be satisfied.

4.17 Revision Problems. Clausal form

Put the following into clausal form.

Problem 4.7 *No man is an island*

Problem 4.8 *Every dog, except for Hugo, is frightened of some cat. But Hugo is not frightened of any cat.*

Problem 4.9 *If you can get from A to B by rail and you can get from B to C by rail, then you can get from A to C by rail, except on Wednesday.*

Problem 4.10 *Jane only likes men who are crazy.*

Problem 4.11 *Every bar stool has a person sitting on it, and every such person has a mother, alive or dead.*

Problem 4.12 *Substitution α is a most general unifier of terms T and S.*

Problem 4.13 *If $A(0)$ is true and if $A(x)$ implies $A(x+1)$ for all x, then $A(x)$ is true for all x.*

Problem 4.14 $(\forall \epsilon)(\epsilon > 0 \rightarrow (\exists \delta)(\delta > 0 \wedge (\forall X)(\mid X - X_0 \mid < \delta \rightarrow \mid f(X) - f(X_0) \mid < \epsilon)))$

Problem 4.15 $((\exists X)A(X) \rightarrow (\exists X)B(X))$

Problem 4.16 $((\forall X)A(X) \rightarrow (\forall X)B(X))$

Problem 4.17 $((\exists X)A(X) \rightarrow ((\exists X)B(X) \rightarrow (\exists X)C(X)))$

4.18 Solutions

1. $m(X) \wedge is(X) \rightarrow$
2. $d(X) \rightarrow X = h \vee c(\gamma(X))$
   $d(X) \rightarrow X = h \vee fright(X, \gamma(X))$
3. $r(A,B) \wedge r(B,C) \rightarrow wednesday \vee r(A,C)$
4. $male(X) \wedge likes(j,X) \rightarrow crazy(X)$
5. $barstool(B) \rightarrow human(\alpha(B))$
   $barstool(B) \rightarrow mother(\mu(B), \alpha(B))$
   $barstool(B) \rightarrow d(\mu(B)) \vee al(\mu(B))$

6. $\rightarrow T_\alpha = S_\alpha$
   $T_\beta = S_\beta \rightarrow T_{\alpha\circ\delta(\alpha,\beta)} = T_\beta$
7. $A(0) \rightarrow A(\tau) \vee A(Y)$
   $A(0) \wedge A(\tau + 1) \rightarrow A(Y)$
8. see previous notes
9. $B(X) \rightarrow C(\tau(X))$
10. $A(\tau) \rightarrow B(Y)$
11. $A(X) \wedge B(Y) \rightarrow C(\tau(X,Y))$

4.19 Examples and Solutions

1. Translate the following into some first order language. In each case, give the signature of the language. Where there seems to be ambiguity in the English, point this out and explain which possibility you have chosen. (30)

   (a) If some of Al's chickens have mange, then all of Al's chickens have mange.
   $(\exists X)(a(X) \wedge ch(X) \wedge mg(X)) \rightarrow (\forall X)((a(X) \wedge ch(X)) \rightarrow mg(X))$

   (b) Some people like other people who do not like anyone.
   $(\exists X)(p(X) \wedge (\exists Y)(p(Y) \wedge likes(X,Y) \wedge (\forall Z)(p(Z) \rightarrow \neg likes(Y,Z))))$

   (c) George must be eliminated unless Alice only saw one kangaroo.
   I translate "A unless B" as "A or B".
   $elim(g) \vee ((\exists X)(saw(a,X) \wedge kang(X)) \wedge (\forall Y)(\forall W)(kang(Y) \wedge kang(W) \wedge saw(a,Y) \wedge saw(a,W)) \rightarrow Y = W)$

2. Put the statements of the previous section into clausal form. (30)

   Prenex normal form is

   $(\forall X)(\forall Y)((a(X) \wedge ch(X) \wedge mg(X) \wedge a(Y) \wedge ch(Y)) \rightarrow mg(Y))$

   a)

   Clausal form is

   $(a(X) \wedge ch(X) \wedge mg(X) \wedge a(Y) \wedge ch(Y)) \rightarrow mg(Y)$

   b) Clausal form is

   $\rightarrow p(\tau)$

$\rightarrow p(\rho)$

$\rightarrow likes(\tau, \rho)$

$p(Z) \wedge likes(\rho, Z) \rightarrow$

c) Clausal form is

$\rightarrow elim(g) \vee saw(a, \tau)$

$\rightarrow elim(g) \vee kang(\tau)$

$saw(a, X) \wedge saw(a, W) \wedge kang(X) \wedge kang(W) \rightarrow elim(g) \vee X = W$

3. Put the following statement into conjunctive normal form: $(((p \wedge q) \rightarrow r) \rightarrow \neg q)$ (10)

   $(p \vee \neg q) \wedge (\neg r \vee \neg q)$

4. Put the following statement into Skolem form:

   $((\forall X)a(X) \rightarrow (\forall X)(\exists Z)b(X, Z))$ (10)

   $(\exists X)(\forall Y)(\exists Z)(a(X) \rightarrow b(Y, Z))$ is prenex normal form.

   Skolem form is

   $(\forall Y)(a(\tau) \rightarrow b(Y, h(Y))$

Chapter 5

Semantic Tableaux

Assume that we have a set of axioms Γ, written in some first order language, for some field of knowledge. Imagine that a lot of work and experience has gone into Γ. We believe that Γ summarises every known truth in some area which concerns us.

The nice feature of this situation is that Γ may be quite small. It is a compact representation for all of its logical consequences. We believe that the information we want is in Γ. But how do we get this information?

It becomes clear that we need to be able to answer the following question.

Given Γ, and formula A, decide whether or not $\Gamma \models A$.

This is called the logical consequence problem, for the predicate calculus.

Of course this may be hard. As if it were not hard enough, another problem immediately suggests itself.

Given formula $A(X_1, ..., X_n)$ with free variables $X_1, ..., X_n$, find, if possible, values $v_1, ..., v_n$ so that $\Gamma \models A(v_1, ..., v_n)$.

If we are to make some progress with these problems computationally, we need some formalisation of the concept of proof.

There are a number of formal systems for deduction in predicate logic. One technique is called forward chaining. A forward chaining system is given by a certain number of rules of inference. Sometimes, for example, such systems include the following rule, which is called modus ponens:

A, $(A \to B)$

B

This says that if we have proved A and also $(A \to B)$, we may conclude B.

In a forward chaining system, we start with the axioms Γ and apply the rules of inference. We regard all the formulae in Γ as true, and every time we apply a rule of inference we enlarge the set of known truths. We will say $\Gamma \vdash A$ if we can eventually derive A from Γ using the rules of inference. If we think that A really is a logical consequence of Γ, we must try to get from Γ to A using the rules of inference.

There are several severe difficulties here. Even if we know that $\Gamma \models A$, it may be extremely difficult to find the right way to apply the rules of inference to get from Γ to A. If we are not confident that A is a logical consequence of Γ, our problem is even worse. A forward chaining system will never establish that A is not a logical consequence of Γ.

Backward chaining systems are based on the method of proof by contradiction. We begin by assuming that A is not a logical consequence of Γ and proceed to attempt to construct a counterexample. (A counterexample would be an interpretation in which Γ is true but A is false.) If our attempt to construct a counterexample is eventually blocked by contradictions, we conclude that there is no counterexample, and thus our original assumption was incorrect, and $\Gamma \models A$. On the other hand, if some branch of the construction never gets blocked, the construction, in the limit, should produce a counterexample, and in this case A is not a logical consequence of Γ.

As mentioned above there are a large number of formalisations of proof in predicate logic. The semantic tableaux method is one of these. This is a backward chaining system.

5.1 Introduction

We should sort out the notation first. $\Gamma \models A$ means that A is a logical consequence of Γ. As explained previously, this means that if I is any interpretation which makes Γ true, then I must make A true. So $\Gamma \models A$ is about semantics. It means that in *reality* A is true whenever Γ is true.

On the other hand, we will have a syntactic definition of proof.

Definition 5.1 *If Γ is a list of sentences and S is a sentence, we will write*

$$\Gamma \vdash S$$

if we can prove S from Γ in the semantic tableaux system.

Note that at this stage we are only dealing with sentences. So we will need to replace our axioms by their universal closure.

$\Gamma \vdash A$ means that there is a formal proof of A from Γ. We now have to explain what such a formal proof would be.

Since we are trying to construct counterexamples, we will have to explore a number of alternatives. To reflect this the data structures we will use will be trees labelled with formulae.

We will first give some examples, and then describe the technique more precisely.

Example 5.1 *Suppose* Γ *is the empty set, and* A *is* $(p \to (q \to p))$*. We want to know whether or not* $(p \to (q \to p))$ *is logically valid. We are using proof by contradiction. So assume that it is false. This gives us the beginning of our proof tree, i.e. one node.*

```
(1) not  (p->(q->p))
```

The form of the supposedly false expression is $A \to B$*. How can such a formula be false? (Review the truth table definition of implication.) In fact there is only one way that* $(A \to B)$ *can be false. It must be that* A *is true and* B *is false. So we can see how to continue our construction.*

```
(1) not (p->(q->p))
 |
(2, from 1) p
 |
(3, from 1) not (q->p)
```

We now continue the construction at node (3).

```
(1) not (p->(q->p))
 |
(2, from 1) p
 |
(3, from 1) not (q->p)
 |
```

```
(4, from 3) q
 |
(5, from 3) not p , contradicts (2)
```

The construction ends in a contradiction. That is, node (2) tells us that p is true, and node (5) tells us that p is false. Therefore, there is no counterexample. Thus $\models (p \rightarrow (q \rightarrow p))$.

Furthermore, the tree shown above is a **proof** *of this in the semantic tableau system.*

(Note that in these examples "not p" is used instead of $\neg p$, purely for the convenience of the typist. In later semantic tableaux, "(For all X)" and "(Exists X)" will be used instead of $(\forall X)$ and $(\exists X)$ for the same reason.)

Here is another example, in which there is some branching.

Example 5.2 *Decide whether or not* $\models ((p \rightarrow q) \rightarrow q)$.

```
(1) not ((p->q)->q)
 |
(2, from 1) (p ->q)
 |
(3, from 1) not q
```

At this point we have $(p \rightarrow q)$ *asserted to be true. How can an implication be true? There are two ways in which this can happen. (Please review truth table definition of implication.) Either the premise is false or the conclusion is true. So we continue the construction by branching to reflect this.*

```
                  (1) not((p->q)->q)
                   |
                  (2, from 1) (p ->q)
                   |
                  (3, from 1) not q
                /    \
               /      \


(4, from 2) not p       (4, from 2) q  contradicts 3)
```

Please look at the tree below node 3). It is supposed to say that either p is false or q is true.

We now have a contradiction on one branch. There is no contradiction on the other branch. There is also nothing more we can do. So the construction on the left hand branch has ended in a counterexample. Reading down the left hand branch, we can see that p and q are both false. (Check that this is a counterexample.)

In a tree, the node at the top is called the root. The node or nodes immediately below a node are called its children. A node with no children is called a leaf. The collection of leaves is called the frontier of the tree. A path which starts at the root and goes all the way to the frontier will be called a branch. If a node is labelled with formula A, we will say that A is asserted at the node. If a node is labelled with $\neg A$, we will say that A is denied at the node.

In a semantic tableau, we will say that a branch is closed if it contains a contradiction. This means that some formula is both asserted and denied on the branch. The tableau is closed if all branches are closed.

Suppose we wish to try to decide whether or not $\Gamma \models A$. Assume that A is a sentence, and all the formulae in Γ are also sentences. The semantic tableau method is the following. Form an initial semantic tableau with all the formulae of Γ asserted, and A denied. Then apply the semantic tableau rules, which will be described below. If a closed tableau is eventually obtained, it follows that no counterexample is possible, and so A is a logical consequence of Γ. The closed tableau is a proof of A from axioms Γ.

5.2 Semantic tableau rules

1. If $(A \to B)$ is denied, add nodes to assert A and deny B.

2. If $(A \to B)$ is asserted, form two branches and add a node denying A on one and a node asserting B on the other.

3. If $\neg A$ is denied, add a node asserting A

4. If $(A \wedge B)$ is asserted, add two nodes, one asserting A and the other asserting B.

5. If $(A \wedge B)$ is denied, form two branches, and deny A on one and deny B on the other.

6. If $(A \vee B)$ is denied, add two nodes, one denying A and the other denying B .

7. If $(A \vee B)$ is asserted, form two branches, and assert A on one, and assert B on the other.

8. If $(A \leftrightarrow B)$ is asserted, form two branches, and deny both A and B on one and assert both A and B on the other.

9. If $(A \leftrightarrow B)$ is denied, form two branches and assert A and deny B on one and assert B and deny A on the other.

10. If $(\forall X)A(X)$ is asserted, and if t is any variable free term, assert $A(t)$. This may be done for any number of variable free terms t.

11. If $(\forall X)A(X)$ is denied invent a new Skolem constant c, and deny $A(c)$.

12. If $(\exists X)A(X)$ is asserted, invent a new Skolem constant c and assert $A(c)$.

13. If $(\exists X)A(X)$ is denied, and if t is any variable free term, deny $A(t)$. This may be done fro any number of variable free terms t.

Rules 10) and 13) are called substitution rules. They may be applied any number of times. Some proofs need many substitutions. Note that only variable free terms may be substituted. So, for example, in arithmetic, $(0'''''' + 0'')$ may be substituted, but $(X + 0')$ may not be substituted. This restriction ensures that all the formulae in a semantic tableau are sentences. Variable free terms may involve arbitrarily many function symbols and constants, and may be nested to any depth. Rule 10) says that if we assert $(\forall X)A(X)$ and if t is any variable free term, then we may also assert $A(t)$. Rule 13) says that if we deny $(\exists X)A(X)$ and if t is any variable free term, then we may also deny $A(t)$.

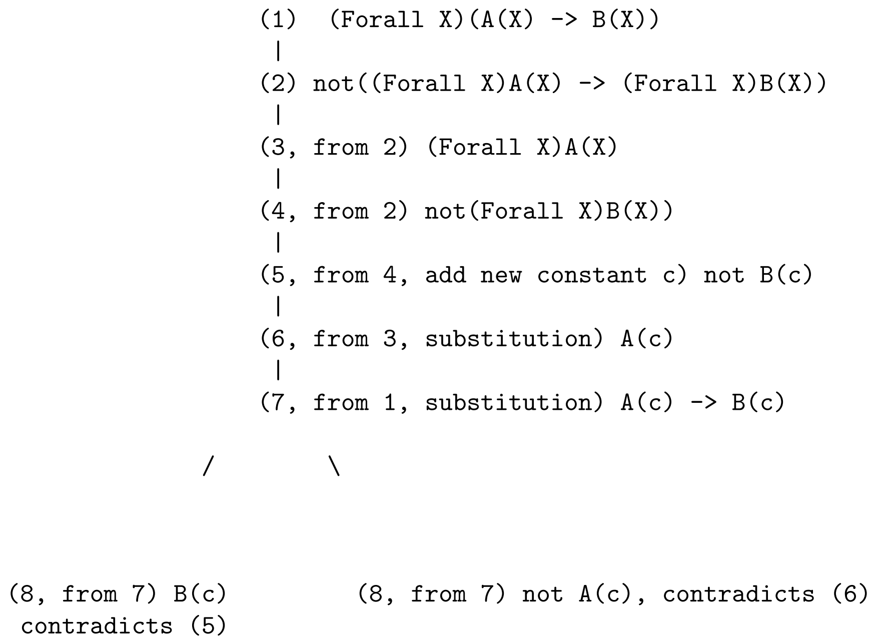
In rules 11) and 12), a new Skolem constant is introduced. It is important to realize that the constant which is introduced must not have been previously used anywhere in the tree. The constant names an object which either makes $(\exists X)A(X)$ true, or makes $(\forall X)A(X)$ false. If we correctly assert $(\exists X)A(X)$, then there must exist an object which makes $A(X)$ true; we can call such an object by any name we choose, provided only that this name has not been used previously. Similarly, if we correctly deny

$(\forall X)A(X)$, it must follow that there is at least one object which makes $A(X)$ false; and we can invent a new name for such an object.

Note that a denied disjunction behaves like a conjunction of denials. That is $\neg(A \vee B)$ is the same as $\neg A \wedge \neg B$. Furthermore, a denied conjunction behaves like a disjunction of denials.

The universal quantifier $\forall$ is, in a sense, just a big conjunction, and $\exists$ is, in a sense, just a big disjunction. So a universal quantifier asserted behaves like an existential quantifier denied.

Example 5.3 *Suppose we want to test whether or not* $(\forall X)(A(X) \rightarrow B(X))$ *logically implies* $((\forall X)A(X) \rightarrow (\forall X)B(X))$. *We start by asserting* $(\forall X)(A(X) \rightarrow B(X))$ *and denying* $((\forall X)A(X) \rightarrow (\forall X)B(X))$



```
                    (1)  (Forall X)(A(X) -> B(X))
                     |
                    (2) not((Forall X)A(X) -> (Forall X)B(X))
                     |
                    (3, from 2) (Forall X)A(X)
                     |
                    (4, from 2) not(Forall X)B(X))
                     |
                    (5, from 4, add new constant c) not B(c)
                     |
                    (6, from 3, substitution) A(c)
                     |
                    (7, from 1, substitution) A(c) -> B(c)

                  /         \


(8, from 7) B(c)         (8, from 7) not A(c), contradicts (6)
 contradicts (5)
```

Note that the Skolem constant c does not occur above node 5), where it was introduced, using rule 13). All branches are closed. So there is no counterexample. Therefore

$(\forall X)(A(X) \rightarrow B(X)) \models (\forall X)A(X) \rightarrow (\forall X)B(X)$. *The above tree is a proof of this in the semantic tableau system.*

Definition 5.2 *If T_1 and T_2 are semantic tableaux, we will say $T_1 \Rightarrow T_2$ if T_2 can be obtained from T_1 by one application of the rules.*

Definition 5.3 *Let Γ be a set of sentences, and let A be a sentence in some first order language. We will say $\Gamma \vdash A$ in the semantic tableau system if $T_1 \Rightarrow^* T_k$ where T_k is closed, and T_1 is the initial tableau with Γ asserted and A denied.*

5.3 Some advice about the use of the rules

The semantic tableau rules may be applied in any order. But it seems advisable to apply certain rules before others. With this in mind, we can divide the rules into categories.

1. Propositional rules (involving $\wedge, \vee, \neg, \rightarrow$) which do not increase the number of branches.
2. Quantifier rules involving introduction of a new Skolem constant.
3. Propositional rules which may increase the number of branches.
4. Substitution rules (which involve choose a variable free term and substituting it). That is, when $(\forall X)A(X)$ is asserted, or $(\exists X)A(X)$ is denied.

On each branch, the rules in the first three categories only need to be applied once. The substitution rules can be applied any number of times with different terms on any branch. Since we do not want to have a lot of branches, it seems a good idea to make all applications of rules in the first two categories, before doing anything else. In general, it also seems a good idea to do all possible category 3) operations, before doing any substitutions. Of course there are cases in which this advice is not useful.

Whatever strategy we follow, we do not want indefinitely to defer taking any allowed operation. Everything which can be done on any open branch should eventually be done. Unless of course the branch gets closed for some other reason. Since if there are function symbols in our language, there may be infinitely many variable free terms, and thus infinitely many possible substitutions, a perverse user of the semantic tableau system could continue forever without getting a proof by insisting on doing one useless substitution after another, even though closure might be obtained in some other way.

Definition 5.4 *We will say that the semantic tableau construction is done systematically if any operation which is possible on any open branch (including all substitutions) is eventually done, unless all branches extending it get closed first.*

We will return later to the important question of how to choose useful substitutions, from among the many possibilities. You may already guess that unification will be involved in this.

Before reading the next section, you should probably try the following. Note that in the one case in which validity can be proved, a stubborn user of the system can avoid ever getting closure, and thus may never find the proof, by insisting on doing an infinite sequence of useless substitutions.

Problem 5.1 *Either give a proof of validity or a counterexample.*

1. $((\exists X)a(X) \rightarrow (\exists X)b(X)) \rightarrow (\exists X)(a(X) \rightarrow b(X))$
2. $((\exists X)(a(X) \rightarrow b(X)) \rightarrow (\exists X)a(X) \rightarrow (\exists X)b(X))$

5.4 How much use is this system?

5.4.1 Gödel completeness theorem

To begin with, we have an extraordinary theorem, due to Kurt Gödel.

Assume that we are using a first order language with at least one constant, so that we have something to substitute to begin with. We can always add such a constant to any first order language.

Theorem 5.1 Gödel's completeness and correctness theorem !!

$$\Gamma \models S$$

if and only if

$$\Gamma \vdash S$$

for any list of sentences Γ and any S in any first order language.

We will eventually sketch a proof of this. Before the proof, we need some definitions and a lemma.

Definition 5.5 *We will say that an interpretation I satisfies a branch α of a semantic tableau, T, if every sentence asserted in α is true in I and every sentence denied in α is false in I.*

Definition 5.6 *We will say that a semantic tableau T is satisfiable if there exists an interpretation I and a branch α of T so that I satisfies α.*

Lemma 5.1 One hop Lemma

If $T_1 \Rightarrow T_2$ and T_1 is satisfiable, then T_2 is satisfiable.

We suppose that T_1 is satisfiable. So there is some branch α_1 in T_1 so that α_1 is satisfiable.

T_2 is obtained from T_1 in one step, by one application of the rules. This means that there is some path β_1 in T_1 so that T_2 is obtained from T_1 by an extension of β_1. If $\alpha_1 \neq \beta_1$, then the branch α_1 is still in T_2, and so T_2 remains satisfiable. Now consider the case in which $\alpha_1 = \beta_1$.

We need to do is show that the rules preserve satisfiability. This can be done by considering each rule in turn. Suppose $T_1 \Rightarrow T_2$ using rule 1). Then branch α_1 of T_1 contains a node labelled $\neg(A \to B)$. T_2 is obtained by adding two nodes to α_1, and denying A on one and asserting B on the other. The branch α_1 in T_1 is extended to a branch α_2 in T_2. We suppose α_1 is satisfiable. The satisfying interpretation makes $A \to B$ false. Thus it must make A true and B false. Thus it must satisfy the extension of α_1 in T_2. Thus T_2 is satisfiable. The other cases are similar.

Corollary 5.1 *If $T_1 \Rightarrow^* T_k$ and T_1 is satisfiable, then T_k is satisfiable.*

Proof. By induction on the number of steps in the derivation, using the one hop lemma.

$\Box$.

Correctness

We can now prove the correctness part of the theorem above.

Suppose $\Gamma \vdash A$. Let T_1 be the initial tableau with Γ asserted and A denied. $T_1 \Rightarrow^* T_k$, where T_k is closed. Since T_k is closed, it has a contradiction on every branch, so it cannot be satisfied. By the corollary above, T_1 is not satisfiable. Thus, no interpretation which makes Γ true can make A false. Therefore, any interpretation which makes Γ true also makes A true. But this says exactly that A is a logical consequence of Γ. Thus $\Gamma \models A$.

So far we have $\Gamma \vdash A$ implies $\Gamma \models A$. In other words, we have correctness.

Completeness

We need to show that $\Gamma \vdash A$ whenever A is a logical consequence of Γ. This is the completeness part. The proof also uses contradiction ! Assume that $\Gamma \vdash A$ is false. This means that no closed tree can be obtained if we start with an initial tableau with Γ asserted and A denied. That means that no closed tree can be obtained from this initial tableau no matter how we apply the rules. Suppose we apply the rules systematically, as explained in the previous section; this means that if any rule can be applied at some node it is eventually applied, unless all the branches through the node are closed. We do not get a closed tree. If the construction terminates, we get a counterexample, and in this case A is not a logical consequence of Γ.

Suppose however that the construction never terminates, no matter what we do. The construction builds a tree with at least one infinite branch.

Such an infinite branch actually describes a counterexample, I claim. Suppose $T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow ....T_k \Rightarrow T_{k+1} \Rightarrow ...$ is the sequence of trees obtained by our non terminating systematic derivation. Let T_ω be the union of all these trees. T_ω has at least one infinite open branch. Let α be such a branch. The claim is that if the derivation has been done systematically, that is if any operation which could be done on α is eventually done, then α describes an interpretation, I, which satisfies T_ω, and thus satisfies T_1, since T_1 is actually the initial part of T_ω. For the domain of I we take the constant terms appearing in α, i.e. all the terms which do not have variables in them. Let D be this domain. D is syntactic, a set of strings of characters, but it is still quite acceptable as a mathematical domain. Note that D can have compound terms as well as constants. In fact, D is closed under application of the function symbols, since we were obliged to try all possible substitutions in the rules. If f is a function symbol of arity n, and $t_1, ..., t_n$ are in D, define the function $\mathbf{f}(t_1, ..., t_n)$ to be $f(t_1, ..., t_n)$. (Nothing could be simpler!) So far we have got a domain and defined the functions. We only need to define the predicates. If p is a predicate symbol of arity n define $\mathbf{p}(t_1, ..., t_n)$ to be true if and only if $p(t_1, ..., t_n)$ appears asserted of α. We now have an interpretation I. It can be shown that I satisfies α.

Lemma 5.2 *If B is asserted on α then $\models_I B$.*
If B is denied on α then $\models_I \neg B$.

Proof. The proof is by induction on the number of logical operators in B. For the basis of the induction, consider the case in which there are no logical operators in B, i.e. B is atomic. So B is just a predicate symbol applied to a list of terms. Note that I has been defined in order to verity the statement of the lemma. Now for the induction step. We will need to use complete induction. We therefore suppose that the lemma is true for sentences B with less than k logical operators, and we suppose that B has k logical operators. There are now seven cases, depending on the outermost operator of B. They are all quite easy.

Case B has the form $C \to D$. Assume B is asserted on α. Then, since the construction is assumed to be systematic, either C is denied on α or D is asserted. If C is denied, then, by induction hypothesis, $\models_I \neg C$. If D is asserted, then, by induction hypothesis, $\models_I D$. In either case $\models_I (C \to D)$. Thus $\models_I B$, which was to be proved.

The other cases you can do yourself.

□

So A is, again, not a logical consequence of Γ.

□

We mentioned earlier that there are a number of formal systems for proof in predicate logic. The Gödel theorem is true of all of them. (Of course the proofs are different.) So all these systems are, in a sense, equivalent.

Can we actually use a system such as semantic tableaux for deciding whether or not a sentence is a logical consequence of a set of axioms?

5.4.2 Unsolvability and intractability

The bad news is:

Theorem 5.2 *The problem of deciding whether or not an arbitrary sentence A of predicate logic is logically valid is recursively unsolvable.*

This means that there is no algorithm to decide whether or not $\models A$.

So obviously there can never be a computer program which, given a set Γ of axioms and a sentence A, will decide whether or not $\Gamma \models A$.

Even though we have a complete and sound formal system for deduction in predicate logic, we have to give up any hope of completely solving the logical consequence problem. On the other hand, there may be special versions of this problem which are of practical interest and which we can solve in practice.

5.5 Problems

Problem 5.2 *Give a definition for:*

term t is free for variable X in formula $A(X)$.

This should be done in such a way that it makes sense to substitute t for X in $A(X)$.

Problem 5.3 *Find a counterexample to the following, or give a proof. In each case, complete the semantic tableau construction, and state how many branches there are in the completed tree. How many are closed? Verify that each branch which is not closed defines a counterexample.*

a) $(((p \to \neg q) \to p) \to q)$

b) $((q \to \neg p) \to ((p \to (\neg q))))$

c) $((p \to q) \to ((\neg p) \to (\neg q)))$

Problem 5.4 *If we know that either A is true or B is true and we know that either B is true or C is false, and we know that A and C can't both be true, then it must happen that A is true. Either give a formal proof of this, or a counterexample.*

Problem 5.5 *Show whether or not the following set of statements is consistent.*

A and B and C implies that D is false unless E is true. E and not B implies C. A and B are equivalent. C is true.

Problem 5.6 *Prove this or construct a counterexample:*

A and (B or C or D) is logically equivalent to (A and B) or (A and C) or (A and D).

Either show that the following statements are logically valid, or provide a counterexample.

Problem 5.7 *Esmerelda is a duck and all ducks like ponds so Esmerelda likes ponds.*

Problem 5.8 *If all burglars are barbers and all barbers are bakers then all burglars are bakers.*

Problem 5.9 *If all burglars are barbers and no bakers are not barbers then some bakers are not burglars.*

Problem 5.10 *If all burglars are barbers and some barbers are bakers and some baker is not a burglar then some barber is not a burglar.*

5.6 Satisfaction Games

We earlier saw that truth of a statement in an interpretation can been understood in terms of a game between two people, a proponent and an opponent.

In a similar way we can understand the semantic tableaux method in terms of a game between two people a builder and a critic. Suppose we are given an initial set of sentences Γ. We don't have an interpretation. The builder wants to build an interpretation in which all the sentences in Γ are true. The critic wants this to fail. He hopes to confound the builder. When there is a branch the builder is allowed to choose which branch to follow. The builder also invents new names for the Skolem constants which have to be introduced. Otherwise the critic (with malicious intent) chooses which rule to apply next, and which substitutions to make. The original set of sentences are not satisfiable if and only if the critic has a winning strategy. Otherwise the builder has a winning strategy. But the builder wins if the game goes on forever. Even teh best human player imaginable will make mistakes in this very difficult game.

Example 5.4 $\Gamma = \{((\forall X)(\forall Y)(r(X,Y) \to r(Y,X)), (\exists X)\neg r(X,X), (\forall X)(\forall Y)(\forall Z)((r(X,Y) \wedge r(Y,Z)) \to r(X,Z))\}$.

5.7 Revision Problems

Problem 5.11 *The two statements* $(\forall X)(\exists Y)p(X,Y)$*, and* $(\exists Y)(\forall X)p(X,Y)$ *are not logically equivalent. Show that one of them implies the other using semantic tableaux. Give a counterexample to show that the implication in the other direction is not valid.*

Consider the following three statements.

1. $((\forall X)a(X) \to (\forall X)b(X))$
2. $(\exists X)(\forall Y)(a(X) \to b(Y))$
3. $(\forall Y)(\exists X)(a(X) \to b(Y))$

Problem 5.12 *Construct parse trees for each of the three statements above. Show the scope of each quantifier.*

Problem 5.13 *Use semantic tableaux to show that the first statement implies the second.*

Problem 5.14 *Use semantic tableaux to show that the first statement implies the third.*

Problem 5.15 *Show that the second statement implies the first.*

Problem 5.16 *Are the three statements logically equivalent?*

Problem 5.17 *Find prenex normal form of*
$((\exists X)a(X) \rightarrow (\exists X)b(X))$

Problem 5.18 *In our prenex normal form algorithm, we proceeded by first getting rid of implication, replacing $A \rightarrow B$ by $(\neg A \vee B)$, and then moving negations inside and then moving quantifiers outside of conjunctions and disjunctions. Show how to modify the algorithm, leaving implications in place, and moving quantifiers outside of implications.*

5.8 Solutions

1. To show
$((\forall X)a(X) \to (\forall X)b(X)) \models (\exists X)(\forall Y)(a(X) \to b(Y))$

```
                        1) (Forall X)a(X) -->(Forall X) b(X)
                        2) not(Exists X)(Forall Y)(a(X) --> b(Y))

           /                                  \

3) not (Forall X) a(X) from 1                 3) (Forall X) b(X) from 1
|                                             |
4) not a(tau)  from 3, Skolem tau             4) not (Forall Y)
                                                (a(rho) -->b(Y))  sub 2
                                              |
|
5) not (Forall Y)(a(tau) --> b(Y)) subs in 2  5) not(a(rho) --> b(alph))
                                                 from 4, Skolem alph
|                                             |
6)  not(a(tau) --> b(sig)) from 5, Skolem sig 6) a(rho)  from 5
|                                             |
7) a(tau), from 6, contradiction              7) not b(alph)  from 5
|                                             |
8) not b(sig) from 6                          8) b(alph) from 3, subs
Contradiction                                    Contradiction
```

The tree is closed. So there is no counterexample. Thus
$((\forall X)a(X) \to (\forall X)b(X)) \models (\exists X)(\forall Y)(a(X) \to b(Y))$

5.9 Summary

We have developed a complete and sound proof system for predicate logic. This is based on proof by contradiction. Given a set of axioms Γ and a sentence A, we attempt systematically to construct a counterexample, i.e. an interpretation in which A is false although Γ is true.

The data structures in this semantic tableau system are trees labelled with assertions or denials of formulae. The transformations are rules which extend the trees.

Chapter 6

Other formal deductive systems for first order logic

6.1 Deduction by Resolution and Unification

Suppose Γ is a set of sentences and A is a sentence. If A is not a logical consequence of Γ then there is an interpretation in which Γ is true and A is false. Thus $\Gamma \models A$ if and only if $\Gamma, \neg A$ is not satisfiable. So the logical consequence problem reduces to the problem of deciding whether or not a set of formulae is satisfiable.

Let Γ be any set of sentences, and let $c(\Gamma)$ be the clausal form of Γ. Γ is satisfiable if and only if $c(\Gamma)$ is satisfiable. (To say that $c(\Gamma)$ is satisfiable means that there is some interpretation in which all the clauses are universally true.) So the logical consequence problem reduces to deciding whether or not a clausal form is satisfiable.

Resolution is a deduction rule for clauses. Suppose we have two clauses:

$\Gamma_1 \to \Delta_1, C$

$C, \Gamma_2 \to \Delta_2$

with some formula C appearing on the left of one and on the right of the other. Assume that Γ_1 and Γ_2 are true. Then either the disjunction of Δ_1 is true or C is true. If C is true, then the disjunction of Δ_2 must be true. So we get the new clause

$\Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2$

and we say that the new clause is obtained by *resolution* of the original

two clauses. Note that although the new clause may be longer, it has one less atomic formula. We start with a pool of clauses in the original clausal form, and we add to the pool by using resolution. If we eventually get to the contradictory clause

$\rightarrow$

which has empty left and right hand sides, this implies that the original clausal form was contradictory, and thus not satisfiable.

We may also use substitution on clauses to obtain new clauses. In order to apply resolution we need pairs of clauses in which some formula C appears on the left of one and on the right of the other. The unification algorithm can be used to find substitutions which create these matches.

Resolution and unification can be used together to make a complete and correct deductive system for clausal forms. This is especially easy to apply in the special case in which all the clauses have exactly one atomic formula on the right hand side. This special case is the basis of the programming language prolog, which will be discussed in detail later.

6.2 The Sequent Calculus

In this section we will briefly give yet another complete and correct formal deductive system for predicate logic. This other system is called the sequent calculus. In the sequent calculus we start with a set of axioms which are obviously valid. The axioms are also considered to be theorems. We obtain new theorems by applying certain rules of inference.

The basic statement in this system is the sequent. A sequent has the form:

$A_1, ..., A_n \vdash B_1, ..., B_m$

where $A_1, ..., A_n, B_1, ..., B_m$ are formulae in some first order language. We will interpret such a statement as meaning:

$A_1 \wedge ... \wedge A_n \models B_1 \vee ... \vee B_m$

That is, the disjunction of the right hand side is a logical consequence of the conjunction of the left hand side. In general, we will consider the left and right hand sides of a sequent as sets; that is, the order of the formulae will not matter. We will consider an empty set on the left hand side as Truth, and the empty set on the right hand side as Falsity. Thus

$\vdash A, B, C$

means that $A \vee B \vee C$ is logically valid. On the other hand

$A, B, C \vdash$

means that $A \wedge B \wedge C$ is contradictory; that is, it is not possible to satisfy A, B, C simultaneously.

A sequent is like a clause except that quantifiers and negation are allowed.

A sequent is obviously valid if the same formula appears on both left and right hand sides.

We will use Γ and Δ as variables for sets of formulae.

In order to make this system similar to the semantic tableaux system, we will assume that all the formulae in sequents are sentences. That is, no free variables are allowed.

Definition 6.1 *A sequent of the form*

$\Gamma, A \vdash \Delta, A$

is an axiom of the sequent calculus.

The rules of inference will allow us to prove some sequents from others.

We have two rules for negation:

One of these rules is:

$\Gamma \vdash \Delta, A$

————————negation left

$\neg A, \Gamma \vdash \Delta$

The other rule for negation is:

$A, \Gamma \vdash \Delta$

———————negation right

$\Gamma \vdash \Delta, \neg A$

Please check that negation left and negation right are correct. Do you get a strange feeling of familiarity when you look at these? What rules should we have for implication?

$A, \Gamma \vdash \Delta, B$

———————implication right

$\Gamma \vdash \Delta, A \to B$

There is also an implication left rule, but it needs two sequents as premises.

$(\Gamma \vdash \Delta, A)$, $(B, \Gamma \vdash \Delta)$

————————————————————-implication left

$A \to B, \Gamma \vdash \Delta$

Please check again that these rules are correct.

Perhaps you can see the connection with semantic tableaux. Consider a branch of a semantic tableau. Let Δ_n be a subset of the formulae which occur on the branch and which begin with negation. Let Γ be the other formulae which occur on the branch. Obtain Δ from Δ_n by removing the initial negation. Now write

$\Gamma \vdash \Delta$.

This translates branches into sequents. The set Γ are the statements asserted on the branch and the Δ are the statements which are denied. A branch is impossible to satisfy if and only if the corresponding sequent is valid.

Notice that a branch is closed if and only if it contains a contradiction if and only if it translates into an axiom in the sequent calculus.

You can now see: the rules for the sequent calculus are obtained just by turning the semantic tableau rules upside down. A semantic tableau proof is just a sequent proof turned upside down. As an exercise, you should make sure that you can write down all the rules of inference for the sequent calculus. Look at the quantifiers!

Please note that if you actually want to find a proof of a given sequent in the sequent calculus, the best way to do this is usually to tackle the problem backwards using semantic tableaux. Once you get a closed tableau, you turn this upside down to get a sequent proof.

As an example, we could prove $A \vdash (B \to A)$ in the semantic tableau system, and then obtain the following two step proof in the sequent calculus:

$A, B, \vdash (B \to A), A$ Axiom
$A \vdash (B \to A)$ implication: right

We could also have used the simpler axiom:
$A, B \vdash A$

6.3 Other deductive systems

There are many other correct and complete deductive systems for first order logic. An important example is natural deduction, which is described in is the book by Huth and Ryan mentioned in the references.

Chapter 7

A Formalisation of Mathematics: ZF set theory

There have been several attempts to formalise mathematics. The most successful of these is Zermelo Fraenkel set theory. Anyone who has studied pure or discrete mathematics will be aware of the existence of this system as a mathematical background and framework. It has had a very strong influence on the development of mathematics. A rough description of ZF set theory is given below in a few paragraphs. This famous formal system consists of a formal language, L_{ZF}, together with a set of axioms and some formal rules of deduction.

The language L_{ZF} consists of all statements, called formulae, which can be built up from equality statements, such as $X = Y$, and set membership statements, such as $X \in Y$, using the logical operators not, or, and, implies, iff, for all and there exists, which are written $\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \forall, \exists$, respectively. For example, to say that $X = Y$ if and only if every element of X is an element of Y and vica versa, we could write, in L_{ZF}:
$(X = Y \leftrightarrow (\forall W)(W \in X \leftrightarrow W \in Y))$.

The axioms for ZF set theory say, essentially, that certain basic sets exist, and that sets are closed under certain operations. Here is a list of the axioms, with some redundancies.

1. Equality axiom. For sets A and B, $A = B$ if and only if $(\forall X)(X \in A \leftrightarrow X \in B)$.

2. Singleton axiom. For any A , there is a set $\{A\}$, called singleton A,

which is the set whose only element is A.

3. Union axiom. If A and B are sets, so also is $A \cup B$, the union of A and B. It is also assumed that if F is a set whose elements are also sets, then there is a set $\cup F = \{Y : (\exists X)(X \in F \wedge Y \in X\}$. In other words, $\cup F$ is the union of all the elements of F.

4. Intersection axiom. If A and B are sets, so also is $A \cap B$, the intersection of A and B. It is also assumed that if F is a set whose elements are also sets, then there is a set $\cap F = \{Y : (\forall X)(X \in F \to Y \in X)$.

5. Set difference axiom. If A and B are sets, so also is $A/B = \{X : X \in A \wedge \neg X \in B\}$. This is the set difference of A and B.

6. Ordered pair axiom. For any A and B, there is an ordered pair (A, B), with first element A and second element B. $(A, B) = (C, D)$ if and only if $A = C$ and $B = D$.

7. Cartesian product axiom. If A and B are sets, so is $A \times B$, the Cartesian product of A and B, i.e. the set of ordered pairs (a, b), with a in A and b in B.

8. Power set axiom. If A is a set, so is $P(A)$, the set of all subsets of a set A. This is called the power set of A.

9. Function space axiom. If A and B are sets, so is $A \to B$, the set of functions from A to B.

10. Comprehension axiom: If A is a set and $C(X)$ is a condition on X which we can write in L_{ZF}, then we can define $\{X : X \in A \wedge C(X)\}$. The comprehension axiom picks out that subset of A for which $C(X)$ is true.

11. Replacement axiom: It is also assumed that if we have succeeded, within L_{ZF}, in defining a function $f(x)$ on a set A, then we can construct a new set as $\{f(x) : x \in A\}$, i.e. the image of f applied to A.

12. Axiom of choice: If F is a set whose elements are disjoint non empty sets, there exists a set $choice(F)$ which has exactly one element in common with each element of F.

13. There is an empty set, denoted $\emptyset$, which has no elements. Also, there exists the set of natural numbers $\mathbf{N} = \{0, 1, 2, ...\}$.

The attentive reader may notice that some of the axioms, as given above can be proved from others. For example, the existence of $A \cap B$ can be proved from the axiom of comprehension, since $A \cap B = \{X : X \in A \wedge X \in B\}$. And in fact the comprehension axiom itself can be proved from the axiom of replacement. I have stated the axioms with these redundancies since I thought they might help the reader to see what was happening. The main sense of the axioms is that we can build up complex sets from simpler ones using common operations.

The axioms of ZF set theory are written in our special language, L_{ZF}. For example, the axiom which says that for any set A there exists a set, singleton A, whose only element is A, would be written as follows:

$(\forall A)(\exists B)(\forall W)(W \in B \leftrightarrow W = A)$

In order to write the axioms entirely in L_{ZF}, rather than in the mixture of L_{ZF} and informal mathematical English used above, we need to define functions, ordered pairs, and the natural numbers in terms of L_{ZF}. That is to say, we need to express these ideas in terms of set membership and equality.

The definition of function is especially important.

Definition 7.1 *A set f is a function if f is a set of ordered pairs such that whenever $(X, Y) \in f$ and $(X, Z) \in f$ then $Y = Z$.*

In other words a function is defined to be a set of ordered pairs with the property that the second element is determined by the first element.

We will write $f : A \to B$ and say that f is a function with domain A and codomain B if A is the set of first elements of f and if all the second elements of f are in B.

Roughly speaking, ZF set theory considers functions not as processes but as look-up tables. For example, the squaring function on the natural numbers is the set

$\{(n, n^2) : n \in \{\mathbf{N}\}\}$

In order to write the above axioms in L_{ZF}, we also need to represent ordered pairs in general and the natural numbers in particular as sets. There are a number of ways of doing this. For example, we could define the ordered pair (X, Y) to be the same as the set $\{\{X\}, \{X, Y\}\}$.

To represent the natural numbers we could identify 0 with $\emptyset$, and $n + 1$ with $n \cup \{n\}$. Then $\mathbf{N}$ is the smallest set so that $\emptyset \in \mathbf{N}$ and whenever $X \in \mathbf{N}$ so is $X \cup \{X\}$.

In the usual version of ZF set theory all the above representations are used.

From the natural numbers, the axioms allow us to construct the set $\mathbf{Z}$ of integers, the set $\mathbf{Q}$ of rational numbers, and the set $\mathbf{R}$ of real numbers.

The rules of deduction are, of course, formal rules, which can be applied mechanically. The theorems of ZF set theory are the set of formulae which can be obtained by starting with the axioms and applying the rules of inference.

The original specification for this formal system was quite amazing. It was that the theorems of this system should contain no contradictions, and should contain the translations into L_{ZF} of *all the currently accepted theorems of mathematics, together with all the translations into* L_{ZF} *of all theorems of mathematics which may become accepted in the future.* So ZF set theory is an attempt to characterise the apparently ultimate infinite object of mathematical reality.

The Occam's razor principle was applied very seriously by the developers of ZF set theory. Since every statement in L_{ZF} is built up from set membership and equality, and the logical operations $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \forall, \exists$, all mathematical ideas must also be constructed from these elements.

In order to get simplicity, some aspects of naturalness were sacrificed. For example, in order to keep the basic elements minimal, the natural numbers are built up from the empty set in a way which may seem arbitrary: 0 is $\emptyset$, 1 is $\{O\}$, 2 is $\{O, \{O\}\}$, and, in general, $n + 1$ is $n \cup \{n\}$.

So in ZF set theory, The natural numbers is

$\{O, \{O\}, \{O, \{O\}\}, ...\}$

which does not seem friendly or necessary. We do lose something if we adopt a nasty notation, such as this. A good notation encourages correct and fluent thinking.

Some of these problems of notation can be overcome without much difficulty, by extending L_{ZF}, adding names of important objects and operations. So we can define 0 to mean $\emptyset$, 1 to mean $\{O\}$, and so on. Similarly, without serious difficulty, we can extend the language L_{ZF} to include familiar notations such as $\{X\}, X \cup Y, X \cap Y, X \subseteq Y$, etc. The point is that these notations can be unambiguously defined in the original L_{ZF}. For example, we can define: $X \subseteq Y$ to mean $(\forall W)(W \in X \rightarrow W \in Y)$.

There is a more serious problem about the definition of functions in ZF set theory.

The designers of this system restricted themselves to what could be said with set membership, equality, and the logical operators. Once this restriction had been accepted, they had to explain the notion of function in these terms.

The problem with defining a function as a kind of set is that, intuitively, functions are active and dynamic, and sets are static. However Occam's razor was given priority over intuition in this case. Consequently, a function in ZF set theory is defined to be a set of ordered pairs, corresponding to the graph of the intuitive function, as described above.

The ZF notion of function is useful but not entirely satisfactory for computing. From the point of view of computing, it might be worth having a more complicated idea which was closer to intuition. The lambda calculus, discussed in another course, is an attempt to supply such an idea.

Unfortunately, the completeness and correctness status of ZF set theory is also less than ideal. We do not know whether or not ZF set theory is consistent. So $0 = 1$ might be a theorem, for all we know.

We do know, due to the wonderful work of K. Gödel, that ZF set theory can not prove itself consistent, unless it is actually inconsistent. So either there is a mathematical truth which is not a theorem, or correctness fails. This is discussed later.

Nevertheless, ZF set theory is a great accomplishment. It does seem successfully to formalise almost all of contemporary mathematics. It gives us a universally recognised language for mathematics and a universally recognised standard for rigorous proof. If two mathematicians disagree about the meaning of a term or about the validity of a proof, they can, in principle, resolve their differences by a process, called formalisation, of translating their ideas into ZF set theory. The existence of the ZF formalism is one reason for the great progress which has been made in mathematics in the last few generations.

A dialect of L_{ZF}, called the Z specification language, can also be used to write unambiguous specifications for computer programs. Since it is formal, the Z specification language can also be read, transformed, and processed by computer.

Chapter 8

Gödel Incompleteness theorems

We previously defined L_N, the language of first order arithmetic, and described an infinite axiom set, Γ, which is called the Peano postulates. A sentence, S, of L_N is provable from the Peano postulates if

$\Gamma \vdash S$

We are really interested in the truths in the standard interpretation $\mathbf{N}$. We have $\Gamma \vdash S$ implies $\Gamma \models S$ and this implies $\models_{\mathbf{N}} S$.

Theorem 8.1 *(Gödel Incompleteness theorem for Arithmetic.) There is a sentence S which is true in the standard interpretation but cannot be proved from the Peano postulates.*

Here is a sketch of the proof.

The main idea is to find a way to look at formulae of L_N so that they can be understood as making comments about their own provability.

Gödel begins by showing that each formula, A of L_N can be coded as a natural number $n = g(A)$. The number n is now called the Gödel number of the formula A. Similarly, we can find Gödel numbers for proofs from the Peano postulates. This means that we can find a formula $Pr(X, Y)$ with two variables, which is true for natural numbers X and Y if and only if X is the Gödel number of a proof from the Peano postulates of a formula which has the Gödel number Y. Please notice that $Pr(X, Y)$ is an actual formula of L_N, and it says something about the natural numbers X and Y in terms of addition, multiplication, equality, and the usual logical operators. But we know that it has another interpretation also, because of the way it was

constructed. This one piece of syntax with two different meanings is typical Gödelian lateral thinking.

We can say that Y is the number of a provable formula by $(\exists X)Pr(X,Y)$.

We can now find a formula $diag(X)$ which says that X is the Gödel number of a formula $A(Z)$ with one free variable and if $n = g(A(Z)$ then $A(n)$ is not provable from the Peano postulates. Now let $n = g(diag(X))$. We see that $diag(n)$ is a sentence of L_N. One of its meanings is that its own self is not provable. $diag(n)$ is either true or false in the standard interpretation. If it is false, it is also provable from the Peano postulates. But the Peano postulates are all true in the standard interpretation, and deduction preserves truth. Therefore $\models_{\mathbf{N}} diag(n)$. Also $diag(n)$ is not provable from the Peano postulates.

Remarks:

These results also apply to Zermelo Fraenkel set theory. In fact there is no recursive axiomatisation which is complete for any extension of Peano arithmetic.

It may be called common sense that human knowledge is limited. The incompleteness theorems of Gödel prove this rigorously.

Chapter 9

Logic programming (prolog)

What sort of computational use can we make of the semantic tableau idea?

9.1 Headed Horn clauses

Suppose we have some finite set of axioms Γ, and we wish to somehow compute the logical consequences of these. Note that we are not working in the sequent calculus at this point. Γ is just a set of formulae; in fact we will assume that the axioms in Γ are all sentences. It seems reasonable to begin by putting Γ in clausal form.

Suppose we have done this. In what cases is it feasible to use the semantic tableau method to find the consequences of Γ? To put this question in another way, when the semantic tableau method goes wrong, how does it do this?

The problem with the semantic tableau method is that it creates a lot of branches. In some cases these branches proliferate until they become unmanageable. In order to implement a theorem prover for predicate logic, it seems necessary to look for special cases in which we hope the computation will not get too complicated. (Or at least we can regard this as a reasonable first step.)

Definition 9.1 *A headed Horn clause is one of the form:*
$(A_1 \wedge A_2 \wedge ... \wedge A_k) \rightarrow B$

Headed Horn clauses are especially simple because the conclusion is a single statement, rather than a list of alternatives.

Example 9.1 *We will write a headed Horn clause* $(A1 \wedge A2 \wedge ... \wedge An) \rightarrow B$ *in the form B if A1, A2, ... An.*

Suppose our axioms are $(p \wedge r) \rightarrow q, r \rightarrow p,$ *and* r. *Can we prove* q*?. We can proceed as follows:*

```
                (1) q if p,r
                 |
                (2) p if r
                 |
                (3) r
                 |
                (4) not q
              /    \

          (5) q      (5)not( p,r)
        contr       /   \

                 (6) not p  (6) not r, contr
               /    \

(7, from 2) p      (7) not r contr
contr
```

We take the not q of the original tableau as a goal. We match this with one of the conclusions of the clauses which are asserted, in this case with the conclusion of clause (1). The premises of clause (1) then become sub goals. So we have subgoals p and r. *Proceeding recursively, we match p with one of the conclusions of the clauses. In this case the match is with the conclusion of clause (2). We generate another subgoal, namely* r. *This matches with one of our original premises, so one branch gets closed. The next branch gets closed in the same way.*

It seems that in the case when our axioms are Headed Horn clauses, we have a reasonable search strategy. That is, we seem to know what to do next. The main idea of prolog is to try to exploit this.

From now on, we will suppose Γ is a list of headed Horn clauses which are universally quantified, and A is an existentially quantified conjunction of atomic formulae.

We are trying to decide $\Gamma \models A$.

From now on we will also omit universal quantifiers of statements asserted in our trees, and we will omit existential quantifiers on statements which are denied. (So free variables in asserted formulae are tacitly assumed to be universally quantified, and free variables ion formulae which are denied are tacitly assumed to be existentially quantified.)

Example 9.2 *Suppose* Γ *is*
$\{(\forall X)(\forall Y)(duck(X) \wedge pond(Y)) \rightarrow likes(X, Y)$
$duck(esmerelda), pond(estero), pond(emeryville)\}$,
and A *is* $(\exists Z)likes(Z, estero)$.
We would write the initial tree as:

```
(1) likes (X,Y if duck(X),  pond(Y)

 |
(2) duck (esmerelda)
 |

(3)pond(estero)

 |
(4) pond(emeryville)

(5) not likes(Z, estero)
```

Our convention that constants begin with lower case and variables begin with upper case is very useful here, since it allows us to distinguish constants and variables.

We wish to decide whether or not $\Gamma \models A$. We will need to look for good substitutions which may lead to a closed tree.
The prolog strategy is the following.
Suppose Γ is C1, C2,..., Ck, where each Ci is a headed Horn clause.

1. Put Γ, the list of headed Horn clauses asserted and $\neg A$, a conjunction of atomic formulae denied in the initial tree.

2. If A is the conjunction of n atomic formulae, split $\neg A$ into n branches. The i th branch has Γ asserted and has the i th conjunct of A denied.

3. Pick the leftmost open branch. Let's say this has A1 denied.

4. Scan down the list Γ. For each Ci in Γ, try to find a substitution which unifies A1 and the conclusion of Ci.

5. If we get to the end of Γ without success, return FAIL.

6. Suppose we find a substitution α which unifies A1 and the conclusion of Ci. Let Ci be:

   B if R1, R2, ..., Ra

   so Bα= A1 α. Split Ci in the subteee below A1 so that we get:

```
         ()   A1
        /   \
not B       not(R1 , R2 , ... Ra)
```

   Make the substitution α so that the left hand branch of this closes.

   On the right hand branch we have a new denied conjunction of atomic formulae

   not (R1, R2, ... , Ra) α

7. Continue recursively with the new tree, after application of substitution α.

8. If we do not get closure of the new subtree below A1, backtrack, delete this subtree, undo the substitution α and continue scanning Γ below Ci.

9. If we do get closure, report the final substitution of terms for variables.

9.2 Predicates, facts, constants, variables in prolog

In the following, a "character" is either a letter or a digit.

A *constant* in prolog is either a number, such as "776", or a string of characters beginning with a lower case letter. So, for example, "jane" is a constant, but "Jane" is not a constant.

A *variable* in prolog is a string of characters beginning with an upper case letter.

Prolog also has function and predicate symbols, just like first order languages. We can use any string of characters starting with a lower case letter as the name for a function or a predicate. As in any term language, we can have complex terms built up out of simpler ones. So for example

$f(g(X, hat), Y, Z, h(p, q))$

is a possible prolog term. Prolog also has some built in function symbols. Lists are especially important. In prolog, lists are written surrounded with square brackets and separated by commas. So, for example,

[cat,dog,horse]

is a list of three items. Lists can also have other lists as components. So

[hat,[big,yellow],got]

is a list whose first component is a constant, and whose second component is a list. Lists are terms and lists can have any terms as components. There is am empty list, written [ ].

Predicate names in prolog are strings of characters beginning with lower case letters.

You should try to keep in mind the distinction between predicates, which are semantic things, functions whose codomain is $\{T, F\}$, and predicate names, which are defined by syntax.

Predicate expressions are usually written as predicate names followed by a list of terms enclosed in brackets. For example

$red(cup)$

says that some particular object, called "cup" is red.

There is no systematic way to distinguish function names from predicate names in prolog.

Prolog has a few built in predicates, whose names are already determined. An important built in predicate is equality, written

```
X = Y.
```

Inequality, written

```
X \= Y.
```

is also built in.

As long as we avoid names of built in predicates, we can invent predicate names to suit ourselves. Of course it is necessary to use whatever names we invent in a consistent manner. Suppose, for example, we wish to write some axioms for motherhood. We might begin by asking ourselves: what is the arity of motherhood? We seem to need a predicate of arity 2, which

says that one person is the mother of another. We could decide to write this as

```
qqqqqzsz(A,B).
```

However, this will be hard to remember, and it seems better to use a more natural name. We could call our predicate

```
mother(A,B).
```

This is clearly meant to say either that A is the mother of B, or vica versa. Obviously, we have to determine which we mean. So we have to invent some convention to determine, in our own minds, the order of the arguments. We might decide, for example, that mother(A,B) should mean that A is the mother of B. Having set up this convention, we have to hold to it.

A *fact* in prolog is a predicate expression in which all the terms are constant. For example

```
mother(louise, mabel)
```

is a fact. To say that this is a fact does not, in this context, imply that we think it is true. It may be true or it may be false. A fact, in this context, is an expression which has the form

```
predicate-name(constant-term,..., constant-term).
```

To give another example,

```
likes(joe, alice)
```

which is intended to say that joe likes alice, is a fact.

A predicate of arity zero is written as

```
p
```

9.3 Programs in prolog

A prolog program may be regarded as a set of axioms for some field of knowledge. When the program is consulted we ask whether some statement is a logical consequence of the axioms. If the statement has variables in it we ask for assignments of values to the variables so that the statement is a logical consequence of the axioms.

A prolog program consists of a list of statements, with full stops at the end of each. Example:

```
mortal(X) :- human(X).

featherless(socrates).

bipedal(socrates).

animal(socrates).

human(X) :- featherless(X), bipedal(X), animal(X).
```

As mentioned above, terms starting with upper case letters, such as X, are variables, and other names such as socrates are constants. human(X) is a predicate expression with variable X, meaning X is human. Note that predicate names, such as human, mortal, featherless, should start with lower case letters.

In prolog, the sign :- is interpreted as if. So the first line in the example program above means X is mortal if X is human, for all X. The next three lines say that socrates is a featherless bipedal animal. The comma , is interpreted as and so the last line says that X is human if X is a featherless bipedal animal, for all X.

Among the five statements in the above prolog program, the statements on lines 2,3,4 are facts. The other statements are called rules. A *rule* in prolog, in general, is a statement of the form

```
p :- q1,q2,..., qn.
```

where

```
p, q1, q2,..., qn
```

are predicate expressions. The meaning of this rule is that p is true whenever all of q1,..., qn are true.

Statements in prolog are only of two types, either facts or rules. Prolog statements are also called *clauses*.

Both rules and facts are headed Horn clauses.

In a rule, such as

```
mortal(X) :- human(X).
```

the left hand side is called the conclusion. A rule is supposed to state a truth and also show a possible way to prove the conclusion. So, for example,

```
human(X):- featherless(X), bipedal(X), animal(X).
```

expresses the truth that any featherless bipedal animal is human, and also tells us that if we have an X and want to show

```
human(X)
```

we can do this by checking

```
featherless(X), bipedal(X), animal(X).
```

Note that different occurrences of a variable in a single rule must refer to the same object. If we have the above rule and we want to prove

```
human(fred).
```

we need to check

```
featherless(fred), bipedal(fred), animal(fred).
```

On the other hand, occurrences of the same variable in different rules are not linked at all. The program given above would have the same meaning if the first line were changed to

```
mortal(Y):- human(Y).
```

Exercise: You can create a prolog program with any editor. Try this.

Once a program has been written, we may wish to ask questions about its logical consequences. This process is called consultation and is done with a prolog interpreter and/or compiler.

The reader should at this point discover how to get access to some version of prolog.

At Bath University on the BUCS machines, prolog can be run by typing pl.

Suppose you have written a prolog program. Give whatever local command is necessary to run prolog.

Once prolog is running, type

```
 [file].
```

if file is the name of the program you have written. You can then ask questions, and you should get logical consequences of the facts and rules you have given. For example if you ask mortal(socrates). after reading in the above example, it should, (after thinking for a while), say yes. If you ask

mortal(X).
it should eventually satisfy this with
X=socrates.
To get out of prolog type

```
halt.
```

Note that a full stop is necessary after each statement.
Now, try to create a program and consult it.

9.3.1 Comments in Prolog

For benefit of a human reader, comments may be added to prolog code as follows:

```
/*  This is a comment which might continue over
    several lines  */

%  This is also a comment, but  on one line.
```

9.4 How prolog works

Define the head of a headed Horn clause to be the conclusion, and define the tail to be the list of premises.
So, for example, in
perfect :- beautiful, surprising, useful, at-right-time.
the head is
perfect
and the tail is
beautiful, surprising, useful, at-right-time.
In section 9.1, we explained how prolog works in terms of semantic tableaux. We can also understand prolog's working as a special case of resolution and unification. We can also understand how prolog works in terms of a technique for satisfying goals, given a program Γ. Each clause in Γ is a headed Horn clause. Such a clause,
$A : -B_1, \ldots, B_n$
means that if our goal is to staisfy the head A, we can do this by satisfying the tail $B_1, \ldots, B_n$. (There may, of course, be other ways to

satisfy A.) So from goal A we can get subgoals $B_1, \ldots, B_n$, and we continue recursively.

Suppose G is our current list of goals. We try to find a substitution α so that $\Gamma \models G_\alpha$. We deal with each goal in G in turn. Suppose the first goal is G_1. We try to match G_1 with the head of a clause in Γ, starting at the beginning of Γ. Matching is done by unification. Suppose the first match is found a line k of Γ. Suppose this is

$A : -B_1, \ldots, B_n$

The unification algorithm (assumed to succeed in this case) gives us a substitution α which is a most general unifier between G_1 and A. So we take $(B_1)_\alpha, \ldots, (B_n)_\alpha$ as subgoals. We remember (in case we have to backtrack later) that substitution α was applied at line k. We continue recursively, attempting to satisfy the subgoals.

Suppose this succeeds, with final substitution γ, (which includes α). So $\Gamma \models (G_1)_\gamma$. We now apply γ to the tail of G and continue recursively. If the tail is empty, we report the final substitution γ so that $\Gamma \models G_\gamma$.

Suppose however that we fail to satisfy subgoals $(B_1, \ldots, B_n)_\alpha$. In this case we backtrack. This means that we undo substitution α and return to line k of Γ. We look below line k for another match with goal G_1. If none is found, we report "no". This means that Γ does not logically imply the existence of a way to satisfy G.

9.5 Programs with just facts

According to what was said above, a prolog program is a list of statements, which are either facts or rules. In particular, a list of facts is a prolog program. Suppose, for example, we had a list of facts of the form

criminal(X), meaning that X is a criminal

locate(X,City, Date), meaning that X is known to have been in City on Date, and

associate(X,Y), meaning that persons X and Y are associates.

We could just list all our facts in a program. For example,

```
criminal(boxcarjoe).
criminal(bigred).
criminal(eaglehat).
locate(bigred,ny,july3).
locate(bigred,sf,june4).
associate(bigred,boxcarjoe).
```

```
locate(boxcarjoe,milan,june5).
```

etc.

We store this in some file. We then call prolog. We read in the file with

```
['filename'].
```

We can then ask questions. If we ask, for example,

```
criminal(bigred).
```

we get the answer

```
yes.
```

We can also ask questions with variables, such as

```
criminal(X).
```

In this case we would get the first criminal

```
 X = boxcarjoe.
```

In general, given a query with variables in it, prolog attempts to find values of the variables which make the query true. If we want another example, we type

;

and we get the next crook, bigred. If we type ; again we get the next one. When the list is exhausted, we get answer

```
no.
```

We can also ask compound queries. Suppose, for example we are concerned to discover a criminal associate of bigred who was in milan on some date.

We ask

```
associate(bigred,X), criminal(X), locate(X,milan,Date).
```

and we will get an answer such as

```
X=boxcarjoe, Date=june5.
```

9.6 Backtracking

When we were working with semantic tableaux, we made substitutions and attempted to close branches of our trees. If a substitution did not result in closure, we tried another substitution; we never deleted anything. But prolog does delete the results of unsuccessful substitutions.

Suppose the current goal of prolog is query $q(X_1, ..., X_n)$. prolog scans down its program trying to unify the query with one of its facts or rules. Suppose the first match is with the head of a rule, with unifier σ, at line k of the program. The unifier σ is applied to the tail of the rule, and the conditions in this tail become new goals. Prolog continues recursively, starting with the first subgoal in the tail. If this attempt ultimately fails, prolog backtracks. This means that it discards the substitution σ and resumes its attempt to match the goal $q(X_1, ..., X_n)$ just below the last match, i.e. at line $k + 1$ in the program.

Consider, for example, the query

```
criminal(X), locate(X,ny,july3).
```

with the program given above. This will first try the substitution $[X := boxcarjoe]$; but this will not succeed, and backtracking will ensue.

9.7 Warning: problems with prolog

It would be nice if prolog really implemented predicate logic. However, this is not the case. Prolog sometimes works correctly and sometimes does not. Since we have a definition of logical consequence, we know exactly what it means to say that prolog works correctly. We would hope that prolog would show that A is a consequence of Γ if and only if $\Gamma \models A$, as long as Γ is a list of headed Horn clauses, and A is atomic. But no such luck.

Prolog uses a certain strategy for trying to get closure of its proof trees. This enables it to finish quickly in some cases, but causes problems in other cases. For example, suppose we are trying to decide whether or not

$(p \to p) \models p$.

We could do this by semantic tableau. We assert $(p \to p)$ and deny p ; we get one split, and stop. One branch is closed, the other is open, and by setting p to FALSE we get a counterexample.

What does prolog do?

It begins with

```
(1) p:- p.
 |
(2) not p.
```

The goal is p. This is matched with the head of p:-p, and we obtain a subgoal of p. Unfortunately, prolog now continues recursively, and so it never terminates. It comes to no conclusion. Prolog is unable to recognise that it is considering a goal which it has considered previously.

Obviously there are many ways in which prolog can tie itself into such infinite loops, and it frequently does so. In predicate logic, we often consider symmetric relations. For example,

married(X,Y):- married(Y,X).

To write such a rule in prolog would be to invite non termination.

To give another example, we could define fatherhood in terms of maleness and parenthood.

father(X,Y):- male(X), parent(X,Y).

Of course it is also true that fatherhood implies parenthood. We would include this if we were writing axioms of fatherhood in predicate logic. But in a prolog program, we would be unwise to have both

father(X,Y):- male(X), parent(X,Y).

and
parent(X,Y):- father(X,Y).
since this would be likely to cause non termination.

Such difficulties are very annoying. There are ways of alleviating such problems, but the alleviations are almost as irritating as the original problems. It seems to me that it is fair to say that prolog is a reasonable first step toward implementing predicate logic, but that, at present, it is extremely limited.

9.8 How to write simple prolog

Suppose we have a situation which we wish to describe in prolog. We first decide what are the important predicates in the situation. We then think about the predicates to try to see which ones can be defined in terms of the others. We should arrange the predicates in a hierarchy of complexity, with complex predicates defined in terms of simpler ones. Usually there are many different ways to do this! If possible, all the facts should be entered using the simplest predicates. If possible, the simplest predicates should be logically independent, so that no single piece of information is held in two different ways. These are *desirable* features, which can't all be realized in every case. So writing a prolog program is a problem in design, and different people will have different solutions, and some solutions will be simpler and more elegant than others. There may not exist any perfect solution.

In general, it is best to put the facts before the rules in a prolog program. It is best to put definitions of relatively simple predicates before definitions of relatively complex predicates. In other words, the order of the program should reflect, if possible, the hierarchy of complexity which you are using. This usually makes the program easier to read for a human being. And in some cases, as we will see, it may also improve the computational behaviour of the program.

9.9 Examples

Imagine a small community where everyone is related to everyone else in several different ways. We wish to express the relationships in a prolog program.

Some of the predicates we might be concerned with here are: parent(X,Y), father(X,Y), mother(X,Y), male(X), female(X), sister(X,Y), grandfather(X,Y), aunt(X,Y), cousin(X,Y). Evidently there are many logical relationships, for example:

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
grandfather(X,Y) :- father(X,Z),
                    parent(Z,Y).
```

The problem is to decide which are the basic predicates, and then to build up definitions of the more complex predicates in terms of the simpler ones.

To begin with, suppose we took the above definition of parent in terms of father and mother, and of grandfather in terms of father and parent. Suppose we added some facts to the program given above.

```
father(jud,rubel).
father(rubel,jubel).
parent(jubel,marx).
parent(rubel,jed).
father(jubel,lu).
parent(marx,mabel).

parent(X,Y) :- mother(X,Y).

parent(X,Y) :- father(X,Y).

grandfather(X,Y) :- father(X,Z),

                    parent(Z,Y).
```

If the above sample were consulted with

```
grandfather(rubel, lu).
```

the answer should be

```
yes.
```

We can also have questions with variables in them, e.g.

```
grandfather(X,Y).
```

In this case, prolog will try to find values of X and Y which make this true. It might reply

```
X=jud, Y=jubel.
```

If you then type

```
;
```

you get another possibility such as

```
X=rubel, Y=marx.
```

By giving more semicolons, you should get all the possibilities, one after the other.

You can also have compound queries, such as

```
grandfather(X,Y), parent(Y,lu).
```

And again all the possibilities should be generated,e.g. X=jud,Y=jubel.
Here is another example.

Example 9.3 ————————————

```
cat(bags).
cat(felix).
bird(harold).
hunt(X,Y) : - cat(X), bird(Y).
```

————————————

Suppose we ask the question:

```
hunt(felix, harold).
```

Prolog matches this goal with the left hand side of the first rule, using unifying substitution $\alpha = (X, felix), (Y, harold)$*. It then has two subgoals*

```
cat(felix), bird(harold).
```

It satisfies these from left to right, and eventually says "yes".

In this case there were no variables in the query. If we had asked

```
hunt(felix,W).
```

prolog would have eventually said

```
yes, W=harold.
```

You can see what prolog is doing, step by step, by turning on the trace. This is done as follows:

```
trace, hunt(felix,harold).
```

It may happen that the same variable is used in two different statements in a program. In order to prevent clashes in the substitutions, most versions of prolog will begin by renaming the variables. So you will find that new, perhaps peculiar variable names are used in trace.

The trace can be turned off by calling

```
notrace.
```

9.10 Circular definitions and Recursion

You are lost in the country and you ask someone how to get to the old Stanley place. “You go down Stanley road, “ you are told. So you ask: How do you get to Stanley Road? “It goes right past the old Stanley Place”, you are told.

This type of muddled definition or description will be called circular. The person who has given it actually has the information which is wanted, and has in fact said something possibly useful, but it is not sufficient to find what we want. Compare that with the following, possibly from an obscure nineteenth century anthropological work.

“If the relation Kram holds between a man and woman they are not allowed to marry. Kram holds between a man and woman if they share the same mother. Kram also holds between a man and woman, if Kram holds between the mother of the man and the father of the woman. Kram never happens for any other reasons.”

This definition might at first sight seem to be circular. Kram is defined in terms of another instance of Kram. If you look at it carefully, however, you will see that the explanation is arranged quite subtly to give us just the right amount of information to let us know what Kram is. All the information has been conveyed in a very compact way in this definition.

It is quite common for concepts to be self referential. A predicate at one set of arguments is defined in terms of itself, but with other instances of the arguments. This is a powerful technique but is obviously easy to misuse. The problem is that a person attempting to apply the definition may be sent off into considering an infinite loop of possibilities. A definition or description which is self referential but eventually terminating is called

recursive. On the other hand, a muddled self referential definition will be called circular.

Consider a definition of $n!$. We could say that $0! = 1$. Also, $(X')! = (X' * (X!))$. This recursive definition gives us an algorithm to calculate $n!$ for any natural number n. We get

$7! = 7*6! = 7*6*5! = 7*6*5*4! = 7*6*5*4*3! = 7*6*5*4*3*2! = 7*6*5*4*3*2*1! = 7*6*5*4*3*2*1*0! = 7*6*5*4*3*2*1.$

In terms of prolog, we will say that a rule is *recursive* if the same predicate appears on both the left and right hand sides. Of course the arguments on the two sides may be different. In fact if the arguments are not different, the rule is obviously circular, i.e. muddled.

Suppose, for example that we have a prolog program in which parent(X,Y) is defined. We want to define “Z is a descendent of X”; This should mean that either X is a parent of Z, or X is a parent of a parent of Z, or X is a parent of a parent of Z or ....

We can express this recursively as follows.

————————-

```
descendent(Z,X) :- parent(X,Z).

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

————————

9.11 The Cut, and negation

Before reading this section, you should be sure that you understand the idea of backtracking, which was explained above.

Example 9.4 ————————-

```
q(a).
s(a).
p(X) :- q(X), r(X).
p(Y) :- s(Y).
```

———————

Suppose we ask

p(a).

Prolog will find a match with the first rule, using $\alpha = \{(X, a)\}$*. It then has subgoals*

q(a), r(a).

It satisfies q(a), but then fails to satisfy r(a). So it backtracks, and discards the substitution α*; it gives up on the first rule and tries the second. It matches, with substitution* $\{(Y, a)\}$*, and gets subgoal s(a), which it satisfies. So it says "yes".*

Prolog's search for ways to satisfy its goal, is like the exploration of a maze. The process of backtracking is like the retracing of steps in a maze, after a dead end has been found.

Backtracking may be inhibited by writing

!

The instruction ! is called a cut. An example of how this is used would be:

```
p :- q1, !, q2
```

This statement would tell prolog that if it is trying to satisfy p and has got as far as q1, it can't backtrack in order to satisfy p. The only way to satisfy p is then to satisfy q2.

Continuing with the maze analogy, the cut is like a one way door in the maze of possibilities. Notice that this has absolutely no axiomatic analogy. There is no such thing as a cut, or anything like a cut, in a set of axioms. With the appearance of the cut, prolog blatantly diverges from its original idea. Nevertheless, the cut is interesting and useful.

Example 9.5 ——————-

```
a :- b,c.
c:- d,!,e.
c:- f.
a:- d,f.
b.
d.
f.
```

——————–

The goal

c

will fail, but the goal

a

will succeed.

We can understand the cut more fully by returning to semantic tableaux. As a goal, the cut always succeeds immediately. A branch with

!

at the leaf denied is just declared to be closed. As a predicate then

!

is like

True.

So a cut has no effect until a situation arises in which prolog tries to backtrack through the cut. The cut then comes into effect. Normally prolog would backtrack to the last goal, deleting the tree and the substitutions as it goes; this usually means reconsidering the branch immediately to the left of the current one. In the presence of the cut however, prolog backtracks *up* the tree to the parent goal, deletes the parent goal and the whole subtree of subgoals below it, and backtracks from there, as if no match for the parent goal could be found in the whole program.

Example 9.6 *Suppose, for example, we defined*

————–

```
brother(X,Y):- father(Z,X), father(Z,Y), male(X), X \=Y.

brother(X,Y) :- mother(Z,X), mother(Z,Y), male(X), X \=Y.
```

————–

Suppose you are trying to decide whether or not

brother(a,b)

and you already know that a and b have a common father. You need now to check that a is male and not the same as b. We know however that there is no use looking for another common father, or in looking for a common mother, since that will still leave us with the same subgoals. So we can add cuts which speed up the computation without changing its results.

——————————–

```
brother(X,Y):- father(Z,X), father(Z,Y), !,  male(X), X \=Y.

brother(X,Y) :- mother(Z,X), mother(Z,Y), !,  male(X), X \=Y.
```

————————–

This is what we might call an innocuous cut, since we believe it never can change the results of a computation. There are, however, other uses of the cut which are not innocuous in this way.

9.11.1 The Cut and fail combination

There is a predicate

fail

which is always false.

It is clear that with cut and fail we can, in effect, build mazes with traps in them. This can be used to express a sort of logical negation. For example:

```
p :- q, !, fail.

p.
```

This means that if q can be satisfied, p must fail. On the other hand, if q can not be satisfied, p is true. In this situation, p means that prolog can't prove q. We could think of p as

$not \vdash_{prolog} q$.

For example, prolog has a built in equality predicate, written

```
X == Y.
```

We can use this, together with the cut and fail combination to define inequality, as follows.

```
notequal(X,Y) :- X == Y, !, fail.

notequal(X,Y).
```

Another way of thinking of the above is that we take inequality to be true by default; it will be false only when we can prove equality.

To give another example, we could define femaleness as the negation of maleness, or vica verse

---

```
female(X) :- male(X), !, fail.

female(X).
```

Note that the whole effect is ruined if we change the order of the statements in a prolog program of this kind. So the meaning of such a prolog program depends on the order of the statements in it.

Another annoying feature of the cut is that it interferes with the capacity of a prolog program to generate a list of substitutions which satisfy a given condition. This is because the

;

depends on backtracking, which may be inhibited. For example, the above program will not generate a list of females. It can only be used to test whether or not a known individual is female.

To make the example even more annoying, consider

```
female(X) :- male(X), !, fail.
female(X).
female(louise).
male(john).
```

If this program is given query

```
female(X).
```

it will incorrectly say no.

9.12 Family Tree

Problem 9.1 *Finish the family tree example, which is started above, containing information about real or imaginary people over a number of generations. Define at least ten predicates, and include at least thirty facts. As well as a program,* draw a family tree for your community.

9.13 Lists, Member(X,Y), Append(X,Y,Z)

A list is a type of data structure. It is supposed to represent a set whose elements are arranged in some order.

We assume that there is such a thing as the empty list. This is denoted by [ ]. [ ] is a peculiar object but it is evidently necessary.

In Prolog all lists are finite.

In Prolog a list is written as a finite sequence of terms

$[t_1, t_2, ..., t_n]$

separated by commas and proceeded by a [and followed by a].

The terms $t_1, t_2, ..., t_n$ in list $[t_1, t_2, ..., t_n]$ are called the components of the list. The components of a list may also be lists.

Examples of Lists

[a,b,c] is a list with three components.

[[cat,X],Y] is a list with two components. The first component is also a list.

[[the, quick, brown, fox], [jumped], [over, the, lazy, dog]] is a list with three components.

The *head* of a list $[t_1, t_2, ..., t_n]$ is the first component, t_1. The *tail* is the rest of the list, $[t_2, t_3, ..., t_n]$.

We use

```
[X | Y]
```

to denote the list with head X and tail Y.

```
[X | Y]
```

is usually read as "X cons Y".

More examples

[a,b,c,d] has head a and tail [b,c,d].
[[a,a],[b,c],d] had head [a,a] and tail [[b,c],d].

```
[a | [b,c]] = [a,b,c]

[cat,cow,pig] = [cat | [cow,pig]] = [cat | [cow | [pig]]] =
[cat | [cow |  [pig | [  ] ]]]
```

Prolog represents lists internally as built up from the cons operation.

[p, imp, [q, imp, p]] is a list with three components. It is not the same list as [[q,imp,p], imp, p], i.e. order is important in lists. A list may also have repeated components. So [a,a,b] is not the same as [a,b].

Predicates involving lists are usually defined by recursion. For example,

---

```
samelength([ ], [ ]).

samelength([X | List1], [Y | List2])
:-samelength(List1,List2).
```

First the predicate is defined in the simplest case, when both lists are empty. Then if we are given two lists, neither of which is empty, we chop off their heads, and compare the lengths of the tails.

Most versions of prolog have built in predicates
member(X,Y)
and
append(X,Y,Z).

The predicate member(X,Y) is true if and only if Y is a list and X is a component of Y. If our prolog does not have member(X,Y) already defined, we can define it as follows:

```
member(X, [X | Z]).

member(X, [W | Z]) :- member(X,Z).
```

The idea here is: X is a member of Y if X is the first component of Y. Otherwise check to see if X is a member of the tail of Y.

The predicate append(X,Y,Z) is true if X, Y, and Z are all lists, and the list Z is obtained by appending list Y to list X.

We could give a recursive definition. The idea here is first to think of the simplest possible case: append(X,Y,Z) is true if X is empty and Y and Z are the same. We then express a more complicated case in terms of simpler cases. Suppose X is not empty, but the head of X is the same as the head of Z. We chop off these two heads and continue recursively.

```
append( [  ], Y, Y).

append([X | Y], Z, [X | W]) :- append (Y,Z,W).
```

Exercise. Try to see what prolog actually does when given a question such as:

append([cat],[dog,horse],[cat,dog,horse]).

9.14 Sorting

Prolog has built in predicates X = < Y and X < Y which have the usual meaning as applied to numbers. We will say a list of numbers is monotone non decreasing if every number in the list is less than or equal to the next number, if any, in the list. We can define this as follows:

```
mnd([X ]).

mnd([X | [Y | Z]]) :- X = <  Y, mnd([Y | Z]).
```

There is an algorithm called bubble sort, which takes a list, X, of numbers and rearranges it to get a monotone non decreasing list, Y. If the first list, X, is already monotone non decreasing, then bubble sort does nothing and Y is the same as X. On the other hand, if X is not already monotone non decreasing, bubble sort finds a pair of numbers in X which is out of

order, swaps them to get list Z and then applies bubble sort recursively to Z.

Problem 9.2 *Let bubble*(X, Y) *mean that* X *is a list of numbers and* Y *is a monotone non decreasing list which is obtained from* X *by bubble sort.*

Define bubble(X, Y) *in prolog without using the cut or negation. (Hint: If a list* X *is not in order, there is an adjacent pair of numbers in the list which is out of order.)*

If you give it a specific list X*, and leave* Y *as a variable, it should return a sorted list. For example*
bubble$([3, 5, 4], Y)$
should get the response $Y = [3, 4, 5]$.

This example shows how an algorithm, such as bubble sort, can be expressed in a declarative style. However it should be clear that a solution to the above problem actually relies on knowledge of how prolog behaves. So the distinction between procedural and imperative styles tends to break down when examined.

A solution to the above problem may look like a list of axioms for the predicate *bubble*(X, Y), but it is also a detailed specification of prolog's behaviour.

9.15 Equivalence relations; and how to find your way out of a labyrinth

A labyrinth is defined by a data structure with finitely many nodes and finitely many arrows. We will say that if an arrow goes from node X to node Y then it is possible to go either from X to Y or from Y to X in one step.

The labyrinth problem is to decide, given two nodes Z and W, whether or not there is a path from Z to W, and if there is such a path to find it.

A slightly harder problem we will call the labyrinth problem with monsters. In this case we age given a labyrinth, and a subset A of nodes which we wish to avoid. The problem is, given two nodes, to decide whether or not there is a path from one to the other which does not go through any node in the subset A.

Both these problems are extremely ancient. The main difficulty is that although the labyrinth is finite, it seems possible that in the search for a path we might go around in circles forever.

In the following, we will make a series of attempts to solve this problem. None of the proposed solutions are correct, except possibly the last one.

This section traces the evolution of a process of development of a solution to this problem. To fix ideas, please stop at this point and create an example labyrinth.

9.15.1 First Attempt

To begin with, we will represent the immediate connections by a list of facts. This is:

```
c(i,n).
c(t,e).
c(f,g).
c(w,e).
c(w,l).
```

etc.

A binary relation on a domain D is a set of ordered pairs of elements from D. We may represent a binary relation by a predicate of arity 2.

An *equivalence relation* on a domain D is a binary relation $r(X, Y)$, defined over D so that the statements

1) r is reflexive, i.e.:

$r(X, X)$

2) r is symmetric, i.e.:

$r(X, Y) \rightarrow r(Y, X)$

3) r is transitive, i.e.:

$(r(X, Y) \wedge r(Y, Z)) \rightarrow r(X, Z)$

are true over D. This means that these statements are true for all possible values of the variables X and Y in D.

There is a path from a node X to another node Y if and only if we can prove $r(X, Y)$ from the axioms for an equivalence relation, and

$(\forall X)(\forall Y)(c(X, Y) \rightarrow r(X, Y))$

and the above list of facts. Therefore we might try to solve the labyrinth problem with the following program.

.

---

facts as above

```
r(X,X).
```

```
r(X,Y) :- r(Y,X).
r(X,Z) :- r(X,Y), r(Y,Z).
r(X,Y) :- c(X,Y).
```

However this will certainly not work in prolog.

Prolog, given these axioms and a few facts, will just go into an infinite loop. For example, trying to decide r(a,b), prolog might choose the second of the two rules, and take as subgoal r(b,a); in trying to satisfy this, it might again use the second rule, and find r(a,b) as a sub-sub goal, etc.

The fact that prolog falls on its face when given one of the simplest axiom systems in mathematics shows that prolog does not fulfil its ideal of directly representing logic. On the other hand, the fact that prolog gets mixed up here can also be seen as a criticism of the usual way of doing mathematics. The standard definition does not really give us an effective definition of an equivalence relation.

Although the first attempt does not work, it seems that we have learned something.

9.15.2 Second attempt

We have a problem with saying directly that r(X,Y) is symmetric. It would not have been necessary to say that r(X,Y) was symmetric if all the arrows originally went in both directions. If that were true, then r(X,Y) would turn out to be symmetric, since it is defined symmetrically from c(X,Y).

So we could avoid having to say that r(X,Y) is symmetric just by putting all the arrows in twice. A better solution is just to say that r(X,Y) is to be true if either c(X,Y) is true or c(Y,X) is true.

So we get the following:

---

facts as above

```
r(X,X).
r(X,Z) :- r(X,Y), r(Y,Z).
r(X,Y) :- c(X,Y).
r(X,Y) :- c(Y,X).
```

Problem 9.3 *Why does this not work?*

9.15.3 Third attempt

It seems we should put the recursive step last. So we get

---

facts as above

```
r(X,Y) :- c(X,Y).
r(X,Y) :- c(Y,X).
r(X,X).
r(X,Z) :- r(X,Y), r(Y,Z).
```

Remarks. This is an improvement. It sometimes works and sometimes does not work.

Problem 9.4 *Give an example in which the above program does not work.*

9.15.4 Fourth attempt

It seems that we need to keep track of the path and avoid going in loops. We can use a variable

Route

to be the path of intermediate steps between the start and the finish.

The predicate

path(X,Y,Route)

will mean that Route is a path of intermediate steps which goes from X to Y.

Take out the recursive step in the program above, and replace it with.

```
path(X,Y,[ ]):- r(X,Y).
path(X,Y,[Z | Route ]):- r(X,Z), nonmember(Z,Route),path(Z,Y,Route).
nonmember(X,R) :- member(X,R), !, fail.
nonmember(X,R).
```

Problem 9.5 *What does the above program do?*

Problem 9.6 *What happens if we change the definition of nonmember(X,Y) as follows.*

———————

```
nonmember(X,[ ]).
nonmember(X, [Y | Z]) :- X \=Y, nonmember(X,Z).
```

———————

9.15.5 Fifth attempt

Define

path(*X*, *Z*, *Route*, *Avoid*)

to mean that *Route* is a path of intermediate steps which goes from *X* to *Z* but avoids going through any element in the list *Avoid*.

The definition is:

————————-

```
path(X, Z, [  ], Avoid) :- r(X,Z).
path(X,Z, [ A | B], Avoid) :- r(X,A),
                          nonmember(A, Avoid), path(A, Z, B, [A | Avoid]).
nonmember(X,Y) :-  member(X,Y) , !, fail.
nonmember(X,Y).
```

———————————

Then to find a route, for example, from a to z, we ask:

path(*a*, *z*, *Route*, [ ]).

Prolog will give us the route if there is one, and will inform us if there is none.

Problem 9.7 *Do you believe the above claim? What would it mean to say that the path finding program was correct? Try to state this carefully. Note that if we ask*

path(a,z,Route,A)

the program will fall on its face. How does it fall on its face? The area to be avoided must be set initially to a constant. Once you have decided how to state correctness, try either to prove or disprove it.

Would you be worried if your life depended on the correctness of a four line program written by an expert?

Problem 9.8 *Try moving the innocuous statement*

r(X,X).

around in the program. Try listing all possible paths from one point to another. On the basis of this, criticise the above program.

9.15.6 Sixth Attempt

Please write this yourself.

9.15.7 Testing and Proof

The series of examples above is supposed to convince you that there can be short programs which look right to begin with but do not work. Also, there is no way to prove that a program is correct by testing it. All a test can do is reveal that a program is wrong.

Also it is quite possible for an experienced and conscientious person to believe that an incorrect program is correct.

When you write a program and believe that it is correct, that belief should contain at least a vague idea for a proof of correctness. It seems that the only way to make any progress with this situation is to work toward making these vague ideas about correctness more explicit. It is usually quite difficult even to state clearly what we mean by correctness.

Problem 9.9 *If you think the program above, which you wrote, is correct, try to say why it is correct. If you are not able to say why it is correct, but you think it is correct, can you offer any justification to support your assertion? Can you at least say what correctness would be in this case?*

It has been known from the beginning of the history of science that anecdotal evidence, in situations in which the variables are not controlled, and in which the witnesses are not neutral, is not reliable at all.

9.16 Labyrinth Program

Problem 9.10 *Read the discussion of how to find your way out of a labyrinth in the section 9.15. On the basis of this, write a short program in Prolog which solves the problem of finding a path from one place to another in a labyrinth. Explain just what you think your program does. Give some evidence, in the form of tests and results, or, where possible, proofs, that your claims are true.*

9.17 Language Recognisers in Prolog

A language recogniser is a program or an algorithm which recognises whether or not a given expression is a grammatical member of some formal language.

Consider, for example, the *statement forms* involving p,q, and r, as defined by the grammar:

$sf \leadsto (sf \vee sf) \mid (sf \wedge sf) \mid (sf \rightarrow sf) \mid (\neg sf) \mid (sf \leftrightarrow sf) \mid p \mid q \mid r.$

If we represent strings of symbols in a language by lists, it is fairly easy to write language recognisers in prolog for context free languages, such as those defining statement forms.

We could represent a statement form such as $(p \rightarrow (q \vee r))$ by a list

$[p, imp, [q, or, r]]$

using "imp" for $\rightarrow$, and "or" for $\vee$. A language recogniser for the propositional statement forms involving p, q, and r can be obtained directly from the grammar.

```
sf([X, imp, Y]) :- sf(X), sf(Y).
sf([X, or, Y]) :- sf(X), sf(Y).
sf([X, and, Y]) :- sf(X), sf(Y).
sf([X, iff, Y]) :- sf(X), sf(Y).
sf([neg, X]) :- sf(X).
sf(X) :- member(X, [p,q,r]).
```

The number of *proposition name*s can be increased just by adding to the list in the last line. If we really need infinitely many *proposition name*s, we could include

```
sf(X) :- atom(X).
```

using the built in prolog predicate atom(X), which will return true for any sequence of characters beginning with a lower case letter.

The above prolog program will not only recognise grammatical statement forms in variables p, q, r, but it will also generate them. For example, if we ask a question

sf(X).

prolog will find an X which is a statement form. Presumably it will say

yes. X=p.

If we say

;

we will get another grammatical statement form, presumably q, and if we continue to request more, we will get $r, [p, imp, p]$, etc.

So this program appears both to generate and also to recognise statement forms.

Most context free languages can be dealt with in this way. (Some extra difficulty may occur when the grammar allows erasure of some grammatical symbols, i.e.

V := λ

or when the rewrite rules contain a loop.)

9.18 kitchen table propositional theorem prover

We can also implement the semantic tableau method, as so far described, in prolog.

In the following, A is supposed to be the list of statements asserted in a semantic tableau, and D is supposed to be the list of statements denied. The predicate contra(A,D) means that if the construction is started with initial tableau (A,D), all branches will end in contradiction.

```
contra(A,D) :- member(X,A), member (X,D).
contra(A,D) :- remove(A,[neg,X ],A1),contra(A1,[X | D]).
contra(A,D) :- remove(D, [neg,X ],D1), contra([X | A],D1).
contra(A,D) :- remove(D, [X,imp,Y ],D1),
             contra([X | A ], [Y | D1 ]).
contra(A,D) :- remove(A, [ X,imp,Y ],A1),
             contra(A1,[X | D ],
             contra( [Y | A1 ],D).
remove(List,Item,Shorterlist) :- append(A, [Item | B ],List),
append(A,B,Shorterlist).
```

The six lines of this program are supposed to correspond to the semantic tableau rules for negation and implication. You will need to add some more rules for other logical operators. To try to prove $(p \rightarrow p)$ you would query: contra([ ],[[p, imp ,p]]), and hope to get the answer yes. Since the semantic

tableau system was sound and complete, this program is also a sound and complete formal system for the propositional calculus.

As it stands this kitchen table theorem prover does a lot of unnecessary backtracking. You may wish to add some cuts to speed things up. Another idea is arranging the statements so that the branching is done as late as possible. You may also want to use some write statements, to see what is happening.

Note that although prolog is defined only using implication and conjunction in a very special form, we have obtained a complete theorem prover for the propositional calculus inside prolog. In a sense this is a triumph. We have encapsulated almost everything we know about the propositional calculus in a few lines of code.

9.19 Problems

Problem 9.11 *What is the relation between the Assertion and Denial lists which occur in the theorem prover above, and the branches which occurred in the semantic tableau algorithm?*

Problem 9.12 *Explain what prolog will do with*

```
contra([p],[[q,imp,p]]).
```

Problem 9.13 *Suppose the propositional theorem prover is given*

```
contra([],[[[p,imp,q],imp,[q,imp,r]]])
```

Following what prolog is doing, find the first subgoal

```
contra(A,D)
```

which fails. This means that there are only variables left in the live part of the branch, and that the same variable does not occur on both sides. Check that this gives a counterexample. What does prolog do next? Modify the code, using cuts, so that as soon as prolog finds a counterexample, it stops the computation and prints the counterexample.

To get something printed on the screen, you can use the built in
write(X)
which will write whatever term X is bound to. For example
write(A, 'are true')
will print the current list of Assertions followed by the text "are true".

Problem 9.14 *Write a prolog program which will check whether or not an expression represents a grammatically correct formula of L_N, the language of first order single sorted arithmetic. (Such a program is called a language recogniser.) Then translate the following statements into L_N, give them to your language recogniser, and see if you were correct, at least grammatically. If not, rewrite your translations, correcting the grammar. You can save yourself some time by telling your language recogniser to report errors as it finds them.*

a) X is prime.

b) X can be written as the sum of three squares in just one way.

c) some linear combination of X and Y can not be written as the sum of two squares unless Z is even.

d)X is congruent to Y modulo Z

e) There are infinitely many prime numbers.

f) There are infinitely many prime numbers P such that P+2 is also prime.

g)There are only finitely many prime numbers P so that P+2 is also prime.

Problem 9.15 *Implement the propositional theorem prover which was described above, and apply it to formulae of L_N. As a result you should be able to prove all tautologies in L_N, a tautology being a formula obtained by substitution from a statement form tautology.*

Problem 9.16 ** Invent some axioms, which seem correct to you, and add them to your theorem prover so that you can prove some statements which are true in the normal sense of the word about the natural numbers, but which are not tautologies. Try to make your theorem prover as good as you can, in a couple of hours of work.*

Try your theorem prover on f) and g) in the above list.

What does "true in the normal sense" mean in the above? Is everything which your theorem prover proves true in the normal sense? Do you think there are any statements which are true in the normal sense but which are not consequences of your theorem prover? Please think quite hard about this and explain your opinion, using common sense arguments. We will return to this later.

Problem 9.17 *Give a recursive definition of finite tree and translate this into prolog.*

Problem 9.18 *Write a prolog program which finds conjunctive normal form for a statement form, represented as a list. Use the rewriting method rather than the truth table method. $cnf(X, Y)$ should be true if X is a list*

corresponding to a statement form and Y is obtained by applying a sequence of rewrite rules, according to some priority, starting with X, and Y is in conjunctive normal form. This problem is solved, in ML, in the book by Paulson. Your program will probably not successfully test whether or not Y is a possible conjunctive normal form for X; however given X the program should compute Y which is one of the possible conjunctive normal forms for X.

This problem convinces me that pure declarativeness should not be taken as an ideal feature of programming. The predicate

Y is a possible conjunctive normal form for X

is much harder to describe than the conjunctive normal form algorithm.

In general it quite often happens that the set of solutions to some problem is quite complicated, but a particular solution can be found relatively easily by performing a certain sequence of operations.

There is a saying that there are two types of mathematicians, Greek and Babylonian, Greek ones being concerned with finding essential relationships in reality, and Babylonian ones being concerned with calculations. In my opinion this is a false distinction; and attempting to make sharp distinction between procedural and declarative programming is equally flawed.

9.20 Input and Output in Prolog

When prolog is running it will have at each instant a current input stream, which is initially from the keyboard, and a current output stream, which is initially the display.

We can switch streams as follows. Suppose X is instantiated to a filename. Then

tell(X)

switches the current output stream to X. The first time this goal occurs a new file with name X is created.

Similarly

see(X)

switches the current input stream to X.

Prolog has built in predicates

write(X)

and

read(X)

which write to and read from the current input and output streams.

9.20.1 write(X)

If a variable X is instantiated to a term, then write(X) will succeed and cause the term to be printed on the current output stream. So, for example, the items in a list, L, would be printed one after another by

expose(L)

with the following definition

```
--------------

expose([]).

expose([X | Y]) :- write(X), expose(Y).

---------------
```

In connection with this, there is another useful built in predicate

nl

which means "new line" and has the effect that all succeeding output is printed on the next line.

Another built in predicate is

tab(N)

which, when N is instantiated to a natural number, causes the cursor to move to the right by N spaces.

9.20.2 read(X)

The predicate

read(X)

will attempt to unify X with the next term that is input on the current input stream. The term must be followed by a full stop and a non printing character, such as a space or a return.

9.20.3 Debugging and tracing

You can get a full running report of what prolog is doing with the instruction

trace.

When this occurs as a goal the tracing facility is turned on. The result of this is that prolog will write every goal which it attempts to satisfy, including subgoals, failures and backtracking. The trace can be turned off with

notrace.

The problem with trace is that such a large amount of information is reported. You may only wish to see what prolog is doing in regard to certain goals. Suppose we are interested in a predicate

p(X,Y,Z)

with arity 3. We could ask for a report of attempts to satisfy goals involving this predicate with the instruction

spy(p/3)

and this can be turned off with

nospy(p/3).

In general the correct syntax for the spy argument is

predicate-name/arity.

So, for example if we wanted to follow what prolog was doing with append(X,Y,Z), we could use

spy(append/3).

9.20.4 Assertions and Retractions

Suppose that in the middle of running prolog we decide that we would like to add a new fact, say

p(a,b).

to our program. We could of course stop, add the new fact and rerun prolog. However it is also possible to add the fact immediately, as follows

asserta(p(a,b)).

In general if X is a clause, we can add X to the beginning of the program with

asserta(X).

A clause X can be added to the end of the program with

assertz(X).

A clause X can be removed from the program with

retract(X).

You can see what clauses are currently in the program with the query

clause(Head,Tail).

9.21 Arithmetic in Prolog

Prolog has built in all the usual binary predicates for inequality between integers: X < Y, X =<Y, X> Y, X>=Y.

Prolog also has built in function symbols for some arithmetic operations: X + Y, X - Y, X * Y, X / Y, X mod Y.

There is also an assignment command in prolog which sets a variable X to the value of an expression, E, which can be evaluated to give an integer or a real number. This is written

X is E.

Thus

X is (1+2)*3.

succeeds with X=9. Similarly

(2+3) < 4*5.

succeeds, first evaluating 2+3 and then evaluating 4*5 and then comparing.

Notice however that we need another version of equality, since, for example

4 = 2+2.

will fail, since the response of prolog to this will be to try to unify 4 and 2+2, which is not possible. What we need is a binary predicate which evaluates two arithmetic expressions and then checks to see if the results are the same. This is written

X =:=Y.

Example 9.7 *Suppose we want to define a predicate sum(X,Y) meaning that X is a natural number and Y is the sum of the natural numbers from 0 to X inclusive. This could be done as follows.*

——————————-

```
sum(X,0):- X = < 0.
sum(X,Y) :- Z is X-1, sum(Z,W),Y is W +X.
```

———————-

If we give this the query

sum(999,S).

we will get the answer S=999500. The program, given X, will find Y so that sum(X,Y) is true. However it will not work in reverse. So the program, given S=999500 will not find X so that sum(X,S). This is because when the program gets to line 2, it will try to evaluate X-1 and fail since X is uninstantiated.

Example 9.8 *The following program finds the greatest common divisor K of two integers I and J.*

——————————–

```
gcd(I,0,I).
gcd(I,J,K) :- R is I mod J, gcd(J,R,K).
```

————————

As above this only works when I and J are instantiated.

Prolog has a built in predicate
integer(X)
which succeeds if X is a whole number. If X is an expression, it is not evaluated inside this predicate. So integer(9) will succeed but integer (7+2) will fail. Also, if X is uninstantiated, integer(X) will fail. In particular the goal
integer(X).
will not result in prolog's choosing an integer.
Suppose we wanted a predicate which would list the natural numbers. We could write this as follows.

————————

```
natural(0).
natural(X):- X is Y+1, natural(Y).
```

————————

9.22 Typed Prolog

There are several logic programming languages which have attempted to improve on prolog. A particularly interesting one is the Gödel system. See [Hill and Lloyd,1994].

Prolog is a language with only one type. A variable in prolog stands for some unspecified object, and the objects are understood to belong to one big domain.

The Gödel language on the other hand, has a multiplicity of types. There are constructors, such as, for example, lists, which produce new types from old ones. So if we had a type, integers, we might also have lists of integers, lists of lists of integers, lists of lists of lists of integers, lists of integers and lists of integers, and so on. In fact, as long as there is one constructor and one type to begin with, an infinite number of distinct types are generated.

Another difference between the Gödel language and prolog is that the former has abandoned the cut, but uses a related construction called *commit.* A virtue of the commit operation is that programs written with it have a meaning which is not entirely destroyed by changing the order of the statements in the program.

Gödel allows definitions with quantifiers. This implies that a full, complete implementation of the language can not be specified.

Gödel also continues to follow the extreme form of the declarative ideal, which is that the procedural part of a program should be automatically and unobtrusively derived from the declarative part. My opinion is that this is a mistake.

9.23 Summary of this chapter

A prolog program is a compact way to represent information about a situation.

A predicate is a function whose codomain is the truth values. Prolog is concerned with representing logical relationships between predicates by rules and facts.

A fact is a single predicate expression.

A rule is an implication, written in the form

p :- q1, q2, ..., qn.

where p, q1, q2,..., qn are all predicate expressions. The rule

p :- q1,q2,..., qn.

means that p is true whenever all of q1,q2,..., qn are true.

From a procedural point of view, such a rule says that in order to find values which make p true, we should search for values which make q1, q2,..., qn true.

In order to create a prolog program about a situation, do the following.

- Decide what are the important predicates.
- Arrange the predicates in a hierarchy of complexity.
- Define more complex predicates in terms of simpler ones, using rules.
- Enter facts about predicates which are as simple as possible. Put the facts at the top of the program.

The declarative meaning of a prolog program is as a list of axioms. The procedural meaning is the computation prolog will do given a query. This procedure may be understood either in terms of semantic tableaux

(explained in section 9.1), or in terms of an algorithm for satisfaction of goals (explained in section 9.4).

Lists in prolog may be written as a sequence of terms, separated by commas and enclosed in square brackets.

```
[X | Y],
```

read as X cons Y is the list with head X and tail Y.

Predicates on lists are usually defined by recursion. An important example is append(X,Y,Z) which is defined by two rules

```
append( [], X, X).
append([X | Y ], Z, [ X | W ]) :- append(Y,Z,W).
```

Terms in prolog include constants and variables and the set of terms is closed under formation of lists and application of function symbols.

Two terms A and B are unified by a substitution α if $A_\alpha = B_\alpha$.

A substitution α is a most general unifier of two terms A and B if

- α unifies A and B.
- If β is any other unifier of A and B, there is a substitution δ so that $\alpha \circ \delta = \beta$

We have an algorithm to find a most general unifier of any two terms. When prolog matches a goal with a fact or the head of a rule, it does so by finding a most general unifier.

Using lists and recursion, we can write short programs in prolog with with very powerful computational behaviour. As an example of this, the propositional part of the semantic tableau algorithm is implemented in prolog. We can also easily write recognisers for context free languages in prolog.

On the other hand, there are many short programs in prolog which look right at first but are badly flawed. There are also programs which may be correct but for which correctness is hard to prove.

It should be pointed out that it is unscientific to claim that a program is somehow validated by reported correctness on some examples invented by the writer of the program.

Prolog programs using recursion and cuts should be viewed with especially sharp scepticism.

Chapter 10

Multisorted, higher order languages, and non classical logics

Up to this point we have been dealing with first order languages. In this situation there is only one type of variable, and in the semantics there is only one universal domain of objects. This is also the situation in prolog. We have seen that it is possible to express almost anything in this framework. However in computing languages, as in real life, we often wish to use languages in which variables are given different types to show that they are intended to range over different domains. For example, we might have a type for the integers and another type for the reals, and various functions which go from one domain to the other. In general, we will write $X : A$ to mean that variable X has type A. So for example $X : integer$ means that X is an integer. A multisorted language is a first order language in which the variables and function and predicate symbols have all been typed. We can also have a multisorted term language.

We may also wish to have variables for sets or functions, and to quantify over these variables. For example we can obtain higher order arithmetic by extending L_N by adding variables for sets of natural numbers, and a new predicate for set membership.

Bound variables can also be given types. So, for example,

$(\forall X : A)p(X)$

means that $p(X)$ is true for all X of type A. Similarly,

$(\exists X : A)p(X)$

means that there exists and X of type A so that $p(X)$. Many of the

ideas and methods we have considered for first order languages naturally extend to these typed languages. For example, CNF, DNF, prenex normal form, Skolem form, clausal form. The semantic tableau method also extends in a natural way. However, since the types usually have a fixed intended interpretation, the semantic tableau method is no longer complete.

10.1 Notation for types

We will assume certain basic types, such as integer, real, bool (for the truth values), nat(for the natural numbers). More complex types will be formed by type constructors, some of which are in the following list.

Suppose A and B are types.

1. $A \to B$ is the type of all functions from A to B.
2. $A \times B$ is the type of all ordered pairs (x, y) where $x : A$ and $y : B$.
3. $A \cup B$ is the disjoint union of A and B. This is obtained by first making two copies of A and B in such a way that they do not intersect, and then taking the union. So, for example, $integer \cup real$ is the disjoint union of the integers and the reals. The integer 5 in this context is not the same as the real number 5.0.
4. $P(A)$ is the type of all subsets of A.
5. A *list* is the type of lists of elements from A

So, for example, the factorial function has type $nat \to nat$. This would be written $fact : nat \to nat$. The logical operator of conjunction has type $bool \times bool \to bool$. Addition over the real numbers has type $real \times real \to real$. Reversal of a list of natural numbers has type $list\ nat \to list\ nat$.

10.2 Problems with types

Problem 10.1 *We will consider two types to be essentially the same if we can find a natural bijection from one to the other. Suppose A, B, C are types. Are the following pairs of types essentially the same? Why or why not? (If you say there is a bijection, you must define it!) If you say that they are not the same, you should give an example to show this.*

- $(A \times B) \times C$ *and* $A \times (B \times C)$*?*
- *What about* $A \to (B \to C)$ *and* $(A \times B) \to C$*?*

- *Is* $A \times (B \cup C)$ *essentially the same as* $A \times B \cup A \times C$*?*
- $(A \to (B \to C))$ *and* $((A \to B) \to C)$ *?*

10.2.1 Solutions to Types

In the following, maps are written on the left.

1. $((A \times B) \times C)$ is in 1-1 correspondence with $(A \times (B \times C))$, via the bijection f defined as follows.

   Pick x in $((A \times B) \times C$. x has the form $((a, b), c)$, where $a : A, b : B, c : C$. Define $f(x) = (a, (b, c))$. Note $f(x) : (A \times (B \times C))$. We can define function g in the other direction as follows. $g((a, (b, c))) = ((a, b), c)$. We have $f \circ g$ is the identity on $(A \times (B \times C))$, and $g \circ f$ is the identity on $((A \times B) \times C)$. (Check this by calculation, from the definitions.) Thus g is a two sided inverse for f and f is a bijection.

2. There is a bijection in this case. Define

   $f : (A \to (B \to C)) \to ((A \times B) \to C)$ as follows. Pick $x : A \to (B \to C)$. Define $f(x) = y : (A \times B) \to C$ by

   $y(a, b) = x(a)(b)$

   for $a : A, b : B$. Note that $x(a) : B \to C$, so this makes sense. As above, we define an inverse, which we will call g. Pick arbitrary $y : (A \times B) \to C$. Define $g(y) = x : A \to (B \to C)$ by

   $x(a)(b) = y(a, b)$

   for $a : A, b : B$. $f \circ g$ and $g \circ f$ are the identity on their respective domains. So f is a bijection. (You may feel that this is not a natural bijection. Note, however, that it is quite universal, i.e. independent of A, B, C.)

3. Same sets. The identity is a bijection.

4. Not the same cardinality in most cases. So there is no bijection in most cases.

10.3 Non classical logics

We may also wish to question our original assumptions about the truth values. Are there really only two truth values? Do we believe in the excluded middle principle which we have used? If we find that $\neg A$ is impossible,

does this imply that A is true? To see some of the developments and applications of this, the reader may refer to *Logic in Computer Science* by Huth and Ryan.

10.4 Proof Assistants

Even in the case of first order languages, we are not so far able successfully to automate the whole deductive process in a practical way. We have seen that prolog is quite limited since it always uses the same search strategy, and that the semantic tableaux method somewhat computationally difficult, since it leaves so many choices open. The multisorted higher order situation, which is what we usually want for applications, is certainly not going to be easier. A great deal of work has been done to develop systems in which proofs are constructed by interaction between a human being and a computer. The computer has at its disposal something like unification and the semantic tableau system. The human gives the initial proof goal. If the computer gets stuck, the human can suggest lemmas which are, hopefully, intermediate steps toward proving the goal. Example systems of this kind are HOL, Isabelle, and COQ. HOL means higher order logic. Isabelle is a proof assistant for higher order logic based on HOL. Isabelle also does first order logic and Zermelo Fraenkel set theory. It is written in the language ML. See

`www.cl.cam.ac.uk/Research/HVG/Isabelle`

for documentation and a tutorial.

Chapter 11

Semantics and Specification for Programs

Consider an imperative programming language, such as C. In such a language, a program is a list of instructions, which is given structure by branching and looping constructions. A typical line of such a program is an assignment, in which the value of a variable is reset. The program itself is just a piece of text. We would like to give a semantics to such programs. That is, we would like to be able to give such a program a precise interpretation, as some kind of mathematical object. We all have some such object in mind when we discuss what a program is meant to do. If we are able to clarify our ideas in this area, we may also in some cases be able to construct mathematical proofs of the correctness of computer programs or parts of computer programs. Even if we are not able to prove all the results we would like to prove, the existence of a formalisation has a wholesome effect on the whole subject of computing, since two practitioners who disagree about some fundamental issue can dispute in a common framework. This is also true for disputes between client and programmer. There is obviously a practical use for precise specification.

11.1 Programming Language Semantics

A programming language is, in a sense, similar to a first order language in that it has a signature, which consists of a list of function symbols and a list of predicate symbols. There are two important differences however.

1. First order languages only have variables of one type. In a first order

language, a variable stands for some unspecified object in the domain of the interpretation. We do not determine in advance what the domain of the interpretation is. It is true that some programming languages, such as prolog for example, or some versions of Lisp, are also untyped. However, most programming languages give types to variables, as discussed previously. For this reason we need to use multi sorted languages.

2. Formulae in first order languages are interpreted as statements. They return truth values. A set of formulae is interpreted as a list of conditions. On the other hand, a piece of software is naturally interpreted as an action of some sort. The software does something. The action of the software is the interpretation.

There are a number of differing approaches to the semantics for computer software. Two important examples are *operational semantics* and *denotational semantics*.

Operational semantics interpretes software as operations of some machine M. A compiler or interpreter from a programming language L for a machine M translates code in L into sequences of machine instructions for M, which are then realised as machine operations. The operational semantics for a language L is different for each target machine M.

Denotational semantics, on the other hand, gives each part of the signature of a programming language L a mathematical interpretation. Types are interpreted as mathematical sets. Function symbols are interpreted as functions. Predicate symbols are interpreted as predicates.

11.2 Denotational Semantics

We previously gave a semantics to formulae in a first order language. Given a formula of the language, we can find the function symbols, predicate symbols and constants which appear in the formula; we can call this the signature of the formula. We can analyse a piece of code or pseudocode in the same way. Consider, for example,

```
gcd(x:integer, y:integer):integer;
r:=remainder(x,y);
while (r not = 0)

  x:=y;
  y:=r;
```

```
  r:=remainder(x,y);

return y;
```

We find a constant, 0, the equality predicate and one function symbol, remainder(x,y). We can give types to all of these. So far so good. However, the program does not seem to be a formula. It is an action rather than a statement. Nevertheless, we can begin by considering an interpretation for the language with this signature. Thus we need a domain, D, and an interpretation for $=$ and for $remainder(x, y)$.

The code we are looking at has type declarations, for example $x : integer$. This says that we are not free to choose the domain for x. In general types in computing are sets which are associated with standard operations and predicates. So when you know the type of an object, you also know how to apply the common functions to it. In particular, we would regard equality and $remainder(x, y)$, i.e. the remainder when x is divided by y as fixed by convention in the integer type. Also, 0 is fixed by convention. So the whole interpretation is fixed.

We previously defined a valuation to be a function which gave values in the domain to some subset of of the variables. As before, we have valuations. In the semantics of programming languages, these are usually called *states*. An example of a state would be $\{x := 52, y := 13\}$.

If we start with any state, and apply a program, it will either not terminate or it will terminate with some other state. This is the basic idea of denotational semantics: computer code is interpreted as a partially defined transformation between states. We would say that the code denotes the transformation.

Once we are clear about the interpretation, the transformation is specified. We then have a clear semantics for programming languages.

11.3 Correctness and Completeness of software

Let α be a piece of software with inputs $X_1, ..., X_n$ and outputs $X'_1, ..., X'_n$.

Let Δ be a specification for α.

$- - - - - - - - - - - - - - - - - - \Delta - - - - - - - - - - - - - - - - - -$

Declarations

$X_1 : A_1, \ldots, X_n : A_n$

Predicates

$Precondition(X_1, \ldots, X_n)$

$Postcondition(X_1, \ldots X_n, X'_1, \ldots, X_n)$

We will say that α is correct with respect to Δ if whenever α is given $(X_1, \ldots, X_n)$ of the right types and $(X_1, \ldots, X_n)$ satisfies the precondition, and α terminates with $(X'_1, \ldots, X'_n)$, then $(X_1, \ldots, X_n, X'_1, \ldots, X'_n)$ satisfies the postcondition.

We will say that an input $(X_1, \ldots, X_n)$ is feasible for Δ if there exist output values $(X'_1, \ldots, X'_n)$ so that $(X_1, \ldots, X_n, X'_1, \ldots, X'_n)$ satisfies Δ.

We will say that α is complete with respect to Δ if α terminates for every input $(X_1, \ldots, X_n)$ which is feasible for Δ.

11.4 Formal Methods

In some situations, where software reliability is very important, we may try to prove, with some computer assistance, that a piece of code has certain properties. You now have all the ideas which you need to begin to understand how such a system could work. Look up Isabelle or COQ for examples. If software engineering is ever to be comparable in reliability to ordinary engineering, these techniques will need to be developed.

Chapter 12

References

Boolos, G. S., Jeffrey, R., *Computability and Logic*, Cambridge University Press, 1999

Bratko, I., Prolog programming for artificial intelligence, Addison Wesley

Huth, M. R. A., Ryan, M. D., *Logic in Computer Science*, Cambridge University Press, 2000

`See www.cs.bham.ac.uk/research/lcs/`

`Follow wwwtutor link for review questions about Huth and Ryan book, also useful for this course.`

See

`www.afm.sbu.ac.uk`

for many links concerning formal methods.

See

`turing.wins.uva.nl/~johan/Phil.298.html`

for Johan van Bentham's excellent notes on logic in games.

See

`mas.colognet.org/objectives.html`

for information about agents and logic.

Chomsky, N, *Aspects of the Theory of Syntax*. This argues for the connection between formal grammars and natural languages.

Slonneger, K., Kurz, B.L., *Formal Syntax and Semantics of Programming Languages*, Addison Wesley, 1995. This has a nice discussion of the lambda calculus, and of Scott domains. A beta reduction program is written in prolog.

Stoy, J.E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977. This is an advanced text.

Winskel, Glyn, The formal semantics of programming languages, The MIT press, Foundations of computing series, Cambridge Mass, 1993.

Wooldridge, Michael, An Introduction to MultiAgent Systems, John Wiley, 2002. Chapters 3 and 12 describe how some deductive ability can be built in to agents. The main idea is that selection of an action is a form of deduction.