Notes on Discrete Mathematics for Computer Scientists

James Caldwell

Department of Computer Science
University of Wyoming
Laramie, Wyoming 82071

Draft of
December 2005

[1]This material is based upon work supported by the National Science Foundation under Grant No. 9985239. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

Preface

Discrete mathematics is a required course in the undergraduate Computer Science curriculum. In a perhaps unsympathetic view, the standard presentations (and there are many)the material in the course is treated as a discrete collection of so many techniques that the students must master for further studies in Computer Science. Our philosophy, and the one embodied in this book is different. Of course the development of the students abilities to do logic and proofs, to know about naive set theory, relations, functions, graphs, inductively defined structures, definitions by recursion on inductively defined structures and elementary combinatorics is important. But we believe that rather than so many assorted topics and techniques to be learned, the course can flow continuously as a single narrative, each topic linked by a formal presentation building on previous topics. We believe that Discrete Mathematics is perhaps the most intellectually exciting and potentially one of the most interesting courses in the computer science curriculum. Rather than a simply viewing the course as a necessary tool for further, and perhaps more interesting developments to come later, we believe it is the place in the curriculum that an appreciation of the deep ideas of computer science can be presented; the relation between syntax and semantics, how it is that unbounded structures can be defined finitely and how to reason about those structure and how to calculate with them.

Most texts, following perhaps standard mathematical practice, attempt to minimize the formalism, assuming that a students intuition will guide them through to the end, often avoiding proofs in favor of examples [2] Mathematical intuition is an entirely variable personal attribute, and even individuals with significant talents can be misguided by intuition. This is shown over and over in the history of mathematics; the history of the characterization of infinity is a prime example, but many others exist like the Tarski-Banach paradox [**?**]. We do not argue that intuition should be banished from teaching mathematics but instead that the discrete mathematics course is a place in the curriculum to cultivate the idea, useful in higher mathematics and in computer science, that formalism is trustworthy and can be used to verify intuition.

Indeed, we believe, contrary to the common conception, that rather than making the material more opaque, a formal presentation gives the students a way to understand the material in a deeper and more satisfying way. The fact that formal objects can be easily represented in ways that they can be consumed by computers lends a concreteness to the ideas presented in the course. The fact that formal proofs can be sometimes be found by a machine and can aways be checked by a machine give an absolute criteria for what counts as a proof; in our experience, this unambiguous nature of of formal proofs is a comfort to students trying to decide if they've achieved a proof or not. Once the formal criteria for proof has been assimilated, it is entirely appropriate to relax the rigid idea of a proof as a machine checkable structure and to allow more simply

[2] As an example we cite the pigeonhole principle which is not proved in any discrete mathematics text we know of but which is motivated by example. The proof is elementary once the ideas of injection, surjection and one-to-one mappings have been presented.

rigorous but informal proofs to be presented.

The formal approach to the presentation of material has, we believe, a number of significant advantages, especially for Computer Science students, but also, for more traditional math students who might find their way into the course.

In mathematics departments proofs are typically learned by students through a process of osmosis. Asked to give a proof, students hand in what they might believe is a proof and the professor, as oracle, will either accept it or reject it. If he rejects it he may point out that a particular part of the purported proof is too vague, or that all cases have not been considered or he might identify some other flaw. In any case, the criteria for what counts as a proof is a vague one, students are left in doubt as to what a proof actually is and what might count as one. We are convinced that this process, of learning by example only works for students who have some innate ability to understand the distinctions being made by repeated exposure to examples. But these distinctions are rarely made explicit. Indeed, the successful student must essentially reconstruct for himself a model of proof that has already been completely developed in explicit detail by logicians starting with Frege. Most mathematicians would agree that, in principle, proofs can be formalized – of course this was Hilbert's attempt to answer the paradoxes. But mathematicians, unlike logicians, do not teach proofs in this way because that is not the way they do them in practice.

For computer scientists and software engineers, formalism is their daily bread. Logic is the mathematical basis of computation as calculus and differential equations are the mathematical basis of engineering physical systems. Programs are formal syntactic objects. Computation, whether based on an abstract model; like a Turing machine, the lambda calculus, or register transfer machines; or based on a more realistic model like the Java virtual machine; is a formal manipulation governed by formal rules. We believe that a formal presentation of discrete mathematics is the best (and perhaps earliest) point in the curriculum to make the distinction between syntax and semantics explicit and to make proofs something that all students can learn to do, not only those students who have some natural talent for making such arguments. Also, recursion is the computational dual of induction and students unable to learn how to do inductive proofs are unlikely to be able to consistently and successfully write recursive procedures or to understand the reasons recursion works.

Of course this approach is not new. Dijkstra, Hoare and a hoard of other computer scientists have argued for such an approach. Dijkstra's famous comment "To suggest that computer science is the study of computers is like suggesting that astronomy is the study of the telescope." captures the idea. Following Dijkstra's perhaps radical views, Robert Boyer at the University of Texas at Austin has proposed a Computer Science curriculum entirely based on formalism and relegates the actual use of computers only to later course in the curriculum. We may never know the results of such a radical approach to computer science education because no institution may ever be in a position to carry out such an experiment, but certainly, within the context of the discrete mathematics course such an approach is not radical. And yet, a survey of the most popular texts reveals that it is rarely approached in his way.

The text which is most closely embodies the approach taken here is Gries and Schneider's [14]. Gries and Schneider developed an equational approach to logic based principally on the connectives for bi-equivalence and exclusive-or. Our approach differs in that we use a standard form of proofs based on Gentzen's sequent calculus [11].

As computer scientists we care about the reasons we can make a claim about a computational artifact; among these artifacts we include: algorithms, data-structures, programs, systems, and models of systems. Proofs are the means to this end. Proofs tell us *why* something is true rather than just telling us whether it is true or not. The ability to make such arguments is crucial to the endeavor of the computer scientist and the software engineer. To specify how a computational artifact is supposed to behave requires logic. To be able to prove that a computational artifact has some property, for other than the most trivial properties, requires a proof. As computer scientists and software engineers we must take responsibility for the things we build, to do so requires more than that we simply build them, we must be able to make arguments for their correctness.

Proofs have another advantage; failed proofs of false conjectures usually reveal something about why the conjecture is not true. Proofs in computer science are often not deep, but can be extraordinarily detailed. In this course we learn most of the mathematical structures and proof techniques required to verify properties about computational artifacts. Students who practice with the techniques presented here and will find applications in most aspects of designing, building and testing computational artifacts.

Prerequisites for this course typically include a semester of calculus and at least two semesters of programming. From programming, we assume students know at least one programming language and may have been exposed to two, though we do not assume they are experts. There is no programming in the course as taught at Wyoming, but exposure to these ideas is important. Based on their exposure to a programming language, we assume students have had some experience implementing algorithms, and preferably have had some exposure to a inductively defined data-type, like lists or trees, although that experience may not have emphasized the inductive nature of those types, *e.g.* they may have been more focused on the mechanics of manipulating pointers if their language is C++. Mathematically, we assume students are already familiar with notations for set membership ($x \in A$), explicit enumeration of sets (*e.g.* finite enumerations of the form $\{a, b, c, d\}$ and infinite enumerations of the form $\{0, 2, 4, \cdots\}$). We also assume that a student has seen notation for functions ($f : A \to B$) specifying that f is a function from A to B. Of course all the mathematical prerequisites just mentioned are represented here in some detail in the appropriate chapters.

These notes do not attempt to systematically build the mathematical structures and methods studied here from some absolute or minimal foundation. We certainly attempt to explain things the first time they appear, but often, complex ideas, like the inductive structure of syntax for example, are introduced before a full and precise account can be given. We believe that repeatedly see-

ing the same methods and constructs a number of times throughout the course and in a number of different guises is the best path to the students learning.

Chapter 1

Syntax and Semantics

In this section we give a brief and relatively informal view of syntax and semantics. We describe, without delving to deeply into the details, how to specify abstract syntax using grammars, we will see a mathematical justification of these ideas in the chapter presenting inductively defined sets. We also present simple examples of semantics and recursive functions defined on the abstract syntax in this chapter. A detailed account of the material presented in this chapter would draw heavily on material presented later in these lectures; indeed, we are starting the lectures with an application of discrete mathematics as used in computer science; the interpretation of inductively defined syntax by semantic functions.

1.1 Introduction

Syntax has to do with form and *semantics* has to do with meaning. Syntax is described by specifying a set of structured terms while semantics associates a meaning to the structured terms. In and of itself syntax does not have meaning, only structure. Only after a semantic interpretation has been specified for the syntax do the structured terms acquire meaning. Of course, good syntax suggests the intended meaning in a way that allows us *see though it* to the intended meaning but it is an essential aspect of the formal approach, based on the separation of syntax and semantics, that we do not attach these meanings until they have been specified.

The syntax/semantics distinction is fundamental in Computer Science and goes back to the very beginning of the field. Abstractly, computation is the manipulation of formal (syntactic) representation of objects [1]

For example, when compiling a program in written some language (say C++) the compiler first checks the syntax to verify that the program is in the language.

[1]The abstract characterization of computation as the manipulation of syntax, was first given by logicians in the 1930's who were the first to try to describe what we mean by the word "algorithm".

If not, a syntax error is indicated. However, a program that is accepted by the compiler is not necessarily correct, to tell if the program is correct we must consider the semantics of the language. One reasonable semantics for programs is its execution. Just because a program is in the language (*i.e* the compiler produces an executable output) does not guarantee the correctness of the program (*i.e.* that the program *means* the right thing) with regard to the intended computation.

1.2 Formal Languages

Mathematically, a *formal language* is a (finite or infinite) set of structured terms (think of them as trees.) These terms are of finite size and are defined over a basis of lexical primitives or an alphabet. An alphabet is a finite collection of symbols and each term itself is a finite structured collection of these symbols, although there may an infinite number of terms.

Finite languages can (in principle) be specified by enumerating the terms in the language; infinite languages are usually characterized by specifying a finite set of rules for constructing the set of term structures included in the language. Such a set of rules is called a *grammar*. In 1959 Noam Chomsky, a linguist at MIT, first characterized the complexity of formal languages by characterizing the structure of the grammars used to specify them [3]. Chomsky's characterization of formal languages lead to huge progress in the early development of the theory of programming languages and in their implementations, especially in parser and compiler development. Essentially, his idea for classifying formal languages was to classify them according to the computational complexity required to identify if a term was in the language.

The relationship between syntax and semantics goes back to the 1930's at least. The study of formal languages has an extensive literature of it's own [**?**, 8, 21, 29, **?**]. Similarly, the study of mathematical semantics of programming languages is a rich area in its own right [31, 32, 27, 30, 33, 16, 17, 15, 36, 1, 24]

1.3 Syntax

We can finitely describe abstract syntax in a number of ways. A common way is to give a formal grammar for the terms of the language. We give an abstract description of a grammar over an alphabet and then, in later sections we provide examples to make the ideas more concrete.

A grammar over an alphabet (say Σ) is of the form

$$class_T ::= C_1 \mid C_2 \mid \cdots \mid C_n$$

where

T: is a set which, if empty is omitted from the specification, and
class: is the name of the syntactic class being defined, and
C_i: are constructors $1 \leq i \leq n,\ n > 0$

The symbol ::= separates the name of the syntactic class being defined from the collection of rules that define it. Note that the vertical bar "|" is read as "or" and it separates the rules (or productions) used to construct the terms of *class*. The order of the rules does not matter, but in more complex cases it is conventional to write the simpler cases first. Sometimes it is convenient to parametrize the class being defined by some set. We show an example of this below were we simultaneously define lists over some set T all at once, rather than making separate syntactic definitions for each kind of list.

Traditionally, the constructors are also sometimes called *rules* or *productions*. The constructors are either constants from the alphabet, are elements from some collection of sets, or are functions of elements from the alphabet, the sets, and possibly of previously constructed elements of the syntactic class; the constructor functions return new elements of the syntactic class. At least one constructor must not include arguments consisting of previously constructed elements of the class being defined; this insures that the language defined by the grammar is non-empty.

1.3.1 Concrete vs. Abstract Syntax

A text is a linear sequence of symbols which, on the printed page, we read from left to right[2] and top to bottom. We can specify syntax *concretely* so that it can be read unambiguously as linear sequence of symbols, or *abstractly* which simply specifies the structure of terms without telling us how they must appear to be read as text. We use parentheses to indicate the order of application of the constructors in a grammar when writing abstract syntax as linear text.

Concrete syntax completely specifies the language in such a way that there is no ambiguity in reading terms in the language as text, *i.e.* as a linear sequence of symbols read from left to right. For example, does the ambiguous arithmetic statement $(a * b + c)$ mean $(a * (b + c))$ or $((a * b) + c)$? In informal usage we might stipulate that multiplication "binds tighter than addition" so the common interpretation would be he second form; however, in specifying a concrete syntax for arithmetic we would typically completely parenthesize statements (*e.g.* we would write $((a * b) + c)$ or $(a * (b + c))$) and perhaps specify conventions that allow us to drop parentheses to make reading the statement easier.

In *abstract syntax* the productions of the grammar are considered to be constructors for structured terms having tree-like structure. We do not include parentheses in the specification of the language, there is no ambiguity because we are specifying trees which explicitly show the structure of the term without the use of parentheses. When we write abstract syntax as text, we add parentheses as needed to indicate the structure of the term, *e.g.* in the example about we

[2]Of course the fact that we read and write from left to right is only an arbitrary convention, Hebrew and Egyptian hieroglyphics are read from right to left. But even the notion of left and right are simply conventions, Herodotus [19] tells us in his book *The History* (written about 440 B.C.) that the ancient Egyptians wrote moving from right to left but he reports "they *say* they are are moving [when writing] to right", *i.e.* what we (in agreement with the ancient Greeks) call left the ancient Egyptians called right and vice versa.

Figure 1.1: Abstract Syntax Trees

would write $a * (b + c)$ or $(a * b) + c$ depending on which term we intend.

Abstract syntax can be displayed in tree form. For example, the formula $a * (b + c)$ is displayed by the abstract syntax tree on the left in Fig. 1.1 and the formula $(a * b) + c$ is displayed by the tree on the right of Fig. 1.1. Notice that the ambiguity disappears when displayed in tree form since the principle constructor labels the top of the tree. The immediate subterms are at the next level and so on. For arithmetic formulas, you can think of the topmost (or principle) operator as the last one you would evaluate.

1.3.2 Some examples of Syntax

We give some examples of grammars to describe the syntax of the Booleans, the natural numbers, a simple language of Boolean expressions which includes the Booleans and an *if-then-else* construct, and describe a grammar for constructing lists where the elements are selected from some specified set.

Syntax of $\mathbb{B}$

The *Booleans*[3] consist of two elements. We denote the elements by the alphabet consisting of the symbols **T** and **F**. Although this is enough, *i.e.* it is enough to say that a Boolean is either the symbol **T** or is the symbol **F**, we can define the Booleans (denoted $\mathbb{B}$) by the following grammar:

$$\mathbb{B} ::= \mathbf{T} \mid \mathbf{F}$$

Read the definition as follows:

A Boolean is either the symbol **T** *or the symbol* **F**.

The syntax of these terms is trivial, they have no more structure than the individual symbols of the alphabet do. The syntax trees are simply individual nodes labeled either **T** or **F**. There are no other ASTs for the class $\mathbb{B}$.

Syntax of $\mathbb{N}$

The syntax of the natural numbers (denoted by the symbol $\mathbb{N}$) can be defined as follows:

[3] "Boolean" is eponymous for George Boole, the English mathematician who first formulated symbolic logic in mathematical form.

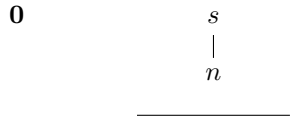


Figure 1.2: Abstract Syntax Trees for $\mathbb{N}$

Definition 1.1.

$$\mathbb{N} ::= \mathbf{0} \mid \mathtt{s}\, n$$

where the alphabet consists of the symbols $\{\mathbf{0}, s\}$ and n is a variable denoting some previously constructed element of the set $\mathbb{N}$.

The definition is read as follows:

> *A natural number is either: the constant symbol* $\mathbf{0}$ *or is of the form* $s\,n$ *where* n *is a previously constructed natural number.*

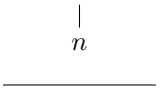
Implicitly, we also stipulate that *nothing else* is in $\mathbb{N}$, *i.e.* the only elements of $\mathbb{N}$ are those terms which can be constructed by the rules of the grammar.

Thus, $\mathbb{N} = \{\mathbf{0}, s\mathbf{0}, ss\mathbf{0}, sss\mathbf{0}, \cdots\}$ are all elements of $\mathbb{N}$. Note that the variable "n" used in the definition of the rules never occurs in an element of $\mathbb{N}$, it is simply a place-holder for an term of type $\mathbb{N}$, *i.e.* it must be replaced by some term from the set $\{\mathbf{0}, s\mathbf{0}, ss\mathbf{0}, \cdots\}$. Such place-holders are called *meta-variables* and are required if the language has inductive structure, *i.e.* if we define the elements of the language using previously constructed elements of the language.

Although the grammar for $\mathbb{N}$ contains only two rules, the language it describes is far more complex than the language of $\mathbb{B}$ (which also consists of two rules.) There are an infinite number of syntactically well-formed terms in the language of $\mathbb{N}$. To do so it relies on n being a previously defined element of $\mathbb{N}$; thus $\mathbb{N}$ is an inductively defined structure.

Abstract Syntax Trees for $\mathbb{N}$

The trees are of one of two forms shown in Fig. 1.2. The subtree for a previously constructed $\mathbb{N}$ labeled n is displayed by the figure:



The triangular shape below the n is intended to suggest that n itself is an abstract syntax tree whose exact shape is unknown. Of course, any *actual* abstract syntax tree would not contain any of these triangular forms. For example, the abstract syntax trees for the terms $s\mathbf{0}$ and $sss\mathbf{0}$ are displayed in Fig. 1.3.

Figure 1.3: Syntax trees for terms $s\mathbf{0}$ and $sss\mathbf{0}$

Syntax of a simple computational language

We define a simple language PLB (Programming Language Booleans) for computing with Booleans. The alphabet of the language includes the symbols {`if` , `then` , `else` , `fi` }, *i.e.* the alphabet includes a collection of keywords suitable for constructing *if-then-else* statements. We use the Booleans as the basis.

$$PLB ::= b \mid \texttt{if}\ p\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{fi}$$

where

$b \in \mathbb{B}$: is a Boolean, and
p, p_1, p_2: are previously constructed terms of PLB.

Terms of the language include:

{**T**, **F**, `if` **T** `then` **T** `else` **T** `fi`, `if` **T** `then` **T** `else` **F** `fi`,
`if` **T** `then` **F** `else` **T** `fi`, `if` **T** `then` **F** `else` **F** `fi`,
`if` **F** `then` **T** `else` **T** `fi`, `if` **F** `then` **T** `else` **F** `fi`,
...
`if if` **T** `then` **T** `else` **T** `fi then` **T** `else` **T** `fi`,
`if if` **T** `then` **T** `else` **T** `fi then` **T** `else` **F** `fi`,
...
`if if` **T** `then` **T** `else` **T** `fi then if` **T** `then` **T** `else` **T** `fi else` **T** `fi`,
...
}

Thus, the language PLB includes the Boolean values $\{\mathbf{T}, \mathbf{F}\}$ and allows arbitrarily nested *if-then-else* statements.

Lists

We can define lists containing elements from some set T by two rules. The alphabet of lists is $\{[\,], ::\}$ where "[]" is a constant symbol called "nil" which denotes the empty list and "::" is a symbol denoting the constructor that adds an element of the set T to a previously constructed list. This constructor is, for historical reasons, called "cons". Note that although "[]" and "::" both consist

of sequences of two symbols, we consider them to be atomic symbols for the purposes of this syntax.

This is the first definition where the use of the parameter (in this case T) has been used.

$$List_T ::= [\,] \mid a :: L$$

where

T: is a set,
$[\,]$: is a constant symbol denoting the *empty list*, which is called "nil",
$a \in L$: is an element of the set T, and
L: is a previously constructed $List_T$.

A list of the form $a::L$ is called a *cons*. The element a from T in $a::L$ is called the *head* and the list L in the cons $a::L$ is called the *tail*.

As an example, let $A = \{a, b\}$, then the set of terms in the class $List_A$ is the following:

$$\{[\,], a::[\,], b::[\,], a::a::[\,], a::b::[\,], b::a::[\,], b::b::[\,], a::a::a::[\,], a::a::b::[\,], \cdots\}$$

We call terms in the class $List_T$ *lists*. The set of all lists in class $List_A$ is infinite, but each list is finite because lists must always end with the symbol $[\,]$. Note that we assume $a::b::[\,]$ means $a::(b::[\,])$ and not $(a::b)::[\,])$, to express this we say *cons associates to the right.* The second form violates the rule for cons because $a::b$ is not well-formed since b is an element of A, it is not a previously constructed $List_A$. To make reading lists easier we simply separate the consed elements with commas and enclose them in square brackets "[" and "]", thus, we write $a::[\,]$ as $[a]$ and write $a::b::[\,]$ as $[a, b]$. Using this notation we can rewrite the set of lists in the class $List_A$ more succinctly as follows:

$$\{[\,], [a], [b], [a, a], [a, b], [b, a], [b, b], [a, a, a], [a, a, b], \cdots\}$$

Note that the set T need not be finite, for example, the class of $List_{\mathbb{N}}$ is perfectly sensible, in this case, there are an infinite number of lists containing only one element *e.g.*

$$\{[0], [1], [2], [3] \cdots\}$$

Abstract Syntax Trees for Lists

Note that the pretty linear notation for trees in only intended to make them more readable, the syntactic structure underlying the list $[a, b, a]$ is displayed by the following abstract syntax tree:

We give a semantics for the languages PLB, $\mathbb{B}$ and $\mathbb{N}$ in the next section and define functions by recursion on the syntax of lists.

1.4 Semantics

Semantics associates meaning to syntax. Before a semantics is introduced, an element in a syntactic class can only be seen as a meaningless structured term, or if expressed linearly as text, it is simply a meaningless sequence of symbols. Since semantics are intended to present the meanings of the syntax, they are taken from some mathematical domain which is already assumed to be understood or is, by some measure, simpler. In the case of a program, the meaning might be the sequence of states an abstract machine goes through in the evaluation of the program on some input (in this case, meanings would consist of pairs of input values and sequences of states); or perhaps the meaning is described simply as the input/output behavior of the program (in this case the meaning would consist of pairs of input values and output values.) In either case, the meaning is described in terms of (well understood) mathematical structures . Semantics establish the relationship between the syntax and its interpretation as a mathematical structure.

Semantics of $\mathbb{B}$

Suppose that we intend the meanings of $\mathbb{B}$ to be among the set $\{0, 1\}$. Then, functions assigning the values $\mathbf{T}$ and $\mathbf{F}$ to elements of $\{0, 1\}$ count as a semantics. Following the tradition of denotational semantics, if $b \in \mathbb{B}$ we write $[\![b]\!]$ to denote the meaning of b. Using this notation one semantics would be:

$$[\![\mathbf{T}]\!] = 0$$
$$[\![\mathbf{F}]\!] = 1$$

Thus, the meaning of $\mathbf{T}$ is 0 and the meaning of $\mathbf{F}$ is 1. This interpretation might not be the one you expected (*i.e.* you may think of 1 as $\mathbf{T}$ and 0 as $\mathbf{F}$) but, an essential point of formal semantics is that the meanings of symbols or terms need not be the one you impose through convention or force of habit. Things mean whatever the semantics say they do [4]. Before the semantics has been given, it is a mistake to interpret syntax as anything more than a complex of meaningless symbols.

As another semantics for Booleans we might take the domain of meaning to be *sets* of integers [5]. We will interpret $\mathbf{F}$ to be the set containing the single element 0 and $\mathbf{T}$ to be the set of all non-zero integers.

$$[\![\mathbf{T}]\!] = \{k \in \mathbb{Z} \mid k \neq 0\}$$
$$[\![\mathbf{F}]\!] = \{0\}$$

This semantics can be used to model the interpretation of integers as Booleans in the C++ programming language where any non-zero number is interpreted

[4] Perhaps interestingly, in the logic of CMOS circuit technology, this seemingly backwards semantic interpretation is the one used.

[5] We denote the set of integers $\{\cdots, -1, 0, 1, 2, \cdots\}$ by the symbol $\mathbb{Z}$. This comes from German *Zahlen* which means number.

as **T** and 0 is interpreted as **F** as follows. If i is an integer and b is a Boolean then:

$$(\mathbf{bool})\, i = b \quad \text{iff} \quad i \in [\![b]\!]$$

This says: the integer i, interpreted as a Boolean[6], is equal to the Boolean b if and only if i is in the set of meanings of b; *e.g.* since $[\![\mathbf{T}]\!] = \{k \in \mathbb{Z} \mid k \neq 0\}$ we know $5 \in [\![\mathbf{T}]\!]$ therefore we can conclude that $(\mathbf{bool})5 = \mathbf{T}$.

Semantics of $\mathbb{N}$

We will describe the meaning of terms in $\mathbb{N}$ by mapping them onto non-negative integers. This presumes we already have the integers as an understood mathematical domain[7].

Our idea is to map the term $\mathbf{0} \in \mathbb{N}$ to the actual number $0 \in \mathbb{Z}$, and to map terms having k occurrences of s to the integer k. To do this we define the semantic equations recursively on the structure of the term. This is the standard form of definition for semantic equations over a grammar having inductive structure.

$$\begin{aligned} &[\![\mathbf{0}]\!] = 0 \\ &[\![sn]\!] = [\![n]\!] + 1 \qquad\qquad \text{where } n \in \mathbb{N} \end{aligned}$$

The equations say that the meaning of the term **0** is just 0 and if the term has the form sn (for some $n \in \mathbb{N}$) the meaning is the meaning of n plus one. Note that there are many cases in the recursive definition as there are in the grammar, one case for each possible way of constructing a term in $\mathbb{N}$. This will always be the case for every recursive definition given on the structure of a term.

Under these semantics we calculate the meaning of a few terms to show how the equations work.

$$\begin{aligned} &[\![s\mathbf{0}]\!] \\ &= [\![\mathbf{0}]\!] + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned} \qquad\qquad \begin{aligned} &[\![sssss\mathbf{0}]\!] \\ &= [\![ssss\mathbf{0}]\!] + 1 \\ &= ([\![sss\mathbf{0}]\!] + 1) + 1 \\ &= (([\![ss\mathbf{0}]\!] + 1) + 1) + 1 \\ &= ((([\![s\mathbf{0}]\!] + 1) + 1) + 1) + 1 \\ &= (((([\![\mathbf{0}]\!] + 1) + 1) + 1) + 1) + 1 \\ &= ((((0 + 1) + 1) + 1) + 1) + 1 \\ &= (((1 + 1) + 1) + 1) + 1 \\ &= (((2 + 1) + 1) + 1 \\ &= ((3 + 1) + 1 \\ &= 4 + 1 \\ &= 5 \end{aligned}$$

Thus, under these semantics, $[\![s\mathbf{0}]\!] = 1$ and $[\![sssss\mathbf{0}]\!] = 5$.

[6] A cast in C++ is specified by putting the type to cast to in parentheses before the term to be cast.

[7] Because the integers are usually constructed from the natural numbers this may seem to be putting the cart before the horse, so to speak, but it provides a good example here.

Semantics of PLB

The intended semantics for the language PLB to reflect evaluation of Boolean expressions where *if-then-else* has the normal interpretation. Thus our semantics will map expressions of PLB to values in $\mathbb{B}$. Recall the syntax of PLB:

$$PLB ::= b \mid \texttt{if } p \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ fi}$$

As always, the semantics will include one equation for each production in the grammar. Informally, if a PLB term is already a Boolean, the semantic function does nothing. For other, more complex, terms we explicitly specify the values when the conditional argument is a Boolean, and if it is not we repeatedly reduce it until it is grounded as a Boolean value. The equation for *if-then-else* is given by case analysis (on the conditional argument).

$$[\![b]\!] = b \tag{1}$$

$$[\![\texttt{if } p \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ fi}]\!] = \begin{cases} [\![p_1]\!] & (p = \mathbf{T}) \\ [\![p_2]\!] & (p = \mathbf{F}) \\ [\![\texttt{if } q \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ fi}]\!] & (p \notin \mathbb{B}, q = [\![p]\!]) \end{cases} \tag{2}$$

We have numbered the semantic equations so we can refer to them in the example derivations below; we have annotated each step in the derivation with: the equation used, the bindings of the variables used to match the equation, and, in the case of justifications based on equation (2) the case used to match the case of the equation. Note that the equations are applied from top down, *i.e.* we apply the case $p \notin \mathbb{B}$ only after considering the possibility that $p = \mathbf{T}$ and $p = \mathbf{F}$.

Here are some equational derivations that show how the equations can be used.

$[\![\texttt{if } \mathbf{T} \texttt{ then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi}]\!]$
$\langle\langle$Equation : 2 $p = \mathbf{T}, p_1 = \mathbf{F}, p_2 = \mathbf{T}$ Case : $p = \mathbf{T}\rangle\rangle$
$= [\![\mathbf{F}]\!]$
$\langle\langle$Equation : 1 $b = \mathbf{F}\rangle\rangle$
$= \mathbf{F}$

$[\![\texttt{if } \mathbf{F} \texttt{ then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi}]\!]$
$\langle\langle$Equation : 2 $p = \mathbf{F}, p_1 = \mathbf{F}, p_2 = \mathbf{T}$ Case : $p = \mathbf{F}\rangle\rangle$
$= [\![\mathbf{T}]\!]$
$\langle\langle$Equation : 1 $b = \mathbf{T}\rangle\rangle$
$= \mathbf{T}$

Note that in these two derivations, it seems needless to evaluate $[\![b]\!]$, the following derivation illustrates an case where the first argument is not a Boolean constant and the evaluation of the condition is needed.

$[\![\texttt{if if } \mathbf{F} \texttt{ then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi}]\!]$
$\langle\langle$Equation : 2 $p = \texttt{if } \mathbf{F} \texttt{ then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi}, p_1 = \mathbf{F}, p_2 = \mathbf{T}$
Case : $p \notin \mathbb{B}$
$q = [\![\texttt{if } \mathbf{F} \texttt{ then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi}]\!]$
$\langle\langle$Equation : 2 $p = \mathbf{F}, p_1 = \mathbf{F}, p_2 = \mathbf{T}$ Case : $p = \mathbf{F}\rangle\rangle$
$= [\![\mathbf{T}]\!]$
$\langle\langle$Equation : 1 $b = \mathbf{T}\rangle\rangle$
$= \mathbf{T}$
$\rangle\rangle$
$= [\![\texttt{if } \mathbf{T} \texttt{ then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi}]\!]$
$\langle\langle$Equation : 2 $p = \mathbf{T}, p_1 = \mathbf{F}, p_2 = \mathbf{T}$ Case : $p = \mathbf{T}\rangle\rangle$
$= [\![\mathbf{F}]\!]$
$\langle\langle$Equation : 1 $b = \mathbf{F}\rangle\rangle$
$= \mathbf{F}$

Using terms of PLB, we can define other logical operators.

$$\begin{array}{lcl} \texttt{not } p & \stackrel{\text{def}}{=} & \texttt{if } p \texttt{ then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi} \\ (p \texttt{ and } q) & \stackrel{\text{def}}{=} & \texttt{if } p \texttt{ then } q \texttt{ else } \mathbf{F} \texttt{ fi} \\ (p \texttt{ or } q) & \stackrel{\text{def}}{=} & \texttt{if } p \texttt{ then } \mathbf{T} \texttt{ else } q \texttt{ fi} \end{array}$$

In the chapter on logic we will prove that these definitions are indeed correct in that the defined operators behave as we expect them to.

Semantics of $List_T$

Perhaps oddly, we do not intend to assign semantics to the class $List_T$. The terms of the class represent themselves, *i.e.* we are interested in lists as lists. But still, semantic functions are not the only functions that can be defined by recursion on the structure of syntax, we can define other interesting functions on lists by recursion on the syntactic structure of one or more of the arguments.

For example, we can define a function that glues two lists together (given inputs L and M where $L, M \in List_T$, $append(L, M)$ is a list in $List_T$). It is defined recursively by on the (syntactic) structure of the first argument as follows:

$$\begin{array}{l} append([\,], M) = M \\ append(a{::}L, M) = a{::}(append(L, M)) \end{array}$$

The first equation of the definition says: if the first argument is the list $[\,]$, the result is just the second argument. The second equation of the definition says, if the first argument is a cons of the form $a{::}L$, then cons a on the *append* of L and M. Thus, there are two equations, one for each rule that could have been used to construct the first argument of the function.

We give some example computations with the definition of *append*.

$$\begin{aligned} &append(a{::}b{::}[\,],[\,]) \\ &= a{::}(append(b{::}[\,],[\,]) \\ &= a{::}b{::}(append([\,],[\,]) \\ &= a{::}b{::}[\,] \end{aligned}$$

Using the more compact notation or lists, we have shown $append((,[)a,b],[\,]) = [a,b]$. Using this notation for lists we can rewrite the derivation as follows:

$$\begin{aligned} &append([a,b],[\,]) \\ &= a{::}(append([b],[\,]) \\ &= a{::}b{::}(append([\,],[\,]) \\ &= a{::}b{::}[\,] \\ &= [a,b] \end{aligned}$$

We will use the more succinct notation for lists from now on, but do not forget that this is just a more readable display for the more cumbersome but precise notation which explicitly uses the cons constructor.

Here is another example.

$$\begin{aligned} &append([\,],[a,b]) \\ &= [a,b] \end{aligned}$$

We will discuss lists and operations on lists as well as ways to prove properties about lists in some depth in later chapters. For example, the rules for append immediately give $append((,[\,]),M) = M$, but the following equation is a theorem as well $append((,M),[\,]) = M$. For any individual list M we can compute with the rules for append and show this, but currently have no way to assert this in general for all M with out proving it by induction.

1.5 Remarks on Implementation

A number of programming languages provide excellent support for implementing abstract syntax almost succinctly as it has been presented above. This is especially true of the ML family of languages [23, 22, 25]. Scheme is also useful in this way [9]. Both ML and Scheme are languages in the family of functional programming languages. Of course we can define term structures in any modern programming language, but these languages provide particularly good support for this. Similarly, semantics is typically defined by recursion on the structure of the syntax and these languages make such definitions quite transparent, implementations appear syntactically close to the mathematical notions used above. The approach to implementing syntax and semantics in ML is taken in [**?**] and a similar approach using Scheme is followed in [9].

Part I

Logic

Chapter 2

Propositional Logic

> *One of the people present said: 'Persuade me that logic is useful.' – 'Do you want me to prove it to you?' He asked. – 'Yes.' – 'So I must produce a probative argument?' – He agreed. – 'Then how will you know if I produce a sophism?' – He said nothing. – 'You see,' he said, 'you yourself agree that all this is necessary, since without it you cannot even learn whether it is necessary or not.'* Epictetus Discourses[1] II xxv1 21.

In this chapter we present propositional logic: its syntax, semantics, and a formal proof system.

Without further ado, we say give the following definition for propositions.

Definition 2.1 (Proposition) A *proposition* is a statement that can, in principle, be either true or false.

Of course the true nature of propositions is open to some philosophical debate, we leave this debate to the philosophers and note that we are essentially adopting Wittgenstein's [37] definition of propositions as truth functional.

In the next sections we define the syntax of formulas and then describe their semantics (given an interpretation of the atomic parts.) The primitive vocabulary of symbols from which more complex terms of the language are constructed is called the *lexical* components. *Syntax* specifies the acceptable form or structures of lexical components allowed in the language. *Semantics* assigns meaning to the syntactic forms.

2.1 Syntax of Propositional Logic

We use *propositional variables* to stand for arbitrary propositions and we assume there is an infinite supply of these variables.

$$\mathcal{V} = \{p, q, r, p_1, q_1, r_1, p_2, \cdots\}$$

[1] Translated by Jonathan Barnes in his book [2].

Note that the fact that the set $\mathcal{V}$ is infinite is unimportant since no formula will ever require more than some fixed finite number of variables, however it is important that the number of variables we can select from is unbounded. The must always be a way to get another one.

We include the constant symbol $\bot$ (say "bottom").

Complex propositions are constructed by combining simpler ones with *propositional connectives*. For now we leave the meaning of the connectives unspecified and simply present them as one of the symbols $\wedge, \vee, \Rightarrow$ standing for *and*, *or* and *implies* respectively.

The syntax of propositional formulas (we denote the set as $\mathcal{P}$) can be described by a grammar as follows:

$$\mathcal{P} ::= \bot \mid x \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi$$

where

$\bot$ is a constant symbol,

$x \in \mathcal{V}$ is a propositional variable, and

$\phi, \psi \in \mathcal{P}$ are meta-variables denoting previously constructed propositional formulas.

To write the terms of the language $\mathcal{P}$ linearly (*i.e.* so that they can be written from left-to-right on a page), we insert parentheses to indicate the order of the construction of the term as needed *e.g.* $p \wedge q \vee r$ is ambiguous in that we do not know if it denotes a conjunction of a variable and a disjunction $(p \wedge (q \vee r))$ or it denotes the disjunction of a conjunction and a variable $((p \wedge q) \vee r)$.

Thus (written linearly) the following are among the terms of $\mathcal{P}$: $\bot$, p, q, $\neg q$, $p \wedge \neg q$, $((p \wedge \neg q) \vee q)$, and $\neg((p \wedge \neg q) \vee r)$.

We use the lowercase Greek letters ϕ and ψ (possibly subscripted) as *meta-variables* ranging over propositional formulas, that is, ϕ and ψ are variables that denote propositional formulas; note that they are not themselves propositional formulas and no actual propositional formula contains either of them.

Other views of syntax Alternaitvely, you can consider the alternatives in the description of the syntax of propostional logic as constructors.

$$\mathcal{P} ::= \bot \mid x \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi$$

The signatures of the constructors are given as follows:

$$\begin{aligned} mk\_bot : &\quad \mathcal{P} \\ mk\_var : &\quad \mathcal{V} \to \mathcal{P} \\ mk\_not : &\quad \mathcal{P} \to \mathcal{P} \\ mk\_and : &\quad (\mathcal{P} \times \mathcal{P}) \to \mathcal{P} \\ mk\_or : &\quad (\mathcal{P} \times \mathcal{P}) \to \mathcal{P} \\ mk\_implies : &\quad (\mathcal{P} \times \mathcal{P}) \to \mathcal{P} \end{aligned}$$

Thus, *mk_bot* is a constant of type $\mathcal{P}$, *i.e.* it is a propositional formula. The constructor *mk_var* maps variables in $\mathcal{V}$ to propositional formulas and so is

labelled as having the type $\mathcal{V} \to \mathcal{P}$. The constructor *mk_not* maps a previously constructed propositional formula to a new propositional formula (by sticking a *not* symbol in front) and so is labelled as having the type $\mathcal{P} \to \mathcal{P}$. We say it is a *unary connective* since it takes one argument. The constructors for *and*, *or*, and *implies* all take two arguments and so are called *binary conectives*. Theie arguments are pairs previously constructed propositional formulas and so they all have the signature $(\mathcal{P} \times \mathcal{P}) \to \mathcal{P}$.

Here are some formulas represented in different ways.

Linear Form	Constructor Form
$\bot$	mk_bot
p	$mk\_var(p)$
$\neg p$	$mk\_not(mk\_var(p))$
$p \wedge \neg p$	$mk\_and(mk\_var(p), mk\_not(mk\_var(p)))$
$p \Rightarrow \neg\bot$	$mk\_implies(mk\_var(p), mk\_not(mk\_bot))$
$((p \wedge \bot) \Rightarrow (p \vee \neg q))$	$mk\_implies(mk\_and(mk\_var(p), mk\_bot),$ $mk\_or(mk\_var(p), mk\_not(mk\_var(q))))$

2.1.1 Definitions

We can extend the language of propositional logic by allowing for definitions.

Definition 2.2. A *definition* is a schematic form that introduces a new symbol or formula as an abbreviation for another (possibly more complicated) formula. Definitions have the form

$$\mathbf{A} \stackrel{\text{def}}{=} \mathbf{B}$$

where the schematic form **A** is the abbreviation for the schematic formula **B**. **A** may introduce new symbols not in the language while **B** must be a formula of the language defined so -far, this includes those formulas given by the syntax as well as formulas that may include previously defined symbols.

This definition of "definition" is perhaps too abstract to be of much use, and yet the idea of introducing new definitions is one of the most natural ideas of mathematics. A few definitions are given below which should make the idea perfectly transparent.

If-and-only-if

Definition 2.3. The so-called *bi-conditional* or *if-and-only-if* connective is defined as follows:

$$(\phi \Leftrightarrow \psi) \stackrel{\text{def}}{=} ((\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi))$$

The symbol " $\stackrel{\text{def}}{=}$ " separates the left side of the definition, the thing being defined, from the right side which contains the definition. The left side of the definition may contain meta-variables which also appear on the right side.

2.2 Semantics

2.2.1 Truth Table Semantics

The meaning of a propositional formula depends only on the meaning of its parts. This fact suggests a method for determining the meaning of any propositional formula; *i.e.* consider all the possible values is parts may take. This leads to the idea of truth table semantics, the meaning of each connective is defined in terms of the meaning of each part, since each part can only take the values **T** or **F**, denoting *true* and *false* respectively. A two valued set like $\{\mathbf{T}, \mathbf{F}\}$ is called Boolean, and **T**and **F**are called Boolean values.

Thus, complete analysis of the possible values of true and false requires us to consider a only finite number of cases. Truth tables were first formulated by the philosopher Ludwig Wittgenstein.

The formula constant $\perp$ is mapped to the constant **F** as the following one row truth table indicates.

$\perp$
F

Negation is a unary connective (*i.e.* it only has one argument) that toggles the value of it's argument as the following truth table shows.

ϕ	$\neg\phi$
T	**F**
F	**T**

The truth functional interpretations of the binary connectives for conjunction, disjunction, implication, and if-and-only-if are summarized in the following truth table.

ϕ	ψ	$(\phi \wedge \psi)$	$(\phi \vee \psi)$	$(\phi \Rightarrow \psi)$	$(\phi \Leftrightarrow \psi)$
T	**T**	**T**	**T**	**T**	**T**
T	**F**	**F**	**T**	**F**	**F**
F	**T**	**F**	**T**	**T**	**F**
F	**F**	**F**	**F**	**T**	**T**

Thus, the truth or falsity of a formula is determined solely by the truth or falsity of its sub-terms:

$\phi \wedge \psi$: is true if both ϕ and ϕ are true and is false otherwise,
$\phi \vee \psi$: is true if one of ϕ or ϕ is true and is false otherwise,
$\phi \Rightarrow \psi$: is true if ϕ is false or if ψ is true and is false otherwise, and
$\phi \Leftrightarrow \psi$: is true if both ϕ and ψ are true or if they are both false and is false otherwise.

We remark that for any element of $\mathcal{P}$, although the number of cases (rows in a truth table) is finite, the total number of cases is exponential in the number of distinct variables. This means that, for each variable we must consider in a formula, the number of cases we must consider doubles. Complete analysis of a formula having no variables (*i.e.* its only base term is $\perp$) has $2^0 = 1$ row; a formula having one distinct variable has $2^1 = 2$ rows, two variables means four cases, three variables means eight, and so on. If the formula contains n distinct variables, there are 2^n possible combinations of true and false that the n variables may take.

Consider the following example of true table for the formula $((p \Rightarrow q) \vee z)$.

p	q	r	$(p \Rightarrow q)$	$((p \Rightarrow q) \vee r)$
T	**T**	**T**	**T**	**T**
T	**T**	**F**	**T**	**T**
T	**F**	**T**	**F**	**T**
T	**F**	**F**	**F**	**F**
F	**T**	**T**	**T**	**T**
F	**T**	**F**	**T**	**T**
F	**F**	**T**	**T**	**T**
F	**F**	**F**	**T**	**T**

Since the formula has three distinct variables, there are $2^3 = 8$ rows in the truth table. Notice that the fourth row of the truth table *falsifies* the formula, *i.e.* if p is true, q is false, and r is false, the formula $((p \Rightarrow q) \vee r)$ is false. All the other rows *satisfy* the formula *i.e.* all the other assignments of true and false to the variables of the formula make it true.

A formula having the same shape (*i.e.* drawn as a tree it has the same structure) , but only having two distinct variables is $((p \Rightarrow q) \vee p)$. Although there are three variable occurrences in the formula, (two occurrences of p and one occurrence of q), the distinct variables are p and q. To completely analyze the formula we only need $2^2 = 4$ rows in the truth table.

p	q	$(p \Rightarrow q)$	$((p \Rightarrow q) \vee p)$
T	**T**	**T**	**T**
T	**F**	**F**	**T**
F	**T**	**T**	**T**
F	**F**	**T**	**T**

Note that this formula is true for every assignment of Boolean values to the variables p and q.

Definition 2.4. A propositional formula is *satisfiable* if the column under the principal connective is true in any row of the truth table.

Definition 2.5. A propositional formula is *falsifiable* if the column under the principal connective is false in any row of the truth table.

Definition 2.6. A propositional formula is *valid* (or a *tautology*) if the column under the principal connective is true in every row of the truth table.

Definition 2.7. A propositional formula is a *contradiction* (or *unsatisfiable*) if the column under the principal connective is false in every row of the truth table.

A formula having the meaning T

We have not included a constant in the language $\mathcal{P}$ whose meaning is **T**. We can model such a constant it with the formula $\neg\bot$. The following truth table shows that this formula is always has meaning **T**.

$\bot$	$\neg\bot$
F	**T**

Note that *any* tautology could serve as our definition of *true*, but this is the simplest such formula in the language $\mathcal{P}$.

2.2.2 Assignments and Valuations

Definition 2.8. An *assignment* is a function that maps propositional variables to one of the Boolean values **T** or **F**.

We use the variables $\alpha, \alpha', \hat{\alpha} \cdots$ to denote assignments, $\alpha : \mathcal{V} \to \{\mathbf{T}, \mathbf{F}\}$. Each assignment corresponds to a row in part of a truth table where variables are given values.

Since the meaning of a propositional formula is determined by the meaning of its parts, it is reasonable to assume that an assignment to just the variables is enough to determine the meaning of a formula; it is.

Definition 2.9. A *valuation* is a function that takes an assignment and a propositional formula as input and returns a Boolean value, depending on whether the assignment determines the formulas value to be **T** or **F**.

We define the (recursive) valuation function *val* by induction on the structure of the formula as follows.

Definition 2.10.

$$
\begin{array}{lll}
val(\alpha, \bot) & = \mathbf{F} & \\
val(\alpha, x) & = \alpha(x) & \text{whenever } x \in \mathcal{V} \\
val(\alpha, \neg\phi) & = \mathbf{not}(val(\alpha, \phi)) & \\
val(\alpha, \phi \wedge \psi) & = val(\alpha, \phi) \ \mathbf{and} \ val(\alpha, \psi) & \\
val(\alpha, \phi \vee \psi) & = val(\alpha, \phi) \ \mathbf{or} \ val(\alpha, \psi) & \\
val(\alpha, \phi \Rightarrow \psi) & = \mathbf{not}(val(\alpha, \phi)) \ \mathbf{or} \ val(\alpha, \psi) & \\
val(\alpha, \phi \Leftrightarrow \psi) & = val(\alpha, \phi \Rightarrow \psi) \ \mathbf{and} \ val(\alpha, \psi \Rightarrow \phi) &
\end{array}
$$

The definition specifies how to compute the valuation of any propositional formula (under assignment α) by including one equation for each rule in the grammar of $\mathcal{P}$..

Definition 2.11. An assignment α *satisfies* a formula ϕ if and only if $val(\alpha, \phi) = \mathbf{T}$. In this case we write $\alpha \models \phi$ and say "α *models* ϕ".

Definition 2.12. An assignment α *falsifies* a formula ϕ if and only if $val(\alpha, \phi) = \mathbf{F}$. In this case we write $\alpha \not\models \phi$ and say "α *does not model* ϕ."

Definition 2.13. If a formula ϕ is satisfied by every assignment (*i.e.* if it is true in in every row of the truth table, it is *valid*) we write $\models \phi$. In this case we say ϕ *is true in all models.*

As an example, suppose we define an assignment α as follows[2]

$$\begin{aligned}\alpha(p) &= \mathbf{T}\\ \alpha(q) &= \mathbf{F}\\ \alpha(r) &= \mathbf{T}\end{aligned}$$

Then, the valuation of the formula $((p \Rightarrow q) \vee r)$ is computed as follows.

$$\begin{aligned}&val(\alpha, ((p \Rightarrow q) \vee r))\\ &\quad = val(\alpha, (p \Rightarrow q)) \textbf{ or } val(\alpha, r)\\ &\quad = (\textbf{not}(val(\alpha, p)) \textbf{ or } val(\alpha, q)) \textbf{ or } \alpha(r)\\ &\quad = (\textbf{not}(\alpha(p)) \textbf{ or } \alpha(q)) \textbf{ or } \mathbf{T}\\ &\quad = (\textbf{not}(\mathbf{T}) \textbf{ or } \mathbf{F}) \textbf{ or } \mathbf{T}\\ &\quad = (\mathbf{F} \textbf{ or } \mathbf{F}) \textbf{ or } \mathbf{T}\\ &\quad = \mathbf{F} \textbf{ or } \mathbf{T}\\ &\quad = \mathbf{T}\end{aligned}$$

Thus, we have shown $\alpha \models ((p \Rightarrow q) \vee r)$.

Consider the valuation of another formula under same assignment.

$$\begin{aligned}&val(\alpha, ((p \Rightarrow q) \vee q))\\ &\quad = val(\alpha, (p \Rightarrow q)) \textbf{ or } val(\alpha, q)\\ &\quad = (\textbf{not}(val(\alpha, p)) \textbf{ or } val(\alpha, q)) \textbf{ or } \alpha(q)\\ &\quad = (\textbf{not}(\alpha(p)) \textbf{ or } \alpha(q)) \textbf{ or } \mathbf{F}\\ &\quad = (\textbf{not}(\mathbf{T}) \textbf{ or } \mathbf{F}) \textbf{ or } \mathbf{F}\\ &\quad = (\mathbf{F} \textbf{ or } \mathbf{F}) \textbf{ or } \mathbf{F}\\ &\quad = \mathbf{F} \textbf{ or } \mathbf{F}\\ &\quad = \mathbf{F}\end{aligned}$$

We have shown $\alpha \not\models ((p \Rightarrow q) \vee q)$.

[2]Technically, since assignments are functions from the set of propositional variables to Boolean values, we should specify what α does on *all* the propositional variables. But if a propositional variable (say p_{127}) does not occur in a formula (say ϕ), then the value of $\alpha(p_{127})$ will never be used in in the evaluation of $val(\alpha, \phi)$. Thus, we *don't care* what value α assigns to any variable not occurring in the formula and so we don't bother to specify what value they have under α. If partially specified functions bother you, pick your favorite Boolean value and assume all unspecified variables get mapped to that value by α.

2.3 Sequents

Sequents are pairs of formula lists used to characterize a point in a proof. One element of the pair lists the assumptions that are in force at the point in a proof characterized by the sequent and the other lists the goals, one of which we must prove to complete a proof of the sequent. The sequent formulation of proofs, presented below, was first given by the German logician Gerhard Gentzen in 1935 [**?**].

We will use letters (possibly subscripted) from the upper-case Greek alphabet as meta-variables that range over (possibly empty) lists of formulas. Thus Γ, Γ_1, Γ_2, Δ, Δ_1, and Δ_2 all stand for arbitrary elements of the class $List_{\mathcal{P}}$[3].

Definition 2.14 (Sequent) A *sequent* is a pair of lists of formulas $\langle \Gamma, \Delta \rangle$. The list Γ is called the *antecedent* of the sequent and the list Δ is called the *succedent* of the sequent.

The standard notational convention is to write the sequent $\langle \Gamma, \Delta \rangle$ as $\Gamma \vdash \Delta$ where the symbol "$\vdash$" is called *turnstile*.

2.3.1 Semantics of Sequents

The meaning of a propositional sequent can be interpreted to be a Boolean value. We do so by transforming the sequent into a propositional formula and then using truth tables to determine the truth value of the sequent. Since all the formulas in the antecedent Γ are assumed to be true and, if they really are all true, then the conjunction of all of them is as well. Assuming the antecedents really are all true, the sequent is valid if at least one of the formulas in the succedent Δ is true, but this is the case only if the disjunction of all the formulas in Δ is true. We will formalize this idea once we have some operations allowing us to form the conjunction and disjunction of the formulas in a lists.

Conjunctions and Disjunctions of lists of Formulas

Informally, if Γ is the list $[\phi_1, \phi_2, \cdots, \phi_n]$ then

$$\bigwedge_{\phi \in \Gamma} \phi = (\phi_1 \wedge (\phi_2 \wedge (\cdots (\phi_n \wedge (\neg \bot)) \cdots)))$$

Dually, if Δ is the list $[\psi_1, \psi_2, \cdots, \psi_m]$ then

$$\bigvee_{\phi \in \Delta} \phi = (\psi_1 \vee (\psi_2 \vee (\cdots (\psi_m \vee (\bot)) \cdots)))$$

These operations can be formally defined by recursion on the structure of their list arguments as follows:

[3]The syntax for the class $List_T$, lists over some set T, was defined in Chapter **??**.

Definition 2.15. Conjunction over a list The function which conjoins all the elements in a list is defined on the structure of the list by the following two recursive equations.

$$\bigwedge_{\phi \in [\,]} \phi = \neg\bot$$

$$\bigwedge_{\phi \in (\psi :: \Gamma)} \phi = (\psi \wedge (\bigwedge_{\phi \in \Gamma} \phi))$$

The first equation defines the conjunction of formulas in the empty list simply to be the formula $\neg\bot$ (*i.e.* the formula having the meaning **T**). The formula $\neg\bot$ is the right identity for conjunction *i.e.* the following is a tautology $((\phi \wedge \neg\bot) \Leftrightarrow \phi)$. You might verify the claim by constructing the truth table for this formula.

You might argue semantically that this is the right choice for the empty list as follows: the conjunction of the formulas in a list is valid if and only if all the formulas in the list are valid, but there are *no* formulas in the empty list, so all of them (all none of them) are valid.

The second equation in the definition says that the conjunction over a list constructed by a cons is the conjunction of the individual formula that is the head of the list with the conjunction over the tail of the list.

Definition 2.16. Disjunction over a list The function which creates a disjunction of all the elements in a list is defined on the structure of the list by the following two recursive equations.

$$\bigvee_{\phi \in [\,]} \phi = \bot$$

$$\bigvee_{\phi \in (\psi :: \Gamma)} \phi = (\psi \vee (\bigvee_{\phi \in \Gamma} \phi))$$

The first equation defines the disjunction of formulas in the empty list simply to be the formula $\bot$ (*i.e.* the formula whose meaning is **F**). The formula $\bot$ is the right identity for disjunction *i.e.* the following is a tautology $((\phi \vee \bot) \Leftrightarrow \phi)$. Again, you might verify the claim by constructing the truth table for this formula.

An informal semantic argument for this choice to represent the empty list might go as follows: The disjunction of the formulas in a list is valid if and only if some formula in the list is valid, but there are *no* formulas in the empty list, so none of them are valid and the disjunction must be false.

The second equation in the definition says that the disjunction over a list constructed by a cons is the disjunction of the individual formula that is the head of the list with the disjunction over the tail of the list.

Semantic interpretation of sequents

Now that we have operators for constructing conjunctions and disjunctions over lists of formulas, we give a definition of sequent validity in terms of the validity of a formula of $\mathcal{P}$.

Definition 2.17. Formula interpretation of a sequent

$$[\![\Gamma \vdash \Delta]\!] = ((\bigwedge_{\phi \in \Gamma} \phi) \Rightarrow (\bigvee_{\psi \in \Delta} \psi))$$

Thus $[\![\Gamma \vdash \Delta]\!]$ is a translation of the sequent $\Gamma \vdash \Delta$ into a formula. Using this translation, we semantically characterize the validity of a sequent as follows.

Definition 2.18. Sequent validity A sequent $\Gamma \vdash \Delta$ is *valid* if and only if

$$\models [\![\Gamma \vdash \Delta]\!]$$

That is, a sequent is valid if and only if

$$\models (\bigwedge_{\phi \in \Gamma} \phi) \Rightarrow (\bigvee_{\psi \in \Delta} \psi)$$

We consider the cases as to whether the antecedent and/or succedent are empty. If $\Gamma = []$ then the sequent $\Gamma \vdash \Delta$ is valid if and only if one of the formulas in Δ is. If $\Delta = []$ then the sequent $\Gamma \vdash \Delta$ is valid if and only if one of the formulas in Γ is not valid. If both the antecedent and the succedent are empty, *i.e.* $\Gamma = \Delta = []$, then the sequent $\Gamma \vdash \Delta$ is not valid since

$$(\bigwedge_{\phi \in []} \phi) \Rightarrow (\bigvee_{\psi \in []} \psi) \text{ is equivalent to the formula } (\neg\bot \Rightarrow \bot)$$

and $(\neg\bot \Rightarrow \bot)$ is a contradiction. We verify this claim with the following truth table.

$\bot$	$\neg\bot$	$(\neg\bot \Rightarrow \bot)$
F	**T**	**F**

2.4 Sequent Schemas, Substitutions and Matching

Proof rules are schemas used to specify a single step of inference. The proof rule schemas are specified by arranging schematic sequents in particular configurations to indicate which parts of the rule are related to which. For example,

the rule for decomposing an implication on the left side of the turnstile is given as:

$$\frac{\Gamma_1, \Gamma_2 \vdash \phi, \Delta \qquad \Gamma_1, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, (\phi \Rightarrow \psi), \Gamma_2 \vdash \Delta}$$

There are three schematic sequents in this rule.

$$\Gamma_1, \Gamma_2 \vdash \phi, \Delta$$
$$\Gamma_1, \psi, \Gamma_2 \vdash \Delta$$
$$\Gamma_1, (\phi \Rightarrow \psi), \Gamma_2 \vdash \Delta$$

Each of these schematic sequents specifies a pattern that an actual sequent might (or might not) match. By an actual sequent, we mean a sequent that contains no meta-variables. A sequent (call it S) *matches* a sequent schema (call it $\hat{\mathcal{S}}$) if there is some way of substituting actual formula lists and formulas for meta-variables in the schema $\hat{\mathcal{S}}$ (elements of $List_{\mathcal{P}}$ for list meta-variables and elements of $\mathcal{P}$ for formula meta-variables) so the that the resulting sequent is identical to S.

2.5 Propositional Proof Rules

Proof rules for the propositional sequent calculus have one of the following three forms:

$$\frac{}{\mathcal{C}}(N) \qquad \frac{\mathcal{H}}{\mathcal{C}}(N) \qquad \frac{\mathcal{H}_1 \ \mathcal{H}_2}{\mathcal{C}}(N)$$

where $\mathcal{C}, \mathcal{H}, \mathcal{H}_1$, and $\mathcal{H}_2$ are all schematic sequents. N is the name of the rule. The $\mathcal{H}$ patterns are the *premises* (or *hypotheses*) of the rule and the pattern $\mathcal{C}$ is the *goal* (or *conclusion*) of the rule. Rules having no premises are called *axioms*.

Rules that operate on formulas in the antecedent (on the left side of $\vdash$) of a sequent are called *elimination rules* and rules that operate on formulas in the consequent (the right side of $\vdash$) of a sequent are called *introduction rules*.

2.5.1 Axiom Rules

If there is a formula that appears in both the antecedent and the consequent of a sequent then the sequent is valid. The axiom rule reflects this and has the following form:

$$\frac{}{\Gamma_1, \phi, \Gamma_2 \vdash \Delta_1, \phi, \Delta_2} \ (\text{Ax})$$

Also, since false ($\bot$) implies anything, if the formula $\bot$ appears in the antecedent of a sequent that sequent is trivially valid.

$$\frac{}{\Gamma_1, \bot, \Gamma_2 \vdash \Delta} \ (\bot\text{Ax})$$

2.5.2 Conjunction Rules

On the right

A conjunction $(\phi \wedge \psi)$ is true when both ϕ is true and when ψ is true. Thus, the proof rule for a conjunction on the right is given as follows:

$$\frac{\Gamma \vdash \Delta_1, \phi, \Delta_2 \qquad \Gamma \vdash \Delta_1, \psi, \Delta_2}{\Gamma \vdash \Delta_1, (\phi \wedge \psi), \Delta_2} \ (\wedge\text{R})$$

On the left

On the other hand, if we have a hypothesis that is a conjunction of the form $(\phi \wedge \psi)$, then we know both ϕ and ψ are true.

$$\frac{\Gamma_1, \phi, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, (\phi \wedge \psi), \Gamma_2 \vdash \Delta} \ (\wedge\text{L})$$

2.5.3 Disjunction Rules

A disjunction $(\phi \vee \psi)$ is true when either ϕ is true or when ψ is true. Thus, the proof rule for proving a goal having disjunctive form is the following.

$$\frac{\Gamma \vdash \Delta_1, \phi, \psi, \Delta_2}{\Gamma, \vdash \Delta_1, (\phi \vee \psi), \Delta_2} \ (\vee\text{R})$$

On the other hand, if we have a hypothesis that is a disjunction of the form $(\phi \vee \psi)$, then, since we don't know which of the disjuncts is true (but since we are assuming the disjunction is true, one of them must be), we must continue by cases on ϕ and ψ, showing that the sequent $\Gamma_1, \phi, \Gamma_2 \vdash \Delta$ is true and that the sequent $\Gamma_1, \psi, \Gamma_2 \vdash \Delta$ is as well.

$$\frac{\Gamma_1, \phi, \Gamma_2 \vdash \Delta \qquad \Gamma_1, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, (\phi \vee \psi), \Gamma_2 \vdash \Delta} \ (\vee\text{L})$$

2.5.4 Implication Rules

A implication $(\phi \Rightarrow \psi)$ is provable when, assuming ϕ, you can prove ψ. Thus, the proof rule for proving a goal having implicational form is the following.

$$\frac{\Gamma, \phi \vdash \Delta_1, \psi, \Delta_2}{\Gamma \vdash \Delta_1, (\phi \Rightarrow \psi), \Delta_2} \ (\Rightarrow\text{R})$$

If we have a hypothesis that is a implication of the form $(\phi \Rightarrow \psi)$ and we wish to prove some formula in the conclusion Δ, working backward, we must show that adding ψ to the hypotheses proves Δ and also that adding ϕ to the conclusion (*i.e.* ϕ, Δ) is provable. Structurally, this rule is the most complicated.

$$\frac{\Gamma_1, \Gamma_2 \vdash \phi, \Delta \quad \Gamma_1, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, (\phi \Rightarrow \psi), \Gamma_2 \vdash \Delta} \ (\Rightarrow\text{L})$$

Note that if ϕ is in Γ then this is just like *Modus Ponens* since the left subgoal becomes an instance of the axiom rule.

2.5.5 Negation Rules

Since a negation $\neg\phi$ can be viewed as an abbreviation for $\phi \Rightarrow \bot$, the proof rule for negation is related to that of implication (see above.)

$$\frac{\Gamma, \phi \vdash \Delta_1, \Delta_2}{\Gamma \vdash \Delta_1, \neg\phi, \Delta_2} \ (\neg\text{R})$$

If you have a negated formula $\neg\phi$ in the antecedent, working backward, you can swap the formula ϕ to the other side of the turnstile and try to prove it directly.

$$\frac{\Gamma_1, \Gamma_2 \vdash \phi, \Delta}{\Gamma_1, \neg\phi, \Gamma_2 \vdash \Delta} \ (\neg\text{L})$$

2.6 Proofs

We have the proof rules, now we define what a proof is. A formal proof is a tree structure where the nodes of the tree are sequents, the leaves of the tree are instances of one of the axiom rules, and there is an edge between sequents if the sequents form an instance of some proof rule. We can formally describe an inductive data-structure for representing sequent proofs.

Definition 2.19. A *proof tree* having root sequent S is defined inductively as follows:

i.) If the sequent S is an instance of one of the axioms rules whose name is N, then

$$\frac{}{S} \ (N)$$

is a proof tree whose root is the sequent S.

ii.) If ρ_1 is a proof tree whose root is the sequent S_1 and, if

$$\frac{S_1}{S} \ (N)$$

is an instance of some proof rule having a single premise, then the tree

$$\dfrac{\begin{array}{c}\vdots\\ \rho_1\end{array}}{S}\,(N)$$

is a proof tree whose root is the sequent S.

iii.) If ρ_1 is a proof tree with root sequent S_1 and ρ_2 is a proof tree with root sequent S_2 and, if

$$\dfrac{S_1 \qquad S_2}{S}\,(N)$$

is an instance the proof rule which has two premises, then the tree

$$\dfrac{\begin{array}{c}\vdots\\ \rho_1\end{array} \qquad \begin{array}{c}\vdots\\ \rho_2\end{array}}{S}\,(N)$$

is a proof tree whose root is the sequent S.

Although proof trees were just defined by starting with the leaves and building them toward the root, the proof rules are typically applied in the reverse order, *i.e.* the goal sequent is scanned to see if it is an instance of an axiom rule, if so we're done. If the sequent is not an instance of an axiom rule and it contains some non-atomic formula on the left or right side, then the rule for the principle connective of that formula is matched against the sequent. The resulting substitution is applied to the schematic sequents in the premises of the rule. The sequents generated by applying the matching substitution to the premises are placed in the proper positions relative to the goal. This process is repeated on incomplete leaves of the tree (leaves that are not instances of axioms) until all leaves are either instances of an axiom rule, or until all the formulas in the sequents at the leaves of the tree are atomic and are not instances of an axiom rule. In this last case, there is no proof of the goal sequent.

We present some examples.

Example 2.1. Consider the sequent $(p \vee q) \vdash (p \vee q)$. The following substitution verifies the match of the sequent against the goal of the axiom rule as follows:

$$\sigma_1 = \begin{cases} \Gamma_1 = [\,] \\ \Gamma_2 = [\,] \\ \Delta_1 = [\,] \\ \Delta_2 = [\,] \\ \phi = (p \vee q) \end{cases}$$

Thus, the following proof tree proves the sequent.

$$\frac{}{(p \vee q) \vdash (p \vee q)}\ (\text{Ax})$$

Example 2.2. Consider the sequent $(p \vee q) \vdash (q \vee p)$. It is not an axiom, since $(p \vee q)$ is distinct from $(q \vee p)$. The sequent matches both the $\vee$L-rule and the $\vee$R-rule. We match sequent against the $\vee$R-rule which results in the following substitution:

$$\sigma_1 = \begin{cases} \Gamma := [(p \vee q)] \\ \Delta_1 := [\,] \\ \Delta_2 := [\,] \\ \phi := p \\ \psi := q \end{cases}$$

The sequent that results from applying this substitution to the schematic sequent in the premise of the rule $\vee$R results in the sequent $(p \vee q) \vdash q, p$.

Thus far we have constructed the following partial proof:

$$\frac{(p \vee q) \vdash q, p}{(p \vee q) \vdash (q \vee p)}\ (\vee\text{R})$$

Now we match the sequent on the incomplete branch of the proof against the $\vee$L-rule. This is the only rule that matches since the sequent is not an axiom and only contains one non-atomic formula, namely the $(q \vee p)$ on the left side. The match generates the following substitution.

$$\sigma_2 = \begin{cases} \Gamma_1 := [\,] \\ \Gamma_2 := [\,] \\ \Delta := [q, p] \\ \phi := q \\ \psi := p \end{cases}$$

Applying this substitution to the premises of the $\vee$L-rule results in the sequents $p \vdash q, p$ and $q \vdash q, p$. Placing them in their proper positions results in the following partial proof tree.

$$\frac{\dfrac{p \vdash q, p \qquad q \vdash q, p}{(p \vee q) \vdash q, p}\ (\vee\text{L})}{(p \vee q) \vdash (q \vee p)}\ (\vee\text{R})$$

In this case, both incomplete branches are instances of the axiom rule. The matches are:

$$\sigma_3 = \begin{cases} \Gamma_1 := [\,] \\ \Gamma_2 := [\,] \\ \Delta_1 := [q] \\ \Delta_2 := [\,] \\ \phi := p \end{cases} \qquad \sigma_4 = \begin{cases} \Gamma_1 := [\,] \\ \Gamma_2 := [\,] \\ \Delta_1 := [\,] \\ \Delta_2 := [p] \\ \phi := q \end{cases}$$

These matches verify that the incomplete branches are indeed axioms and the final proof tree appears as follows:

$$\dfrac{\dfrac{\dfrac{}{p \vdash q, p}\,(\text{Ax}) \qquad \dfrac{}{q \vdash q, p}\,(\text{Ax})}{(p \vee q) \vdash q, p}\,(\vee\text{L})}{(p \vee q) \vdash (q \vee p)}\,(\vee\text{R})$$

2.7 Some Useful Tautologies

The reader should try to prove the following tautologies. We use the symbol $\top$ as an abbreviation for the true formula $\neg\bot$.

i. $\neg\neg\phi \Leftrightarrow \phi$

ii. $\neg\phi \Leftrightarrow (\phi \Rightarrow \bot)$

iii. $(\phi \Rightarrow \psi) \Leftrightarrow \neg\phi \vee \psi$

iv. $\neg(\phi \wedge \psi) \Leftrightarrow (\neg\phi \vee \neg\psi)$

v. $\neg(\phi \vee \psi) \Leftrightarrow (\neg\phi \wedge \neg\psi)$

vi. $(\phi \vee \psi) \Leftrightarrow (\psi \vee \phi)$

vii. $(\phi \wedge \psi) \Leftrightarrow (\psi \wedge \phi)$

viii. $((\phi \vee \psi) \vee \varphi) \Leftrightarrow (\phi \vee (\psi \vee \varphi))$

ix. $((\phi \wedge \psi) \wedge \varphi) \Leftrightarrow (\phi \wedge (\psi \wedge \varphi))$

x. $(\phi \vee \bot) \Leftrightarrow \phi$

xi. $(\phi \wedge \top) \Leftrightarrow \phi$

xii. $(\phi \vee \top) \Leftrightarrow \top$

xiii. $(\phi \wedge \bot) \Leftrightarrow \bot$

xiv. $(\phi \vee \neg\phi) \Leftrightarrow \top$

xv. $(\phi \wedge \neg\phi) \Leftrightarrow \bot$

xvi. $(\phi \wedge (\psi \vee \varphi)) \Leftrightarrow (\phi \wedge \psi) \vee (\phi \wedge \varphi)$

xvii. $(\phi \vee (\psi \wedge \varphi)) \Leftrightarrow (\phi \vee \psi) \wedge (\phi \vee \varphi)$

2.8 Complete Sets of Connectives

Definition 2.20 (Complete set) A set of connectives, $\mathcal{C}$,

$$\mathcal{C} \subseteq \{\bot, \neg, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$$

is *complete* if the connectives not in $\mathcal{C}$ can be defined in terms of the connectives in $\mathcal{C}$.

Example 2.3. The following definitions show that the set $\{\neg, \vee\}$ is complete.

$$\begin{array}{lrcl}
1.) & \bot & \stackrel{\text{def}}{=} & \neg(\phi \vee \neg\phi) \\
2.) & (\phi \wedge \psi) & \stackrel{\text{def}}{=} & \neg(\neg\phi \vee \neg\psi) \\
3.) & (\phi \Rightarrow \psi) & \stackrel{\text{def}}{=} & \neg\phi \vee \psi \\
4.) & (\phi \Leftrightarrow \psi) & \stackrel{\text{def}}{=} & \neg(\neg(\neg\phi \vee \psi) \vee \neg(\neg\psi \vee \phi))
\end{array}$$

To verify that these definitions are indeed correct, you could verify that the columns of the truth table for the defined connective match (row-for-row) the truth table for the definition. Alternatively, you could replace the symbol " $\stackrel{\text{def}}{=}$ " by "$\Leftrightarrow$" and use the sequent proof rules to verify the resulting formulas, *e.g.* to prove the definition for $\bot$ given above is correct, prove the sequent $\vdash \bot \Leftrightarrow \neg(\phi \vee \neg\phi)$. Another method of verification would be to do equational style proofs starting with the left-hand side of the definition and rewriting to the right hand side.

Here are example verifications using the equational style of proof. We label each step in the proof by the equivalence used to justify it or, if the step follows from a definition we say which one.

$$\begin{array}{lll}
1.) & \bot \stackrel{\langle i \rangle}{\Longleftrightarrow} \neg\neg\bot \stackrel{\langle \top\, def \rangle}{\Longleftrightarrow} \neg\top \stackrel{\langle xiv \rangle}{\Longleftrightarrow} \neg(\phi \vee \neg\phi) & \\
2.) & (\phi \wedge \psi) \stackrel{\langle i \rangle}{\Longleftrightarrow} \neg\neg(\phi \wedge \psi) \stackrel{\langle iv \rangle}{\Longleftrightarrow} \neg(\neg\phi \vee \neg\psi) & \\
3.) & (\phi \Rightarrow \psi) \stackrel{\langle iii \rangle}{\Longleftrightarrow} \neg\phi \vee \psi & \\
4.) & (\phi \Leftrightarrow \psi) & \stackrel{\langle \Leftrightarrow\, def. \rangle}{\Longleftrightarrow} (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \\
 & & \stackrel{\langle iii \rangle}{\Longleftrightarrow} (\neg\phi \vee \psi) \wedge (\psi \Rightarrow \phi) \\
 & & \stackrel{\langle iii \rangle}{\Longleftrightarrow} (\neg\phi \vee \psi) \wedge (\neg\psi \vee \phi) \\
 & & \stackrel{\langle 2 \rangle}{\Longleftrightarrow} \neg(\neg(\neg\phi \vee \psi) \vee \neg(\neg\psi \vee \phi))
\end{array}$$

2.9 Boolean Algebra

We have presented propositional logic syntax and have give semantics (meaning) based on truth tables over the set of truth values $\{T, F\}$. An alternative meaning can be assigned to propositional formulas by translating them into algebraic form over the natural numbers and then looking at the congruences modulo 2,

i.e. by claiming they're *congruent* to 0 or 1 depending on whether they're even or odd.

Such an interpretation is correct if it makes all the same formulas true.

2.9.1 Modular Arithmetic

Congruence (of which modular arithmetic is one kind) is an interesting topic of discrete mathematics in its own right. We will only present enough material here to make the association between propositional logic and its algebraic interpretation.

Remark 2.1. Recall that for every integer a and every natural number $m > 0$, there exists a integers q and r where $0 \leq r < m$ such that the following equation holds:

$$a = qm + r$$

We call q the *quotient* and r the *remainder* If $r = 0$ (there is no remainder) them we say m *divides* a *e.g.* $a \div m = q$.

Definition 2.21. Two integers are *congruent* modulo 2, if and only if they have the same remainder when divided by 2. In this case we write

$$a \equiv b (\mathrm{mod}\, 2)$$

Example 2.4.

$$\begin{array}{ll} 0 \equiv 0(\mathrm{mod}\, 2) & a = 0, k = 0, r = 0 \\ 1 \equiv 1(\mathrm{mod}\, 2) & a = 1, k = 0, r = 1 \\ 2 \equiv 0(\mathrm{mod}\, 2) & a = 2, k = 1, r = 0 \\ 3 \equiv 1(\mathrm{mod}\, 2) & a = 3, k = 1, r = 1 \\ 4 \equiv 0(\mathrm{mod}\, 2) & a = 4, k = 2, r = 0 \\ 5 \equiv 1(\mathrm{mod}\, 2) & a = 5, k = 2, r = 1 \end{array}$$

Theorem 2.1. The following three properties hold.

i.) $a \equiv a(\mathrm{mod}\, 2)$
ii.) If $a \equiv b(\mathrm{mod}\, 2)$ then $b \equiv a(\mathrm{mod}\, 2)$
iii.) If $a \equiv b(\mathrm{mod}\, 2)$ and $b \equiv c(\mathrm{mod}\, 2)$ then $a \equiv c(\mathrm{mod}\, 2)$

Theorem 2.2. If $a \in \mathbb{Z}$ is even, then $a \equiv 0(\mathrm{mod}\, 2)$ and if a is odd, then $a \equiv 1(\mathrm{mod}\, 2)$.

Theorem 2.3. If $a \equiv c(\mathrm{mod}\, n)$ and $b \equiv d(\mathrm{mod}\, n)$ then

$$\begin{array}{l} a + b \equiv c + d(\mathrm{mod}\, n) \\ a \cdot b \equiv c \cdot d(\mathrm{mod}\, n) \end{array}$$

Example 2.5. Since $5 \equiv 3 (\mathrm{mod}\, 2)$ and $10 \equiv 98 (\mathrm{mod}\, 2)$

$$5 + 10 \equiv 3 + 98 (\mathrm{mod}\, 2) \quad \text{and} \quad 5 \cdot 10 \equiv 3 \cdot 98 (\mathrm{mod}\, 2)$$

To see this note the following:

$5 + 10 = 15$ and $15 = 7 \cdot 2 + 1$, so $5 + 10 \equiv 1 (\mathrm{mod}\, 2)$
$3 + 98 = 101$ and $101 = 50 \cdot 2 + 1$, so $3 + 98 \equiv 1 (\mathrm{mod}\, 2)$
so by properties (ii) and (iii) of Theorem 1.1
$5 + 10 \equiv 3 + 98 (\mathrm{mod}\, 2)$

Prove to yourself that $5 \cdot 10 \equiv 3 \cdot 98 (\mathrm{mod}\, 2)$.

Definition 2.22. We will write $n (\mathrm{mod}\, 2)$ to denote the remainder of $n \div 2$. So, $5 (\mathrm{mod}\, 2) = 1$ and $28 (\mathrm{mod}\, 2) = 0$.

Theorem 2.4. The following identities hold.

$$2p \equiv 0 (\mathrm{mod}\, 2)$$
$$p^2 \equiv p (\mathrm{mod}\, 2)$$

2.9.2 Translation from Propositional Logic

In this section we define a function that maps propositional formulas to algebraic formulas.

We start the translation with falsity ($\bot$) and conjunction. Conjunction is easily seen to correspond to multiplication. Negation is defined next, and then using DeMorgan's laws, translations for disjunction and implication are given.

Falsity

We interpret $\bot$ as 0, so the translation function maps $\bot$ to 0, no matter what the assignment is.

$$\mathcal{M}[\![\bot]\!] \quad = \quad 0$$

Variables

Propositional variables are just mapped to variables in the algebraic language

$$\mathcal{M}[\![x]\!] \quad = \quad x$$

Conjunction

Consider the following tables for multiplication and the table for conjunction.

a	b	ab
1	1	1
1	0	0
0	1	0
0	0	0

a	b	$a \wedge b$
T	T	T
T	F	F
F	T	F
F	F	F

This table is identical to the truth table for conjunction ($\wedge$) if we replace 1 by T, 0 by F and the symbol for multiplication ($\cdot$) by the symbol for conjunction ($\wedge$). Thus, we get the following translation.

$$\mathcal{M}[\![\phi \wedge \psi]\!] \quad = \quad \mathcal{M}[\![\phi]\!] \cdot \mathcal{M}[\![\psi]\!]$$

Negation

Notice that addition by 1 modulo 2 toggles values.

$$1 + 1 = 2 \text{ and } 2 \equiv 0 (\text{mod}\, 2) \text{ and } 0 + 1 = 1$$

The following tables show addition by 1 modulo 2 and the truth table for negation to illustrate that the translating negations to addition by 1 give the correct results.

a	$a + 1 (\text{mod}\, 2)$
1	0
0	1

a	$\neg a$
T	F
F	T

The translation is defined as follows:

$$\mathcal{M}[\![\neg\phi]\!] = (\mathcal{M}[\![\phi]\!] + 1)$$

Exclusive-Or

We might hope that disjunction would be properly modeled by addition ... "If wishes were horses, beggars would ride." Consider the table for addition modulo 2 and compare it with the table for disjunction – clearly they do not match.

a	b	$a + b (\text{mod}\, 2)$
1	1	0
1	0	1
0	1	1
0	0	0

a	b	$a \vee b$
T	T	T
T	F	T
F	T	T
F	F	F

The problem is that $1 + 1 \equiv 0 (\text{mod}\, 2)$ while we want that entry to be 1, *i.e.* if p and q are both T, $p \vee q$ should be T as well.

But the addition table does correspond to a useful propositional connective (one we haven't introduced so far) – *exclusive or* – often written as $(p \oplus q)$ and which is true if one of the p or q is true *but not both.* It's truth table is given as follows:

a	b	$a \oplus b$
T	T	F
T	F	T
F	T	T
F	F	F

Disjunction

We can derive disjunction using the following identity of propositional logic and the translation rules we have defined so far.

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

Exercise 2.1. Verify this identity by using a truth table.

By the translation we have so far

$$\begin{aligned}
&\mathcal{M}[\![\neg(\neg p \wedge \neg q)]\!] \\
&\quad = \mathcal{M}[\![(\neg p \wedge \neg q)]\!] + 1 \\
&\quad = (\mathcal{M}[\![\neg p]\!] \cdot \mathcal{M}[\![\neg q]\!]) + 1 \\
&\quad = ((\mathcal{M}[\![p]\!] + 1) \cdot (\mathcal{M}[\![q]\!] + 1)) + 1 \\
&\quad = ((p+1) \cdot (q+1)) + 1 \\
&\quad = pq + p + q + 1 + 1 \\
&\quad = pq + p + q + 2
\end{aligned}$$

Since $2 \equiv 0 (\mathrm{mod}\, 2)$, we can cancel the 2 and end up with the term $pq+p+q$. Here are the tables (you might check for yourself that the entries are correct.)

a	b	$ab + a + b (\mathrm{mod}\, 2)$
1	1	1
1	0	1
0	1	1
0	0	0

a	b	$a \vee b$
T	T	T
T	F	T
F	T	T
F	F	F

So, we define the translation of disjunctions as follows.

$$\mathcal{M}[\![\phi \vee \psi]\!] = pq + p + q \qquad \text{where } \; p = \mathcal{M}[\![\phi]\!] \text{ and } q = \mathcal{M}[\![\psi]\!]$$

Implication

The following propositional formula holds.

$$(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$$

Thus, implication can be reformulated in terms of negation and disjunction. Using the translation constructed so far, we get the following

$$\begin{aligned}
&\mathcal{M}[\![\neg p \vee q]\!] \\
&= \mathcal{M}[\![\neg p]\!] \cdot \mathcal{M}[\![q]\!] + \mathcal{M}[\![\neg p]\!] + \mathcal{M}[\![q]\!] \\
&= (\mathcal{M}[\![p]\!] + 1) \cdot q + (\mathcal{M}[\![p]\!] + 1) + q \\
&= (p+1)q + (p+1) + q \\
&\quad = (pq + q + (p+1) + q \\
&\quad = (pq + 2q + (p+1)
\end{aligned}$$

Since $2q \equiv 0 (\mathrm{mod}\, 2)$, we can cancel the $2q$ term and the final formula for the translation of implication is $pq + p + 1$. And we get the following tables.

a	b	$ab + a + 1 (\mathrm{mod}\, 2)$
1	1	1
1	0	0
0	1	1
0	0	1

a	b	$a \Rightarrow b$
T	T	T
T	F	F
F	T	T
F	F	T

Thus,

$$\mathcal{M}[\![\phi \Rightarrow \psi]\!] = pq + p + 1 \qquad \text{where } p = \mathcal{M}[\![\phi]\!] \text{ and } q = \mathcal{M}[\![\psi]\!]$$

2.9.3 The Final Translation

The following function recursively translates a propositional formula into an algebraic formula.

$$\begin{aligned}
\mathcal{M}[\![\bot]\!] &= 0 \\
\mathcal{M}[\![x]\!] &= x \\
\mathcal{M}[\![\neg\phi]\!] &= \mathcal{M}[\![\phi]\!] + 1 \\
\mathcal{M}[\![\phi \wedge \psi]\!] &= (\mathcal{M}[\![\phi]\!] \cdot \mathcal{M}[\![\psi]\!]) \\
\mathcal{M}[\![\phi \vee \psi]\!] &= (\mathcal{M}[\![\phi]\!] \cdot \mathcal{M}[\![\psi]\!]) + \mathcal{M}[\![\phi]\!] + \mathcal{M}[\![\psi]\!] \\
\mathcal{M}[\![\phi \Rightarrow \psi]\!] &= (\mathcal{M}[\![\phi]\!] \cdot \mathcal{M}[\![\psi]\!]) + \mathcal{M}[\![\phi]\!] + 1
\end{aligned}$$

Example 2.6. Consider the formula $(p \vee q) \Rightarrow p$.

$$\begin{aligned}
&\mathcal{M}[\![(p \vee q) \Rightarrow p]\!] \\
&= (\mathcal{M}[\![p \vee q]\!] \cdot \mathcal{M}[\![p]\!]) + \mathcal{M}[\![p \vee q]\!] + 1 \\
&= (((\mathcal{M}[\![p]\!] \cdot \mathcal{M}[\![q]\!]) + \mathcal{M}[\![p]\!] + \mathcal{M}[\![q]\!]) \cdot p) + ((\mathcal{M}[\![p]\!] \cdot \mathcal{M}[\![q]\!]) + \mathcal{M}[\![p]\!] + \mathcal{M}[\![q]\!]) + 1 \\
&= (((p \cdot q) + p + q) \cdot p) + ((p \cdot q) + p + q) + 1 \\
&= (((pq) + p + q)p) + ((pq) + p + q) + 1 \\
&= (p^2q + p^2 + pq) + pq + p + q + 1 \\
&= pq + p + pq + pq + p + q + 1 \\
&= 2(pq) + 2p + pq + q + 1 \\
&= pq + q + 1
\end{aligned}$$

We can check this for all combinations of values for p and q. Instead, we notice that the final formula is the same as the translation for implication of $q \Rightarrow p$. To check our work we could check that:

$$((p \vee q) \Rightarrow p) \Leftrightarrow (q \Rightarrow p)$$

We have presented propositional logic syntax and have give semantics (meaning) based on truth tables over the set of truth values $\{T, F\}$. An alternative meaning can be assigned to propositional formulas by translating them into algebraic form over the natural numbers and then looking at the congruences modulo 2, *i.e.* by claiming they're *congruent* to 0 or 1 depending on whether they're even or odd.

Such an interpretation is correct if it makes all the same formulas true.

2.9.4 Modular Arithmetic

Congruence (of which modular arithmetic is one kind) is an interesting topic of discrete mathematics in its own right. We will only present enough material here to make the association between propositional logic and its algebraic interpretation.

Remark 2.2. Recall that for every integer a and every natural number $m > 0$, there exists a integers q and r where $0 \leq r < m$ such that the following equation holds:

$$a = qm + r$$

We call q the *quotient* and r the *remainder* If $r = 0$ (there is no remainder) them we say m *divides* a *e.g.* $a \div m = q$.

Definition 2.23. Two integers are *congruent* modulo 2, if and only if they have the same remainder when divided by 2. In this case we write

$$a \equiv b (\mathrm{mod}\, 2)$$

Example 2.7.

$$\begin{array}{ll} 0 \equiv 0(\mathrm{mod}\, 2) & a = 0, k = 0, r = 0 \\ 1 \equiv 1(\mathrm{mod}\, 2) & a = 1, k = 0, r = 1 \\ 2 \equiv 0(\mathrm{mod}\, 2) & a = 2, k = 1, r = 0 \\ 3 \equiv 1(\mathrm{mod}\, 2) & a = 3, k = 1, r = 1 \\ 4 \equiv 0(\mathrm{mod}\, 2) & a = 4, k = 2, r = 0 \\ 5 \equiv 1(\mathrm{mod}\, 2) & a = 5, k = 2, r = 1 \end{array}$$

Theorem 2.5. The following three properties hold.

i.) $a \equiv a(\mathrm{mod}\, 2)$
ii.) If $a \equiv b(\mathrm{mod}\, 2)$ then $b \equiv a(\mathrm{mod}\, 2)$
iii.) If $a \equiv b(\mathrm{mod}\, 2)$ and $b \equiv c(\mathrm{mod}\, 2)$ then $a \equiv c(\mathrm{mod}\, 2)$

Theorem 2.6. If $a \in \mathbb{Z}$ is even, then $a \equiv 0(\mathrm{mod}\, 2)$ and if a is odd, then $a \equiv 1(\mathrm{mod}\, 2)$.

Theorem 2.7. If $a \equiv c(\mathrm{mod}\, n)$ and $b \equiv d(\mathrm{mod}\, n)$ then

$$\begin{array}{l} a + b \equiv c + d(\mathrm{mod}\, n) \\ a \cdot b \equiv c \cdot d(\mathrm{mod}\, n) \end{array}$$

Example 2.8. Since $5 \equiv 3(\mathrm{mod}\, 2)$ and $10 \equiv 98(\mathrm{mod}\, 2)$

$$5 + 10 \equiv 3 + 98(\mathrm{mod}\, 2) \quad \text{and} \quad 5 \cdot 10 \equiv 3 \cdot 98(\mathrm{mod}\, 2)$$

To see this note the following:

$5 + 10 = 15$ and $15 = 7 \cdot 2 + 1$, so $5 + 10 \equiv 1(\mathrm{mod}\, 2)$
$3 + 98 = 101$ and $101 = 50 \cdot 2 + 1$, so $3 + 98 \equiv 1(\mathrm{mod}\, 2)$
so by properties (ii) and (iii) of Theorem 1.1
$5 + 10 \equiv 3 + 98(\mathrm{mod}\, 2)$

Prove to yourself that $5 \cdot 10 \equiv 3 \cdot 98(\mathrm{mod}\, 2)$.

Definition 2.24. We will write $n(\mathrm{mod}\,2)$ to denote the remainder of $n \div 2$. So, $5(\mathrm{mod}\,2) = 1$ and $28(\mathrm{mod}\,2) = 0$.

Theorem 2.8. The following identities hold.

$$2p \equiv 0(\mathrm{mod}\,2)$$
$$p^2 \equiv p(\mathrm{mod}\,2)$$

2.9.5 Translation from Propositional Logic

In this section we define a function that maps propositional formulas to algebraic formulas.

We start the translation with falsity ($\bot$) and conjunction. Conjunction is easily seen to correspond to multiplication. Negation is defined next, and then using DeMorgan's laws, translations for disjunction and implication are given.

Falsity

We interpret $\bot$ as 0, so the translation function maps $\bot$ to 0, no matter what the assignment is.

$$\mathcal{M}[\![\bot]\!] \quad = \quad 0$$

Variables

Propositional variables are just mapped to variables in the algebraic language

$$\mathcal{M}[\![x]\!] \quad = \quad x$$

Conjunction

Consider the following tables for multiplication and the table for conjunction.

a	b	ab
1	1	1
1	0	0
0	1	0
0	0	0

a	b	$a \wedge b$
T	T	T
T	F	F
F	T	F
F	F	F

This table is identical to the truth table for conjunction ($\wedge$) if we replace 1 by T, 0 by F and the symbol for multiplication ($\cdot$) by the symbol for conjunction ($\wedge$). Thus, we get the following translation.

$$\mathcal{M}[\![\phi \wedge \psi]\!] \quad = \quad \mathcal{M}[\![\phi]\!] \cdot \mathcal{M}[\![\psi]\!]$$

Negation

Notice that addition by 1 modulo 2 toggles values.

$$1 + 1 = 2 \text{ and } 2 \equiv 0 (\text{mod } 2) \text{ and } 0 + 1 = 1$$

The following tables show addition by 1 modulo 2 and the truth table for negation to illustrate that the translating negations to addition by 1 give the correct results.

a	$a + 1 (\text{mod } 2)$
1	0
0	1

a	$\neg a$
T	F
F	T

The translation is defined as follows:

$$\mathcal{M}[\![\neg\phi]\!] = (\mathcal{M}[\![\phi]\!] + 1)$$

Exclusive-Or

We might hope that disjunction would be properly modeled by addition ... "If wishes were horses, beggars would ride." Consider the table for addition modulo 2 and compare it with the table for disjunction – clearly they do not match.

a	b	$a + b (\text{mod } 2)$
1	1	0
1	0	1
0	1	1
0	0	0

a	b	$a \vee b$
T	T	T
T	F	T
F	T	T
F	F	F

The problem is that $1 + 1 \equiv 0 (\text{mod } 2)$ while we want that entry to be 1, *i.e.* if p and q are both T, $p \vee q$ should be T as well.

But the addition table does correspond to a useful propositional connective (one we haven't introduced so far) – *exclusive or* – often written as $(p \oplus q)$ and which is true if one of the p or q is true *but not both*. It's truth table is given as follows:

a	b	$a \oplus b$
T	T	F
T	F	T
F	T	T
F	F	F

Disjunction

We can derive disjunction using the following identity of propositional logic and the translation rules we have defined so far.

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

Exercise 2.2. Verify this identity by using a truth table.

By the translation we have so far

$$\begin{aligned}
&\mathcal{M}[\![\neg(\neg p \wedge \neg q)]\!] \\
&\quad = \mathcal{M}[\![(\neg p \wedge \neg q)]\!] + 1 \\
&\quad = (\mathcal{M}[\![\neg p]\!] \cdot \mathcal{M}[\![\neg q]\!]) + 1 \\
&\quad = ((\mathcal{M}[\![p]\!] + 1) \cdot (\mathcal{M}[\![q]\!] + 1)) + 1 \\
&\quad = ((p+1) \cdot (q+1)) + 1 \\
&\quad = pq + p + q + 1 + 1 \\
&\quad = pq + p + q + 2
\end{aligned}$$

Since $2 \equiv 0 (\mathrm{mod}\, 2)$, we can cancel the 2 and end up with the term $pq+p+q$. Here are the tables (you might check for yourself that the entries are correct.)

a	b	$ab + a + b (\mathrm{mod}\, 2)$
1	1	1
1	0	1
0	1	1
0	0	0

a	b	$a \vee b$
T	T	T
T	F	T
F	T	T
F	F	F

So, we define the translation of disjunctions as follows.

$$\mathcal{M}[\![\phi \vee \psi]\!] = pq + p + q \qquad \text{where } \; p = \mathcal{M}[\![\phi]\!] \text{ and } q = \mathcal{M}[\![\psi]\!]$$

Implication

The following propositional formula holds.

$$(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$$

Thus, implication can be reformulated in terms of negation and disjunction. Using the translation constructed so far, we get the following

$$\begin{aligned}
&\mathcal{M}[\![\neg p \vee q]\!] \\
&= \mathcal{M}[\![\neg p]\!] \cdot \mathcal{M}[\![q]\!] + \mathcal{M}[\![\neg p]\!] + \mathcal{M}[\![q]\!] \\
&= (\mathcal{M}[\![p]\!] + 1) \cdot q + (\mathcal{M}[\![p]\!] + 1) + q \\
&= (p+1)q + (p+1) + q \\
&\quad = (pq + q + (p+1) + q \\
&\quad = (pq + 2q + (p+1)
\end{aligned}$$

Since $2q \equiv 0 (\mathrm{mod}\, 2)$, we can cancel the $2q$ term and the final formula for the translation of implication is $pq + p + 1$. And we get the following tables.

a	b	$ab + a + 1 (\mathrm{mod}\, 2)$
1	1	1
1	0	0
0	1	1
0	0	1

a	b	$a \Rightarrow b$
T	T	T
T	F	F
F	T	T
F	F	T

Thus,

$$\mathcal{M}[\![\phi \Rightarrow \psi]\!] = pq + p + 1 \qquad \text{where } p = \mathcal{M}[\![\phi]\!] \text{ and } q = \mathcal{M}[\![\psi]\!]$$

2.9.6 The Final Translation

The following function recursively translates a propositional formula into an algebraic formula.

$$\begin{array}{rcl}
\mathcal{M}[\![\bot]\!] & = & 0 \\
\mathcal{M}[\![x]\!] & = & x \\
\mathcal{M}[\![\neg\phi]\!] & = & \mathcal{M}[\![\phi]\!] + 1 \\
\mathcal{M}[\![\phi \wedge \psi]\!] & = & (\mathcal{M}[\![\phi]\!] \cdot \mathcal{M}[\![\psi]\!]) \\
\mathcal{M}[\![\phi \vee \psi]\!] & = & (\mathcal{M}[\![\phi]\!] \cdot \mathcal{M}[\![\psi]\!]) + \mathcal{M}[\![\phi]\!] + \mathcal{M}[\![\psi]\!] \\
\mathcal{M}[\![\phi \Rightarrow \psi]\!] & = & (\mathcal{M}[\![\phi]\!] \cdot \mathcal{M}[\![\psi]\!]) + \mathcal{M}[\![\phi]\!] + 1
\end{array}$$

Example 2.9. Consider the formula $(p \vee q) \Rightarrow p$.

$$\begin{array}{l}
\mathcal{M}[\![(p \vee q) \Rightarrow p]\!] \\
= (\mathcal{M}[\![p \vee q]\!] \cdot \mathcal{M}[\![p]\!]) + \mathcal{M}[\![p \vee q]\!] + 1 \\
= (((\mathcal{M}[\![p]\!] \cdot \mathcal{M}[\![q]\!]) + \mathcal{M}[\![p]\!] + \mathcal{M}[\![q]\!]) \cdot p) + ((\mathcal{M}[\![p]\!] \cdot \mathcal{M}[\![q]\!]) + \mathcal{M}[\![p]\!] + \mathcal{M}[\![q]\!]) + 1 \\
= (((p \cdot q) + p + q) \cdot p) + ((p \cdot q) + p + q) + 1 \\
= (((pq) + p + q)p) + ((pq) + p + q) + 1 \\
= (p^2q + p^2 + pq) + pq + p + q + 1 \\
= pq + p + pq + pq + p + q + 1 \\
= 2(pq) + 2p + pq + q + 1 \\
= pq + q + 1
\end{array}$$

We can check this for all combinations of values for p and q. Instead, we notice that the final formula is the same as the translation for implication of $q \Rightarrow p$. To check our work we could check that:

$$((p \vee q) \Rightarrow p) \Leftrightarrow (q \Rightarrow p)$$

Chapter 3

Predicate Logic

In this section we extend propositional logic presented in the previous chapter to allow for *quantification* of the form:

> *for all things* x, $\cdots$
> *for every* x, $\cdots$
> *there exists a thing* x *such that* $\cdots$
> *for some thing* x, $\cdots$

Where "$\cdots$" is some statement referring to the thing denoted by the variable x that specifies a property of the thing denoted by x. The first two forms are called *universal quantifications*, they are different ways of asserting that everything satisfies some specified property. The second two forms are called *existential quantifications*, they assert that something exists having the specified property.

Symbolically, we write "*for all things* x, $\cdots$ " as $(\forall x. \cdots)$ and "*there exists a thing* x *such that* $\cdots$" as $(\exists x. \cdots)$.

3.1 Predicates

To make this extension to our logic we add truth-valued functions called predicates which map elements from a *domain of discourse* to the values in $\mathbb{B}$.

Definition 3.1. A function is *n-ary* if it takes n arguments, $0 \leq n$. If a function is $n-ary$, we say it has *arity* n. A function of arity 0, *i.e.* a function that takes no arguments, is called a *constant*. We say a 0-ary function is *nullary*, 1-ary function is *unary*. We say a 2-ary function is *binary* and, although we could say 3-ary, 4-ary and 5-ary functions ternary, quaternary and quintary respectively, we do not insist on carrying this increasingly tortured nomenclature any further.

For example, consider the following functions:

i.) $f() = 5$
ii.) $g(x) = x + 5$
iii.) $h(x, y) = (x + y) - 1$
vi.) $f_1(x, y, z) = x * (y + z)$
v.) $g_1(x, y, z, w) = f_1(x, y, w) - z$

The first function is nullary, it takes *no* arguments. Typically, we will drop the parentheses and write f instead of $f()$. The second function takes one argument and so is a *unary function*. The third function is *binary*. The fourth and fifth are 3-ary and 4-ary functions respectively.

Definition 3.2. A function is *Boolean-valued* if its range is the set $\mathbb{B}$.

Definition 3.3. A *predicate* is a n-ary Boolean-valued function over some domain of input.

Example 3.1. In ordinary arithmetic, the binary predicates include *less than* (written $<$) and *equals* (written $=$). Typically these are written in infix notation *i.e.* instead of writing $=(x, y)$ and $<(x, y)$ we write $x = y$ and $x < y$; do not let this infix confuse you, they are still binary predicates. We can define other predicates in terms of these two. For example we can define a binary predicate *less-than-or-equals* as:

$$i \leq j \stackrel{\text{def}}{=} ((i = j) \vee (i < j))$$

We could define a unary predicate which is true when its argument is equal to 0 and is false otherwise:

$$=_0 (i) \stackrel{\text{def}}{=} i = 0$$

We could define a 3-ary predicate which is true if k is strictly between i and j:

$$\texttt{between}(i, j, k) \stackrel{\text{def}}{=} ((i < k) \wedge (k < j))$$

Note that predicate constants act just like propositional variables.

3.2 The Syntax of Predicate Logic

Predicate logic formulas are constructed from two sorts of components:

i.) parts that refer to objects and functions on those objects in the domain of discourse. These components of the formula are called *terms*.

ii.) parts of a formula that denote truth values, these include predicates over the domain of discourse and formulas constructed inductively by connecting previously constructed formulas.

3.2.1 Variables

The definitions of the syntactic classes of terms and formulas (both defined below) depend on an unbounded collection of variable symbols, we call this set $\mathcal{V}$.

$$\mathcal{V} = \{x, y, z, w, x_1, y_1, z_1, w_1, x_2, \cdots\}$$

Unlike propositional variables, which denoted truth-values, these variables will range over individual elements in the domain of discourse. Like propositional variables, we assume the set $\mathcal{V}$ is fixed (and so we do not include it among the parameterize of the definitions that use it.)

3.2.2 Terms

The syntax of terms (the collection of which we will write as $\mathcal{T}$) is determined by a set of n-ary function symbols, call this set $\mathcal{F}$. We assume the arity of a function symbol can be determined.

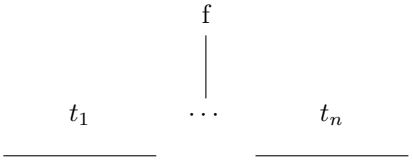
Definition 3.4. *Terms* defined over a set of function symbols $\mathcal{F}$ are given by the following grammar:

$$\mathcal{T}_{[\mathcal{F}]} ::= x \mid f(t_1, \cdots t_n)$$

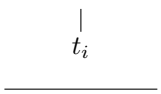
where:

$\mathcal{F}$ is set of function symbols,
$x \in \mathcal{V}$ is a variable,
$f \in \mathcal{F}$ is a function symbol for a function of arity n, where $n \geq 0$ and
$t_i \in \mathcal{T}_{[\mathcal{F}]}$ denote previously constructed terms, $1 \leq i \leq n$.

The syntax tree for a function application might appear as follows.



where the figure displayed as:



denotes the syntax tree for the term t_i.

Note that the definition of terms is parameterized by the set of function symbols. The set of terms in $\mathcal{T}_{[\mathcal{F}]}$ is determined by the set of function symbols in $\mathcal{F}$ and by the arities of those symbols. Also, note that if $n = 0$, the term $f()$ is a constant and we will write it simply as f.

Example 3.2. Let $\mathcal{F} = \{a, b, f, g\}$ where a and b are constants, f is a unary function symbol and g is a binary function symbol. In this case, $\mathcal{T}$ includes:

$$\begin{array}{l}\{a, x, f(a), f(x),\\ g(a,a), g(a,x), g(a,f(a)), g(a,f(x)),\\ g(x,a), g(x,x), g(x,f(a)), g(x,f(x)),\\ b, y, f(b), f(y), f(f(a)), f(f(x)), f(g(a,a)), \cdots\end{array}$$

3.2.3 Formulas

Definition 3.5. *Formulas* of predicate logic are defined over a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$ and are given by the following grammar.

$$\mathcal{PL}_{[\mathcal{F},\mathcal{P}]} ::= \bot \mid P(t_1, \cdots, t_n) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \mid \forall x.\phi \mid \exists x.\phi$$

where:

$\mathcal{F}$ is set of function symbols,

$\mathcal{P}$ is set of predicate symbols,

$\bot$ is a constant symbol,

$P \in \mathcal{P}$ is a predicate symbol for a predicate of arity n, where $n \geq 0$,

$t_i \in \mathcal{T}_{[\mathcal{F}]}$ are terms, $1 \leq i \leq n$,

$\phi, \psi \in \mathcal{PL}_{[\mathcal{F},\mathcal{P}]}$ are previously constructed formulas, and

$x \in \mathcal{V}$ is a variable.

This definition is parameterized by the set of function symbols ($\mathcal{F}$) and the set of predicate symbols ($\mathcal{P}$). As remarked above, a predicate symbol denoting a constant is the equivalent of a propositional variable presented in the previous chapter. Thus, predicate symbols are a generalization of propositional variables; when actual values are substituted for their variables, they denote truth values.

Predicate Logic extends Propositional Logic

Given a rich enough set of predicate symbols $\mathcal{P}$ *i.e.* one that includes one constant symbol for each propositional variable, the language of predicate logic extends the language of propositional logic. Specifically, every formula of propositional logic is a formula of predicate logic. To see this note that: the constant symbol bottom ($\bot$) is included in both languages; the propositional variables are all included in $\mathcal{P}$ as predicate symbols of arity 0. Also, every connective of propositional logic is also a connective of predicate logic. Thus, we conclude that every formula of propositional logic can be identified with a syntactically identical formula of predicate logic.

We will see in later sections that not only is the syntax preserved, both the semantics and the proof system are also preserved.

Some Examples

In the following examples we show uses of the quantifiers to formally encode some theorems of arithmetic.

Example 3.3. The *law of trichotomy* in the language of arithmetic says:

> *For all integers i and j, either: i is less than j or i is equal to j or j is less than i.*

We can formalize this statement, making explicit that *less-than* and *equals* are binary predicates by writing them as $\texttt{lt}(i, j)$ and $\texttt{eq}(i, j)$ respectively:

$$\forall i.\forall j.(\texttt{lt}(i, j) \vee (\texttt{eq}(i, j) \vee \texttt{lt}(j, i)))$$

We can rewrite the same statement as follows using the ordinary notation of arithmetic (which perhaps makes the fact that *less-than* and *equals* are predicates less obvious.)

$$\forall i.\forall j.(i < j \vee (i = j \vee j < i))$$

Example 3.4. As another example in the natural numbers.

> *For every natural number i either: $i = 0$ and there is no number less that i, or there exists a natural number j such that $j < i$.*

We can formalize this statement as

$$\forall i.(i = 0 \wedge \forall j.\neg(j < i)) \vee (\exists j.j < i))$$

Note that if the domain of discourse (the set from which the variables i and j take their values) is the natural numbers, the statement is a theorem but it is false if the domain of discourse is the integers or reals.

Example 3.5. The *Schröder-Bernstein theorem* for numbers is as follows.

> *For all integers n and m, if $n \leq m \leq n$ then $n = m$.*

This is formalized as follows:

$$\forall n.\forall m.(n \leq m \wedge m \leq n) \Rightarrow n = m$$

Note that the commonly used notation $(i \leq j \leq k)$ means $((i \leq j) \wedge (j \leq k))$.

Example 3.6. Consider the following statement:

For every natural number n which is greater than 1, either n is a prime number or there are two integers, both greater than 1 and less than n, whose product is n.

Let P be a unary predicate that is true if and only if its single argument is a prime number. Let `mul` be a binary function symbol denoting multiplication. Then we formalize this statement as follows:

$$\forall n.n > 1 \Rightarrow (P(n) \vee \exists i.\exists j.\texttt{between}(1, n, i) \wedge \texttt{between}(1, n, j) \wedge \texttt{mul}(i, j) = n)$$

We can rewrite this using standard mathematical notion as follows:

$$\forall n.n > 1 \Rightarrow (P(n) \vee \exists i.\exists j.(1 < i \wedge i < n) \wedge (1 < j \wedge j < n) \wedge i \cdot j = n)$$

Remark 3.1. The fact that these statements are true when the symbols are interpreted in the ordinary way we think of numbers is a fact that is external to logic. The predicates *less-than* and *equals* are particular predicates that have particular values when interpreted in ordinary arithmetic. If we swapped the interpretations of the symbols (*i.e.* if we interpreted $i < j$ to be true whenever i and j are equal numbers and interpreted $i < j$ to be false otherwise; and similarly interpreted $i = j$ to be true whenever i was less than j and false otherwise) we would still have well formed formulas in the language of arithmetic, but would interpret the meanings of the predicates differently. So the interpreted meaning of the predicate symbols and function symbols *may* have a bearing on the truth or falsity of a formula. We discuss this later in the section on semantics. Note that there are also formulas of predicate logic which do not depend on the meanings of the predicate and function symbols, we will give examples of such formulas in a later section.

3.2.4 Bindings and Variable Occurrences

Variable Occurrences

Definition 3.6. A variable x *occurs in term* t if and only if $x \in \texttt{occurs}(t)$ where `occurs` is defined as follows:

$$\begin{aligned} \texttt{occurs}(z) &= \{z\} \\ \texttt{occurs}(f(t_1, \cdots, t_n)) &= \bigcup_{i=1}^{n} \texttt{occurs}(t_i) \end{aligned}$$

Thus, the variable z occurs in a term which is simply a variable of the form z. Otherwise, a variable occurs in a term of the form $f(t_1, \cdots, t_n)$ if and only if it occurs in one of the terms $t_i, 1 \leq i \leq n$. To collect them, we simply union [1] all the sets of variables occurring in each t_i.

[1] If n = 0, (*i.e.* if the arity of the function symbol is 0) then $\bigcup_{i=1}^{0} \texttt{occurs}(t_i) = \{\}$

Definition 3.7. A variable x *occurs in formula* ϕ if and only if $x \in \texttt{occurs}(\phi)$ where `occurs` is defined by recursion on the structure of ϕ as follows.

$$\begin{aligned}
\texttt{occurs}(P(t_1, \cdots, t_n)) &= \textstyle\bigcup_{i=1}^{n} \texttt{occurs}(t_i) \\
\texttt{occurs}(\neg\phi) &= \texttt{occurs}(\phi) \\
\texttt{occurs}(\phi \wedge \psi) &= \texttt{occurs}(\phi) \cup \texttt{occurs}(\psi) \\
\texttt{occurs}(\phi \vee \psi) &= \texttt{occurs}(\phi) \cup \texttt{occurs}(\psi) \\
\texttt{occurs}(\phi \Rightarrow \psi) &= \texttt{occurs}(\phi) \cup \texttt{occurs}(\psi) \\
\texttt{occurs}(\forall z.\phi) &= \texttt{occurs}(\phi) \cup \{z\} \\
\texttt{occurs}(\exists z.\phi) &= \texttt{occurs}(\phi) \cup \{z\}
\end{aligned}$$

Thus, a variable x occurs in a formula of the form $P(t_1, \cdots, t_n)$ if and only if x occurs in one of the terms $t_i, 1 \leq i \leq n$. The variable x occurs in $\neg\phi$ iff it occurs in ϕ. Similarly, it occurs in $\phi \wedge \psi$, $\phi \vee \psi$, and $\phi \Rightarrow \psi$ iff it occurs in ϕ or it occurs in ψ. The variable x occurs in $\forall x.\phi$ and $\exists x.\phi$ regardless of whether it occurs in ϕ.

Definition 3.8. In formulas of the form $\forall x.\phi$ and $\exists x.\phi$: the quantifier symbols "$\forall$" and "$\exists$" are *binding operators*, the occurrence of the variable x just after the quantifier is called the *binding occurrence*. The partial syntax tree of the formula ϕ, where sub-trees corresponding to sub-formulas of the form $\forall x.\psi$ and $\exists x.\psi$ have been removed, is called the *scope of the binding*. In the linear notation of formulas, we mark the missing sub-formulas by replacing them with the symbol "$\Box$."

The idea of variable scope is a familiar one to programmers. In programming languages, the scope of a variable declaration specifies what part of the program text refers to which variable declaration. Differer languages have different scoping rules, but the modern standard of *lexical scoping* (or *local scoping*) essentially follow the rules given for logic. These are very close to the rules used in C++ for example [7].

Example 3.7. The scope of the leftmost binding occurrence of the variable x in the formula

$$\forall x.(P(x) \wedge \exists x.Q(x, y)) \qquad \text{is} \qquad (P(x) \wedge \Box)$$

Where, $\Box$ blocks out the part of the formula not in the scope of the first binding occurrence of x. The scope of the rightmost binding occurrence of x in the same formula is $Q(x, y)$.

Free Variables

Definition 3.9. A variable x *occurs free in term* t if and only if $x \in FV(t)$ where FV is defined as follows:

$$\begin{aligned}
\texttt{FV}(z) &= \{z\} \\
\texttt{FV}(f(t_1, \cdots, t_n)) &= \textstyle\bigcup_{i=1}^{n} \texttt{FV}(t_i)
\end{aligned}$$

Thus, a variable x occurs free in a term which is simply a variable of the form z if and only if $x = z$. Otherwise, x occurs in a term of the form $f(t_1, \cdots, t_n)$ if and only if x occurs in one of the terms $t_i, 1 \leq i \leq n$.

Definition 3.10. A variable x *occurs free in formula* ϕ if and only if $x \in \mathtt{FV}(\phi)$ where:

$$\begin{array}{rcl} \mathtt{FV}(P(t_1, \cdots, t_n)) & = & \bigcup_{i=1}^{n} \mathtt{FV}(t_i) \\ \mathtt{FV}(\neg\phi) & = & \mathtt{FV}(\phi) \\ \mathtt{FV}(\phi \wedge \psi) & = & \mathtt{FV}(\phi) \cup \mathtt{FV}(\psi) \\ \mathtt{FV}(\phi \vee \psi) & = & \mathtt{FV}(\phi) \cup \mathtt{FV}(\psi) \\ \mathtt{FV}(\phi \Rightarrow \psi) & = & \mathtt{FV}(\phi) \cup \mathtt{FV}(\psi) \\ \mathtt{FV}(\forall z.\phi) & = & \mathtt{FV}(\phi) - \{z\} \\ \mathtt{FV}(\exists z.\phi) & = & \mathtt{FV}(\phi) - \{z\} \end{array}$$

Thus, a variable x occurs free in a formula iff it occurs in the formula and it is not in the scope of any binding of the variable x.

Bound Variables

Definition 3.11. A variable x *occurs bound in formula* ϕ if and only if $x \in \mathtt{BV}(\phi)$ where BV is defined as follows:

$$\begin{array}{rcl} \mathtt{BV}(P(t_1, \cdots, t_n)) & = & \{\} \\ \mathtt{BV}(\neg\phi) & = & \mathtt{BV}(\phi) \\ \mathtt{BV}(\phi \wedge \psi) & = & \mathtt{BV}(\phi) \cup \mathtt{BV}(\psi) \\ \mathtt{BV}(\phi \vee \psi) & = & \mathtt{BV}(\phi) \cup \mathtt{BV}(\psi) \\ \mathtt{BV}(\phi \Rightarrow \psi) & = & \mathtt{BV}(\phi) \cup \mathtt{BV}(\psi) \\ \mathtt{BV}(\forall z.\phi) & = & \mathtt{BV}(\phi) \cup \{z\} \\ \mathtt{BV}(\exists z.\phi) & = & \mathtt{BV}(\phi) \cup \{z\} \end{array}$$

Thus, a variable x occurs bound in a formula iff it contains a sub-formula of the form $\forall x.\phi$ or $\exists x.\phi$.

Note that there are formulas where x occurs free *and* x occurs bound, *e.g.* consider the formula $P(x) \wedge \forall x.P(x)$. The first occurrence of x is free and the second and third occurrences of x are bound.

3.2.5 Capture Avoiding Substitution

Substitution is perhaps the most basic operation in mathematics and it is usually performed without mention. But to actually specify capture avoiding substitution correctly reveals a sad history of error. Hilbert got it wrong in the first edition of his logic book with Ackermann [20], Quine got it wrong in the first edition of his logic book [28], and almost every automated theorem prover in existence has experienced bugs in their implementations of substitution at some time. Capture avoiding substitution is hard to get right.

More evidence for the pivotal role substitution plays: the only computation mechanism in Church's lambda calculus [4] is substitution, and anything we

currently count as algorithmically computable can be computed by a term of the lambda calculus.

For terms, there are no binding operators so capture avoiding substitution is just ordinary substitution.

Definition 3.12. *Capture avoiding substitution for terms* is defined as follows:

$$\begin{aligned} x[x := t] &= t \\ z[x := t] &= z \qquad \text{if } (x \neq z) \\ f(t_1, \cdots, t_n)[x := t] &= f(t_1[x := t], \cdots t_n[x := t]) \end{aligned}$$

The first clause of definition says that if you are trying to substitute the term t for free occurrences of the variable x in the term that consists of the single variable x, then go ahead and do it – *i.e.* replace x by t and that is the result of the substitution.

The second clause of the definition says that if you're looking to substitute t for x, but you're looking at a variable z where z is different from x, do nothing – the result of the substitution is just the variable z.

The third clause of the definition follows a standard pattern of recursion. The result of substituting t for free occurrences of x in the term $f(t_1, \cdots, t_n)$, is the term obtained by substituting t for x in each of the n arguments $t_i, 1 \leq i \leq n$, and then returning the term assembled from these parts by placing the substituted argument terms in the appropriate places.

Note that substitution of term t for free occurrences of the variable x can *never* affect a function symbol (f) since function symbols are not variables.

Definition 3.13. *Capture avoiding substitution for formulas* is defined as follows:

$$
\begin{array}{rcl}
\bot[x := t] & = & \bot \\
P(t_1, \cdots, t_n)[x := t] & = & P(t_1[x := t], \cdots, t_n[x := t]) \\
(\neg\phi)[x := t] & = & \neg(\phi[x := t]) \\
(\phi \wedge \psi)[x := t] & = & (\phi[x := t] \wedge \psi[x := t]) \\
(\phi \vee \psi)[x := t] & = & (\phi[x := t] \vee \psi[x := t]) \\
(\phi \Rightarrow \psi)[x := t] & = & (\phi[x := t] \Rightarrow \psi[x := t]) \\
(\forall x.\phi)[x := t] & = & (\forall x.\phi) \\
(\forall y.\phi)[x := t] & = & (\forall y.\phi[x := t]) \\
 & & \text{if } (x \neq y, y \notin FV(t)) \\
(\forall y.\phi)[x := t] & = & (\forall z.\phi[y := z][x := t]) \\
 & & \text{if } (x \neq y, y \in FV(t), z \notin (FV(t) \cup FV(\phi) \cup \{x\})) \\
(\exists x.\phi)[x := t] & = & (\exists x.\phi) \\
(\exists y.\phi)[x := t] & = & (\exists y.\phi[x := t]) \\
 & & \text{if } (x \neq y, y \notin FV(t)) \\
(\exists y.\phi)[x := t] & = & (\exists z.\phi[y := z][x := t]) \\
 & & \text{if } (x \neq y, y \in FV(t), z \notin (FV(t) \cup FV(\phi) \cup \{x\}))
\end{array}
$$

3.3 Rules for Quantifiers

3.3.1 Universal Quantifier Rules

On the right

If we have a formula with the principle constructor $\forall$ (say $\forall x.\phi$) on the right of a sequent then it is enough to prove the sequent where $\forall x.\phi$ has been replaced by the formula $\phi[x := y]$, where y is a new variable not occurring free in any formula of the sequent. Choosing a new variable not occuring free anywhere in the sequent, to replace the bound variable x has the effect of selecting an arbitrary element from the domain of discourse *i.e.* by choosing a completely new variable, we know nothing about it — *except* that it stands for some element of the domain of discourse.

$$\frac{\Gamma \vdash \Delta_1, \phi[x := y], \Delta_2}{\Gamma \vdash \Delta_1, \forall x.\phi, \Delta_2}\ (\forall\text{R}) \qquad \begin{array}{l}\text{where variable } y \text{ is not free in any} \\ \text{formula of } (\Gamma \cup \Delta_1 \cup \{\forall x.\phi\} \cup \Delta_2).\end{array}$$

Since y is not free in any formula of the sequent, y represents an arbitrary element of the domain of discourse.

On the left

The rule for a $\forall$ on the left says, to prove a sequent with a $\forall$ occurring as the principle connective of a formula on the left side (say $\forall x.\phi$) it is enough to prove

the sequent obtained by replacing $\forall x.\phi$ by the formula $\phi[x := t]$ where t is any term [2].

$$\frac{\Gamma_1, \phi[x := t], \Gamma_2 \vdash \Delta}{\Gamma_1, \forall x.\phi, \Gamma_2 \vdash \Delta} \ (\forall\text{L}) \qquad \text{where } t \in \mathcal{T}.$$

We justify this by noting that if we assume $\forall x.\phi$ (this is what it means to be on the left) then it must be the case that $\phi[x := t]$ is true for any term t what-so-ever.

3.3.2 Existential Quantifier Rules

On the right

To prove a formula of the form $\exists x.\phi$ it is enough to find a term t such that $\phi[x := t]$ can be proved.

$$\frac{\Gamma \vdash \Delta_1, \phi[x := t], \Delta_2}{\Gamma \vdash \Delta_1, \exists x.\phi, \Delta_2} \ (\exists\text{R}) \qquad \text{where } t \in \mathcal{T}.$$

Note that the choice of t may require some creative thought.

Definition 3.14. The term t substituted for the bound variables in an $\exists$R-rule is called the *witness.*.

On the left

The rule for a $\exists$ on the left says, to prove a sequent with a $\exists$ occurring as the principle connective of a formula on the left side, it is enough to prove the sequent obtained by replacing the bound variable of the forall by an arbitrary variable y where y is not free in any formula of the sequent.

$$\frac{\Gamma_1, \phi[x := y], \Gamma_2 \vdash \Delta}{\Gamma_1, \exists x.\phi, \Gamma_2 \vdash \Delta} \ (\exists\text{L}) \qquad \begin{array}{l}\text{where variable } y \text{ is not free in any} \\ \text{formula of } (\Gamma_1 \cup \Gamma_2 \cup \{\exists x.\phi\} \cup \Delta).\end{array}$$

Since we know $\exists x.\phi$, we know something (call it y) exists which satisfies $\phi[x := y]$, but we can not assume anything about y other than that it has been arbitrarily chosen from the domain of discourse.

3.4 Proofs

The mechanisms for checking whether a labeled tree sequents is a proof is the same here as presented in Chap. 2 on propositional logic. But in the presence of

[2] If you further constrain that the only variables you use to construct t are among the free variables occuring in the sequent, then your proof is valid in every domain of discourse, including the empty one. Logics allowing the empty domain of discourse are called Free Logics [**?**] .

quantifiers, finding proofs is no longer a strictly mechanical process. Creativity may be required.

Example 3.8. Consider the sequent $\vdash (\forall x.P(x)) \Rightarrow (\exists y.P(y))$. Surely if everything satisfies property P, then something satisfies property P.

Initially, the only rule that applies is the propositional $\Rightarrow$R-rule. It matches this sequent by the following substitution:

$$\sigma_1 = \begin{cases} \Gamma := [\,] \\ \Delta_1 := [\,] \\ \Delta_2 := [\,] \\ \phi := (\forall x.P(x)) \\ \psi := (\exists y.P(y)) \end{cases}$$

The result of applying this substitution to the to premise of the $\Rightarrow$R-rule results in the partial proof tree of the following form:

$$\frac{\forall x.P(x) \vdash \exists y.P(y)}{\vdash \forall x.P(x) \Rightarrow \exists y.P(y)}\ (\Rightarrow\text{R})$$

Now, to continue developing the incomplete branch, we examine the $\forall$L-rule and the $\exists$R-rule. Both require the prover to select a term to substitute into the scope of the bound variable. In this case, any term will do, as long as we use the same one on both sides. All variables are terms, so just use z, and we arbitrarily choose to apply the $\forall$L-rule first.

The match of the sequent against the goal of the rule is given by the substitution:

$$\sigma_2 = \begin{cases} \Gamma := [\,] \\ \Delta_1 := [\,] \\ \Delta := [\exists y.P(y)] \\ \phi := P(x) \\ x := x \\ t := z \end{cases}$$

The term t we have chosen is the variable z. Applying the substitution to the premise of the rule results in the sequent $P(x)[x := z] \vdash \exists y.P(y)$. Note that $P(x)[x := z] = P(z)$, thus the resulting partial proof is:

$$\frac{\dfrac{P(z) \vdash \exists y.P(y)}{\forall x.P(x) \vdash \exists y.P(y)}\ (\forall\text{L})}{\vdash \forall x.P(x) \Rightarrow \exists y.P(y)}\ (\Rightarrow\text{R})$$

Now, the only rule that applies is the $\exists$R-rule. We choose t to be z and match by the following substitution.

$$\sigma_3 = \begin{cases} \Gamma := [P(z)] \\ \Delta_1 := [\,] \\ \Delta_2 := [\,] \\ \phi := P(y) \\ x := y \\ t := z \end{cases}$$

The partial proof generated by applying this rule with this substitution is as follows:

$$\dfrac{\dfrac{\dfrac{P(z) \vdash P(z)}{P(z) \vdash \exists y.P(y)}\,(\exists\mathrm{R})}{\forall x.P(x) \vdash \exists y.P(y)}\,(\forall\mathrm{L})}{\vdash \forall x.P(x) \Rightarrow \exists y.P(y)}\,(\Rightarrow\mathrm{R})$$

Now, the incomplete branch of the proof is an instance of an axiom, where the substitution verifying the match is given as follows:

$$\sigma_4 = \begin{cases} \Gamma_1 := [\,] \\ \Gamma_2 := [\,] \\ \Delta_1 := [\,] \\ \Delta_2 := [\,] \\ \phi := P(z) \end{cases}$$

Finally, we have the following complete proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P(z) \vdash P(z)}\,(\mathrm{Ax})}{P(z) \vdash \exists y.P(y)}\,(\exists\mathrm{R})}{\forall x.P(x) \vdash \exists y.P(y)}\,(\forall\mathrm{L})}{\vdash \forall x.P(x) \Rightarrow \exists y.P(y)}\,(\Rightarrow\mathrm{R})$$

3.5 Translating Sequent Proofs into English

Gentzen devised the sequent proof system to reflect how proofs are done in ordinary mathematics. The formal sequent proof is a tree structure and we could easily write and algorithm that would recursively translate sequent proofs into English. The rules for such a transformation are given in the following sections.

Axiom Rule

The rule is:

$$\dfrac{}{\Gamma_1, \phi, \Gamma_2 \vdash \Delta_1, \phi, \Delta_2}\,(\mathrm{Ax})$$

We say: "But we know ϕ is true since we have assumed it." or "ϕ holds since we assumed ϕ to be true and so we are done."

The other axiom is formally given as follows:

$$\frac{}{\Gamma_1, \bot, \Gamma_2 \vdash \Delta} \ (\bot\text{Ax})$$

We say: "But now we have assumed false and the theorem is true." or "But now, we have derived a contradiction and the theorem is true."

Conjunction Rules

The rule on the right is:

$$\frac{\Gamma \vdash \Delta_1, \phi, \Delta_2 \qquad \Gamma \vdash \Delta_1, \psi, \Delta_2}{\Gamma \vdash \Delta_1, (\phi \wedge \psi), \Delta_2} \ (\wedge\text{R})$$

We say: "To show $\phi \wedge \psi$ there are two cases, (case 1.) *insert translated proof of the left branch here* (case 2.) *insert translated proof of the right branch here.*."

Or we say: "To show $\phi \wedge \psi$ we must show ϕ and we must show ψ. To see that ϕ holds: *insert translated proof of left branch here* This completes the proof of ϕ. To see that ψ holds: *insert translated proof of right branch here.* This completes the proof of $\phi \wedge \psi$."

The rule on the left says:

$$\frac{\Gamma_1, \phi, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, (\phi \wedge \psi), \Gamma_2 \vdash \Delta} \ (\wedge\text{L})$$

We say: "Since we have assumed $\phi \wedge \psi$, we assume ϕ and we assume ψ. *Insert translated proof of the premise here.*"

Disjunction

The formal rule for a disjunction on the right is:

$$\frac{\Gamma \vdash \Delta_1, \phi, \psi, \Delta_2}{\Gamma, \vdash \Delta_1, (\phi \vee \psi), \Delta_2} \ (\vee\text{R})$$

We say: "To show $\phi \vee \psi$ we must either show ϕ or show ψ. *Insert translated proof of the premise here.*"

The sequent proof rule for disjunction on the left is:

$$\frac{\Gamma_1, \phi, \Gamma_2 \vdash \Delta \qquad \Gamma_1, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, (\phi \vee \psi), \Gamma_2 \vdash \Delta} \ (\vee\text{L})$$

We say: "Since we know $\phi \vee \psi$ we proceed by cases: suppose ϕ is true, then *insert translated proof from the left branch here.* On the other hand, if ψ holds: *insert translated proof from right branch here.*

or, we say: "Since $\phi \vee \psi$ holds, we proceed consider the two cases: (case 1, ϕ holds:) *insert translated proof from the left branch here.* (case 2. ψ holds:) *insert translated proof from right branch here.*

Implication Rules

The formal rule for an implication on the right is:

$$\frac{\Gamma, \phi \vdash \Delta_1, \psi, \Delta_2}{\Gamma \vdash \Delta_1, (\phi \Rightarrow \psi), \Delta_2} \ (\Rightarrow\text{R})$$

We say: "To prove $\phi \Rightarrow \psi$, assume ϕ and show ψ, *insert translated proof of the subgoal here.*."

The formal rule for an implication on the left is:

$$\frac{\Gamma_1, \Gamma_2 \vdash \phi, \Delta \quad \Gamma_1, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, (\phi \Rightarrow \psi), \Gamma_2 \vdash \Delta} \ (\Rightarrow\text{L})$$

We say: "Since we have assumed $\phi \Rightarrow \psi$, we show ϕ and assume ϕ. To see that ϕ holds: *insert translated proof of left branch here.* Now, we assume ψ. *Insert translated proof of right branch here.*"

Negation

The formal rule for a negation on the right is:

$$\frac{\Gamma, \phi \vdash \Delta_1, \Delta_2}{\Gamma \vdash \Delta_1, \neg\phi, \Delta_2} \ (\neg\text{R})$$

We say: "Assume ϕ. *Insert translated proof of premise here.*" or we say "Since we must show $\neg\phi$, assume ϕ. *Insert translated proof of premise here.*"

The formal rule for a negation on the left is:

$$\frac{\Gamma_1, \neg\phi, \Gamma_2 \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash \phi, \Delta} \ (\neg\text{L})$$

We say: "Since we have assumed $\neg\phi$, we show ϕ. *Insert translated proof of premise here.*" or, we say: "Since we know $\neg\phi$, we prove ϕ. *Insert translated proof of the premise here.*"

Universal Quantifier

The formal rule for a $\forall$ on the right is:

$$\frac{\Gamma \vdash \Delta_1, \phi[x := y], \Delta_2}{\Gamma \vdash \Delta_1, \forall x.\phi, \Delta_2} \ (\forall\text{R}) \qquad \begin{array}{l}\text{where variable } y \text{ is not free in any} \\ \text{formula of } (\Gamma \cup \Delta_1 \cup \{\forall x.\phi\} \cup \Delta_2).\end{array}$$

We say: "To prove $\forall x.\phi$, pick an arbitrary y and show $\phi[x := y]$ [3]. *Insert translated proof of the premise here.*" or, we simply say: "Pick an arbitrary y and show $\phi[x := y]$. *Insert translated proof of the premise here.*"

[3] In this rule, and those that follow, we say $\phi[x := y]$ to be the formula that results from the substitution of y fopr x in *phi*, *i.e.* actually do the substitution before writing the formula in your proof.

The formal rule for $\forall$ on the left says:

$$\frac{\Gamma_1, \phi[x := t], \Gamma_2 \vdash \Delta}{\Gamma_1, \forall x.\phi, \Gamma_2 \vdash \Delta} \ (\forall\text{L}) \qquad \text{where } t \in \mathcal{T}.$$

We say: " Since we know that for every x, ϕ is true, assume $\phi[x := t]$. *Insert translated proof of premise here.*" or, we say: "Assume $\phi[x := t]$."

Existential Quantifiers

The rule for $\exists$ on the right is:

$$\frac{\Gamma \vdash \Delta_1, \phi[x := t], \Delta_2}{\Gamma \vdash \Delta_1, \exists x.\phi, \Delta_2} \ (\exists\text{R}) \qquad \text{where } t \in \mathcal{T}.$$

We say: "Let t be the witness for x in $\exists x.\phi$. We must show $\phi[x := t]$.*Insert translated proof of the premise here.*" or, we say to show $\exists x.\phi$, we choose the witness t and show $\phi[x := t]$. *Insert translated proof of the premise here.*".

The rule for $\exists$ on the left is:

$$\frac{\Gamma_1, \phi[x := y], \Gamma_2 \vdash \Delta}{\Gamma_1, \exists x.\phi, \Gamma_2 \vdash \Delta} \ (\exists\text{L}) \qquad \begin{array}{l}\text{where variable } y \text{ is not free in any} \\ \text{formula of } (\Gamma \cup \Delta_1 \cup \{\exists x.\phi\} \cup \Delta_2).\end{array}$$

We say: "Since we know $\exists x.\phi$, pick an arbitrary element of the domain of discourse, call it y, and assume $\phi[:= y]$. *Insert translated proof of the premise here.*" or, we say: "we know ϕ, holds for arbitrate x, so assume $\phi[x := y]$. *Insert translated proof of the premise here.*"

Part II

Sets, Relations and Functions

Chapter 4

Set Theory

In this chapter we present elementary set theory. Set theory serves as a foundation for mathematics[1], *i.e.* in principle, we can describe all of mathematics using set theory.

Our presentation is based on Mitchell's [24]. A classic presentation can be found in Halmos' [18] *Naive Set Theory*.

4.1 Introduction

Set theory is the mathematical theory of collections. A set is a collection of abstract objects where the order and multiplicity of the elements is not taken into account.

4.1.1 Informal Notation

From given objects we can form *sets* by collecting together some or all of the given objects.

We write down small sets by enclosing the elements in curly brackets "{" and "}". Thus, the following are sets.

$$\{a, 1, 2\}$$
$$\{a\}$$
$$\{1, 2, a\}$$
$$\{a, a, a\}$$

Sometimes, if a pattern is obvious, we use an informal notation to indicate larger sets without writing down the names of all the elements in the set. For example, we might write:

[1] We say it is "a foundation" since alternative approaches to the foundations of mathematics exist *e.g.* category theory or type theory can also serve as the foundations of mathematics. Set theory is the foundational theory accepted by most mathematicians.

$$\{0, 1, 2, ..., 10\}$$
$$\{0, 2, 4, ..., 10\}$$
$$\{2, 3, 5, 7, \cdots, 23, 29\}$$
$$\{0, 1, 2, ...\}$$
$$\{..., -1, 0, 1, 2, ...\}$$

These marks denote: the set of natural numbers from zero to ten; the set of even numbers from zero to ten; the set consisting of the first 10 prime numbers, the set of natural numbers; and the set of integers. The sequence of three dots ("...") notation is called an *ellipsis* and indicates that the some material has been omitted intensionally. In describing sets, the cases of the form $\{..., \Gamma\}$ or $\{\Gamma, ...\}$ indicate some pattern (which should be obvious from Γ) repeats indefinitely. We present more formal ways of concisely writing down these sets later in this chapter.

4.1.2 Membership

The objects included in a set are called the *elements* or *members* of the set.

Membership is a primitive notion in set theory, as such it is not formally defined. Rather, it should be thought of as an undefined primitive relation; it is used in set theory to characterize the properties of sets.

We indicate an object x is a member of a set A by writing

$$x \in A$$

We sometimes will also say, *"A contains x"* or *"x is in A"*.

The statement $x \in A$ is a true proposition if x actually is in A. We read the symbol "$\notin$" as *not in* and define it by negating the membership proposition:

$$x \notin A \stackrel{\text{def}}{=} \neg(x \in A)$$

Evidently, the following are all true propositions.

$$a \in \{a, 1, 2\}$$
$$1 \notin \{a\}$$
$$1 \in \{1, 2, a\}$$
$$2 \notin \{a, a, a\}$$

Note that sets may contain other sets as members. Thus,

$$\{1, \{1\}\}$$

is a set and the following propositions are true.

$$1 \in \{1, \{1\}\}$$
$$\{1\} \in \{1, \{1\}\}$$

Consider the following true proposition.

$$1 \notin \{\{1\}\}$$

Now this last statement can be confusing[2].

$$\{1\} \notin \{1\}$$

4.1.3 Equality

Throughout mathematics and computer science, whenever a new notion or structure is introduced (sets in this case) we must also say when instances of the objects are equal.

Sets are determined by their members or elements. This means, the only property significant for determining when two sets are equal is the membership relation. Thus, in a set, the order of the elements is insignificant and the number of times an element occurs in a set (its *multiplicity*) is also insignificant. This equality (*i.e.* the one that ignores multiplicity and order) is called *extensionality*.

Definition 4.1.

$$A = B \stackrel{\text{def}}{=} \forall x.\, (x \in A \Leftrightarrow x \in B)$$

We write $A \neq B$ for $\neg(A = B)$.

Consider the following sets.

$$\begin{gathered} \{a, 1, 2\} \\ \{a\} \\ \{1, 2, a\} \\ \{a, a, a\} \end{gathered}$$

The first and the third are equal as sets and the second and the fourth are equal. It is not unreasonable to think of these equal sets as different descriptions (or names) of the same mathematical object.

Note that the set $\{1, 2\}$ is not equal[3] to the set $\{1, \{2\}\}$, this is because $2 \in \{1, 2\}$ but $2 \notin \{1, \{2\}\}$.

4.1.4 Subsets

Definition 4.2. A set A is a subset of another set B if every element of A is also an element of B. Formally, we write:

$$A \subseteq B \stackrel{\text{def}}{=} \forall x.\, (x \in A \Rightarrow x \in B)$$

Thus, the set of even numbers (call this set $2\mathbb{N}$) is a subset of the natural numbers $\mathbb{N}$, in symbols $2\mathbb{N} \subseteq \mathbb{N}$.

Theorem 4.1. For every set A, $A \subseteq A$

[2]Indeed there are some serious philosophers who reject it as senseless [13].

[3]It may be interesting to note that some philosophers with *nominalist* tendencies[12, 13] deny the distinction commonly made between these sets, they say that the "don't understand" the distinction being made between these sets.

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \in A \vdash x \in A}\,(\mathrm{Ax})}{\vdash x \in A \Rightarrow x \in A}\,\Rightarrow\mathrm{R}}{\vdash \forall x.\,(x \in A \Rightarrow x \in A)}\,\forall\mathrm{R}}{\vdash A \subseteq A}\,(\subseteq\mathrm{def})}{\vdash \forall A.\,A \subseteq A}\,(\forall\mathrm{R})$$

□

Remarks on the difference between element of and subset.

The reader should not confuse the notions $x \in A$ and $x \subset A$.

Relating equality to subsets

We can prove the following theorem which relates extensionality with subsets.

Theorem 4.2. For every set A and every set B,

$$A = B \Leftrightarrow ((A \subseteq B) \wedge (B \subseteq A))$$

Proof: Since the theorem is an if and only if, we must show two cases; we label them below as ($\Rightarrow$) and ($\Leftarrow$).

($\Rightarrow$) Assume $A = B$, we must show that $A \subseteq B$ and that $B \subseteq A$. By definition, if $A = B$, then $\forall x.\,(x \in A \Leftrightarrow x \in B)$. First, to show that $A \subseteq B$, we must show that $\forall x.\,(x \in A \Rightarrow x \in B)$. Pick an arbitrary thing, call it y and we must show that $y \in A \Rightarrow y \in B$, but we assumed $A = B$, this means (by the definition of equality) $y \in A \Leftrightarrow y \in B$. Since the definition of equality is an iff, we may assume $y \in A \Rightarrow y \in B$ and $y \in B \Rightarrow y \in A$. But then we have show $y \in A \Leftrightarrow y \in B$ as was desired. The for argument to show the case $B \subseteq A$ is similar.

($\Leftarrow$) Assume $((A \subseteq B) \wedge (B \subseteq A))$, *i.e.* that $A \subseteq B$ and $B \subseteq A$, we must show that $A = B$. By definition, this is true if and only if $\forall x.x \in A \Leftrightarrow x \in B$. Pick an arbitrary thing, call it y and show $y \in A \Leftrightarrow y \in B$, *i.e.* show $y \in A \Rightarrow y \in B$ and $y \in B \Rightarrow y \in A$. Using y in the assumption $A \subseteq B$ gives the first case and using y in the assumption that $B \subseteq A$ gives the second.

□

4.1.5 Empty Set

There exists a set containing no elements.

We can write this fact as follows:

$$\exists A.\forall x.x \notin A$$

The definition says that there exists a set A which, for any thing whatsoever (call it x), that thing is not in the set A.

Definition 4.3. Consider a property (say P) if, no matter what two things we pick (say x and y), if whenever $P(x)$ and $P(y)$ hold then $x = y$, we say that the property P is *unique*.

$$unique(P) \stackrel{\text{def}}{=} \forall x, y.(P(x) \wedge P(y)) \Rightarrow (x = y)$$

We can prove that the empty set is unique *i.e.* we can prove the following theorem which says that any two sets having the property that they contain no elements are equal. In this case the property P is defined as $P(z) = \forall x.x \notin z$.

Theorem 4.3. For every set A and every set B, if $\forall x.x \notin A$ and $\forall x.x \notin B$, then $A = B$.

Proof: We give a formal sequent proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \in A \vdash x \in B, x \in A, x \in B}(\text{Ax})}{\vdash x \in B, x \in A, (x \in A \Rightarrow x \in B)}(\Rightarrow\text{R}) \quad \dfrac{\dfrac{}{x \in B \vdash x \in B, x \in A, x \in A}(\text{Ax})}{\vdash x \in B, x \in A, (x \in B \Rightarrow x \in A)}(\Rightarrow\text{R})}{\vdash x \in B, x \in A, (x \in A \Rightarrow x \in B) \wedge (x \in B \Rightarrow x \in A)}(\wedge\text{R})}{\neg(x \in B) \vdash x \in A, (x \in A \Rightarrow x \in B) \wedge (x \in B \Rightarrow x \in A)}(\neg\text{L})}{\neg(x \in A), \neg(x \in B) \vdash (x \in A \Rightarrow x \in B) \wedge (x \in B \Rightarrow x \in A)}(\neg\text{L})}{\neg(x \in A), x \notin B \vdash (x \in A \Rightarrow x \in B) \wedge (x \in B \Rightarrow x \in A)}(\{\notin\}\text{-def})}{x \notin A, x \notin B \vdash (x \in A \Rightarrow x \in B) \wedge (x \in B \Rightarrow x \in A)}(\{\notin\}\text{-def})}{x \notin A, \forall x.x \notin B \vdash (x \in A \Rightarrow x \in B) \wedge (x \in B \Rightarrow x \in A)}(\forall\text{L})}{\forall x.x \notin A, \forall x.x \notin B \vdash (x \in A \Rightarrow x \in B) \wedge (x \in B \Rightarrow x \in A)}(\forall\text{L})}{(\forall x.x \notin A) \wedge (\forall x.x \notin B) \vdash (x \in A \Rightarrow x \in B) \wedge (x \in B \Rightarrow x \in A)}(\wedge\text{L})}{(\forall x.x \notin A) \wedge (\forall x.x \notin B) \vdash x \in A \Leftrightarrow x \in B}(\{\Leftrightarrow\}\text{-def})}{(\forall x.x \notin A) \wedge (\forall x.x \notin B) \vdash \forall x.(x \in A \Leftrightarrow x \in B)}(\forall\text{R})}{(\forall x.x \notin A) \wedge (\forall x.x \notin B) \vdash A = B}(\{=\}\text{-def})}{\vdash ((\forall x.x \notin A) \wedge (\forall x.x \notin B)) \Rightarrow A = B}(\Rightarrow\text{R})}{\vdash \forall B.(((\forall x.x \notin A) \wedge (\forall x.x \notin B)) \Rightarrow A = B)}(\forall\text{R})}{\vdash \forall A.\forall B.(((\forall x.x \notin A) \wedge (\forall x.x \notin B)) \Rightarrow A = B)}(\forall\text{R})$$

□

Since the set is unique we can give it a name, we denote this unique set by the constant[4] symbol $\emptyset$ (and sometimes by just writing empty brackets $\{\}$).

Using this new notation, we can rewrite the axiom as follows.

[4]Do not confuse the symbol $\emptyset$ with the Greek letter ϕ.

Axiom 4.1 (Empty Set)

$$\forall x.x \notin \emptyset$$

With this definition, we can prove the following theorem:

Theorem 4.4. For every set A, $\emptyset \subseteq A$.

4.1.6 Comprehension

If we are given a set and a predicate $\phi(x)$ (a property of elements of the set) we can create the set consisting of those elements that satisfy the property. We write the set created by the operation by instantiating the following schema.

Axiom 4.2 (Comprehension) If $\mathcal{S}$ is a meta-variable denoting an arbitrary set, x is a variable and $\phi(x)$ is a predicate, then the following denotes a set.

$$\{x \in \mathcal{S} \mid \phi(x)\}$$

This is a very powerful mechanism for defining new sets.

Note that we characterize the elements in a set defined by comprehension as follows.

$$y \in \{x \in \mathcal{S} | \phi(x)\} \stackrel{\text{def}}{=} (y \in \mathcal{S} \wedge \phi(y))$$

Lets consider a few examples of how to use comprehension. We assume the natural numbers ($\mathbb{N} = \{0, 1, 2, \cdots\}$) has already been defined.

Example 4.1. The set of natural numbers greater than 5 can be defined using comprehension as:

$$\{n \in \mathbb{N} \mid \exists m.\ m \in \mathbb{N} \wedge m + 6 = n\}$$

Example 4.2. We can define the set of even numbers as follows.

First, note that a natural number n is even if and only if there is another natural number (say m), such that $n = 2m$. (*e.g.* if n is 0 (an even natural number), then if $m = 0$, $2m = n$. If n is 2, then $m = 1$ gives $2m = n$, etc.) Thus, n is even if and only if $\exists m.m \in \mathbb{N} \wedge 2m = n$. Using this predicate of n, we can define the set of even natural numbers as follows.

$$\{n \in \mathbb{N} \mid \exists m.m \in \mathbb{N} \wedge 2m = n\}$$

Here, the set $\mathcal{S}$ from the schema is the set of natural numbers $\mathbb{N}$ and the predicate ϕ is:

$$\phi(n) = \exists m.m \in \mathbb{N} \wedge 2 * m = n$$

4.1.7 Ordered Pairs

The set $\{a, b\}$ and the set $\{b, a\}$ are identical as far as set equality goes. What if we want to be able to distinguish order, is it possible in the set notation? The following encoding of ordered pairs was first given by Kurstowski.

Definition 4.4 (ordered pair)

$$\langle a, b\rangle \stackrel{\text{def}}{=} \{\{a\}, \{a, b\}\}$$

Under this definition $\langle 1, 2\rangle = \{\{1\}, \{1, 2\}\}$ and $\langle 2, 1\rangle = \{\{2\}, \{1, 2\}\}$. As sets, $\langle 1, 2\rangle \neq \langle 2, 1\rangle$. Also, not that the pair consisting of two of the same elements is encoded as the set containing the set containing that element.

$$\langle 1, 1\rangle = \{\{1\}, \{1, 1\}\} = \{\{1\}, \{1\}\} = \{\{1\}\}$$

Theorem 4.5. For every a, b, a', and b',

$$\langle a, b\rangle = \langle a', b'\rangle \Leftrightarrow (a = b \wedge a' = b')$$

Definition 4.5. We define the *projection functions* which map pairs $\langle x, y\rangle$ to their first and second components such that

$$\pi_1\langle x, y\rangle = x$$
$$\pi_2\langle x, y\rangle = y$$

4.1.8 Power Set

Consider the collection of all subsets of a given set, this collection is itself a set and is called the *power set*. We write the power set of a set $\mathcal{S}$ as $\rho(\mathcal{S})$.

The axiom characterizing membership in the powerset says:

$$x \in \rho(\mathcal{S}) \stackrel{\text{def}}{=} x \subseteq \mathcal{S}$$

Consider the following examples:

$A_0 = \{\}$	$\rho A_0 = \{\{\}\}$
$A_1 = \{1\}$	$\rho A_1 = \{\{\}, \{1\}\}$
$A_2 = \{1, 2\}$	$\rho A_2 = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$
$A_3 = \{1, 2, 3\}$	$\rho A_1 = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

Notice that the size of the power set is growing exponentially (as powers of 2, $2^0 = 1, 2_1 = 2, 2^2 = 4, 2^3 = 8$)

Fact 4.1. If a set A has n elements, then the power set ρA has 2^n elements.

4.1.9 Singletons and unordered pairs

Definition 4.6 ((unordered) pair) A set having exactly two elements is called an *unordered pair*.

The following axiom asserts that given any two things, there is a set whose elements are those two things and only those elements.

Axiom 4.3 (Pairing)

$$\forall x.\forall y.\exists A.\forall z.z \in A \Rightarrow (z = x \vee z = y)$$

Note that although we might believe such a set exists, without recourse to the pairing axiom, we would have no justification for asserting such a set exists.

By picking x and y to be the same element we get a *singleton*, a set having exactly one element.

Definition 4.7 (singleton) A set containing exactly one element is called a *singleton*.

Note that the singleton set $\{x\}$ is distinguished from its element x, *i.e.* $x \neq \{x\}$.

Theorem 4.6 (singleton equality)

$$\forall x, y.\ \{x\} = \{y\} \Leftrightarrow x = y$$

Exercise 4.1. Prove the singleton equality theorem.

4.1.10 Set Union

Since we distinguish sets by their members, we can define operations on sets that construct new sets by indicating when an element is a member of the constructed set. Union is the operation of putting two collections together. If A and B are sets, we write $A \cup B$ for the set consisting of the members of A and of B. We characterize the membership in a union as follows:

$$x \in (A \cup B) \stackrel{\text{def}}{=} (x \in A \vee x \in B)$$

Thus if $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$ then $A \cup B = \{1, 2, 3, 4\}$.

The empty set acts as an identity element for the union operation (in the same way 0 is the identity for addition and 1 is the identity for multiplication.) This idea is captured by the following theorem.

Theorem 4.7. For every set A, $A \cup \emptyset = A$.

Proof: We give a formal sequent proof of the theorem.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \in A \vdash x \in A}\,(\text{Ax})}{x \in A \vee x \in \emptyset \vdash x \in A}\,(\vee\text{-L})}{x \in (A \cup \emptyset) \vdash x \in A}\,(\in \cup \text{ def})}{\vdash x \in (A \cup \emptyset) \Rightarrow x \in A}\,(\Rightarrow\text{-R})}{\vdash (x \in (A \cup \emptyset) \Rightarrow x \in A) \wedge (x \in A \Rightarrow A \cup \emptyset)}\,(\wedge\text{-R})}{\vdash x \in (A \cup \emptyset) \Leftrightarrow x \in A}\,(\text{def of } \Leftrightarrow)}{\vdash \forall x.x \in (A \cup \emptyset) \Leftrightarrow x \in A}\,(\forall\text{R})}{\vdash A \cup \emptyset = A}\,(\text{def of } =)}{\vdash \forall A.\ A \cup \emptyset = A}\,(\forall\text{R})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \in \emptyset \vdash x \in \emptyset, x \in A}\,(\text{Ax})}{\neg(x \in \emptyset), x \in \emptyset \vdash x \in A}\,(\neg\text{-L})}{x \notin \emptyset, x \in \emptyset \vdash x \in A}\,(\text{def of } \notin)}{\forall x.x \notin \emptyset, x \in \emptyset \vdash x \in A}\,(\forall\text{-L})}{x \in \emptyset \vdash x \in A}\,(\text{Assert})$$

$$\dfrac{\dfrac{\dfrac{}{x \in A \vdash x \in A, x \in \emptyset)}\,(\text{Ax})}{x \in A \vdash x \in A \vee x \in \emptyset)}\,(\vee\text{-R})}{x \in A \vdash x \in (A \cup \emptyset)}\,(\in \cup \text{ def})}{\vdash x \in A \Rightarrow x \in (A \cup \emptyset)}\,(\Rightarrow\text{-R})$$

□

Theorem 4.8. For sets A and B, $A \cup B = B \cup A$.

Proof: Choose arbitrary sets A and B. By extensionality, $A \cup B = B \cup A$ is true if $\forall x.x \in (A \cup B) \Leftrightarrow x \in (B \cup A)$. Choose an arbitrary x, assume $x \in (A \cup B)$. Now, by the definition of membership in a union, $x \in (A \cup B)$ iff $x \in A \vee x \in B$. $(x \in A \vee x \in B) \Leftrightarrow (x \in B \vee x \in A$ and, again by the union membership property, ($x \in B$ or $x \in A$) $\Leftrightarrow x \in (B \cup A)$.
□

By this theorem, $A \cup \emptyset = \emptyset \cup A$ which, together with Thm 4.7 yields the following corollary.

Corollary 4.1. For every set A, $\emptyset \cup A = A$.

Theorem 4.9. For all sets A and B, $A \subseteq (A \cup B)$.

Proof: Choose arbitrary sets A and B. By the definition of subset, $A \subseteq A \cup B$ is true if $\forall x.x \in A \Rightarrow x \in (A \cup B)$. Choose an arbitrary x, assume $x \in A$. Now, $x \in (A \cup B)$ if $x \in A$ or $x \in B$. Since we have assumed $x \in A$, the theorem holds.
□

By Thm 4.8, we have the following:

Corollary 4.2. For all sets A and B, $A \subseteq (B \cup A)$.

4.1.11 Set Intersection

We define the operation of collecting the elements in common with two sets and call it the *intersection*. Membership in an intersection is defined as follows:

$$x \in (A \cap B) \stackrel{\text{def}}{=} (x \in A \wedge x \in B)$$

Thus if $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$ then $A \cap B = \{2, 3\}$.

Theorem 4.10. For every set A, $A \cap \emptyset = \emptyset$.

Proof: By extensionality, $A \cap \emptyset = \emptyset$ is true iff $\forall x. x \in (A \cap \emptyset) \Leftrightarrow x \in \emptyset$. Choose an arbitrary x. We must show

$$\begin{array}{ll} i.) & x \in (A \cap \emptyset) \Rightarrow x \in \emptyset \\ ii.) & x \in \emptyset \Rightarrow x \in (A \cap \emptyset) \end{array}$$

$i.$) Assume $x \in (A \cap \emptyset)$, then, by the membership property of intersections, $x \in A \wedge x \in \emptyset$. And now, by the emptyset axiom (Axiom **??**, the second conjunct is a contradiction, so the implication in the left to right direction holds.
$ii.$) Assume $x \in \emptyset$. Again, by the emptyset axiom, this is false so the right to left implication holds vacuously.
□

Theorem 4.11. $\forall A.\ A \cap A = A$.

4.1.12 Set Difference

Definition 4.8 (difference) Given a set A and a set B, the difference of A and B, (write $A - B$) is the set of elements in A that are not in the set B. More formally:

$$A - B = \{x : A \mid x \notin B\}$$

Example 4.3. If *Even* is the set of even natural numbers, $\mathbb{N} - \mathit{Even} = \mathit{Odd}$.

Theorem 4.12. For every set A, $A - \emptyset = A$.

Theorem 4.13. For every set A, $A - A = \emptyset$.

Theorem 4.14. For all sets A and B, $A - B = \emptyset \Leftrightarrow A \subseteq B$.

Definition 4.9 (disjoint sets) Two sets A and B are *disjoint* if they share no members in common, *i.e.* if the following holds:

$$A \cap B = \emptyset$$

Theorem 4.15. For all sets A and B, A and B are disjoint sets iff $A - B = A$.

4.2 Cartesian Products and Tuples

Definition 4.10 (Cartesian product) The *Cartesian product* of sets A and B is the set of all ordered pairs having a first element from A and second element from B. We write $A \times B$ to denote the Cartesian product.

$$A \times B \stackrel{\text{def}}{=} \{z \in \rho(\rho(A \cup B)) | \exists a : A. \exists b : B. z = \langle a, b \rangle\}$$

Note that, by Def 4.4. $z = \langle a, b \rangle$ means z is a set of the form $\{\{a\}, \{a, b\}\}$. Evidently, the Cartesian product of two sets is a set of pairs.

Example 4.4. If $A = \{a, b\}$ and $B = \{1, 2, 3\}$

$$A \times A = \{\langle a, a \rangle, \langle a, b \rangle, \langle b, a \rangle, \langle b, b \rangle\}$$
$$A \times B = \{\langle a, 1 \rangle, \langle a, 2 \rangle, \langle a, 3 \rangle, \langle b, 1 \rangle, \langle b, 2 \rangle, \langle b, 3 \rangle\}$$
$$B \times A = \{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, a \rangle, \langle 1, b \rangle, \langle 2, b \rangle, \langle 3, b \rangle\}$$
$$B \times B = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle\}$$

Theorem 4.16. For any set A,

$$i.) \quad A \times \emptyset = \emptyset$$
$$ii.) \quad \emptyset \times A = \emptyset$$

Proof: (of i) Choose an arbitrary x.

($\Rightarrow$): Assume $x \in (A \times \emptyset)$, but this is true only if there exists an a, $a \in A$ and a b, $b \in \emptyset$ such that $x = \langle a, b \rangle$. But by the emptyset axiom there is no such b so this case holds vacuously.
($\Leftarrow$): We assume $x \in \emptyset$ which, by the emptyset axiom, is a contradiction so this case holds vacuously as well.
□

So theorem 4.16 says that $\emptyset$ is both a left and right identity for Cartesian product.

Lemma 4.1 (pair lemma) If A and B are arbitrary sets, $\forall x : A \times B. \exists y : A. \exists z : B. x = \langle y, z \rangle$

Remark 4.1 (Proving membership in comprehensions defined over Cartesian Products) To be syntactically correct, a set defined by comprehension over a Cartesian product appears as follows:

$$\{y \in A \times B | P[y]\}$$

In practice, we often need to refer to the parts of the pair y to express the property P. If so, to be formally correct, we should write :

$$\{y : A \times B \mid \forall z : A. \forall w : B. y = \langle z, w \rangle \Rightarrow P(z, w)\}$$

So,

$$x \in \{y : A \times B \mid \forall z : A. \forall w : B. y = \langle z, w \rangle \Rightarrow P(z, w)\}$$
$$\Leftrightarrow x \in A \times B \wedge \forall z : A. \forall w : B. x = \langle z, w \rangle \Rightarrow P(z, w)$$

By lemma 4.1, we know there exist $z \in A$ and $w \in B$ such that $\langle z, w \rangle$. So, to prove membership of x it is enough to show that $x \in A \times B$ and then assume there are $z \in A$ and $w \in B$ such that $x = \langle z, w \rangle$ and show $P[z, w]$. A more readable syntactic form allows the "destructuring" of the pair to occur on the left side in the place of the variable.

$$\{\langle x, y \rangle \in A \times B \mid P[x, y]\}$$

Under the rule for proof just described, to show membership

$$z \in \{\langle x, y \rangle \in A \times B \mid P[x, y]\}$$

Show $z \in A \times B$ and then, assume $z = \langle x, y \rangle$ (for new variables x and y) and show $P[x, y]$.

Chapter 5

Relations

5.1 Introduction

Relations establish a correspondence between the elements of sets. In keeping with the principle that all of mathematics can be described using set theory, relations (and functions) themselves can be characterized as sets (having a certain kind of structure).

For example, familial relations can be characterized mathematically using the relational structures and/or functions. Thus, if the set $\mathcal{P}$ is the set of all people living and dead, the relationship between a (biological) father and his offspring could be represented by a set of pairs F of the form $\langle x, y\rangle$ to be interpreted as meaning that x is the father of y if $\langle x, y\rangle \in F$. We will write xFy to denote the fact that x is the father of y instead of $\langle x, y\rangle \in F$. Using this notation, the paternal grandfather relation can be characterized by the set

$$\{\langle x, y\rangle \in \mathcal{P} \times \mathcal{P} | \exists z.\ xFz \wedge zFy\}$$

A function is a relation that satisfies certain global properties; most significantly, the functionality property. A relation R is functional if $\langle x, y\rangle \in R$ and $\langle x, z\rangle \in R$ then $y = z$. This is a mathematical way of specifying the condition that there can only be one pair in the relation R having x as its first entry. We discuss this in more detail below. Now, if we consider the father-of relation given above, it clearly is not a function since one father can have more than one child *e.g.*if Joe has two children (say Tommy and Susie) then there will be two entries in F with Joe as the first entry. However, if we take the inverse relation (we might call it *has father*), we get a function since each individual has only one biological father. Mathematically, we could define this relation as $yF^{-1}x \stackrel{\text{def}}{=} xFy$.

Relations and functions play a crucial rile in both mathematics and computer science. Within computer science, programs are usefully thought of as functions. Relations and operations on them form the basis of most modern database systems, so-called relational databases.

5.2 Binary Relations

Definition 5.1.

> A *(binary) relation* is a subset of a Cartesian product. Given sets A, B and R, if $R \subseteq A \times B$ we say R is a binary relation on A and B.

Thus, a (binary) relation is a set of pairs. *A relation is a set of pairs.* Say it to yourself three times and do not forget it. Every time you get in the shower for a week, repeat this as your mantra.

Example 5.1. Let A and B be sets. Any subset R, $R \subseteq A \times B$ is a relation. Thus $A \times B$ itself is a relation. This one is not very interesting since every element of A is related to every element of B.

Example 5.2. The empty set is a relation. Recall that the empty set is a subset of every set and so it is a subset of every Cartesian product (even the empty one). Again, this is not a terribly interesting relation but, by the definition, it clearly is one.

Example 5.3. Less-than ($<$) is a relation on $(\mathbb{Z} \times \mathbb{Z})$.

To aid readability, relations are often written in infix notation *e.g.* $\langle x, y\rangle \in R$ will be written xRy. So, for example, an instance of the less-than relation will be written $3 < 100$ instead of the more pedantic $\langle 3, 100\rangle \in <$.

Definition 5.2. If $R \subseteq A \times A$ we say R *is a relation on* A.

Thus, $<$ is a relation on $\mathbb{Z}$.

5.2.1 Operations on Relations

Note that since relations are sets (of pairs) all the ordinary set operations are defined on them. In particular, unions, intersections, differences and comprehensions are all valid operations to perform on relations. Also, because relations inherit their notion of equality from the fact that they are sets, relations are equal when they are equal as sets. It makes perfect sense to say that one relation is a subset of another.

We define a number of operations that are specifically defined on relations – mainly owing to the fact that they are not simply sets, but have additional structure in that they are sets of pairs.

Definition 5.3 (inverse) If $R \subseteq A \times B$, then the *inverse* of R is

$$R^{-1} = \{\langle y, x\rangle \in B \times A | \langle x, y\rangle \in R\}$$

Example 5.4. The inverse of the less-than relation ($<$) is greater-than ($>$).

Definition 5.4 (complement) If $R \subseteq A \times B$, then the *complement* of R is the relation

$$\overline{R} = \{\langle x, y\rangle \in A \times B | \langle x, y\rangle \notin R\}$$

Exercise 5.1. Prove that if $R \subseteq A \times B$, then $\overline{R} = (A \times B) - R$

Example 5.5. The complement of the less-than relation ($<$) is the greater-than-or-equal-to relation ($\geq$).

Definition 5.5 (composition) If $R \subseteq A \times B$ and $S \subseteq B \times C$, then the *composition* of R and S is the relation defined as follows:

$$S \circ R \stackrel{\text{def}}{=} \{\langle x, y\rangle \in A \times C \mid \exists z{:}B.\ \langle x, z\rangle \in R \wedge \langle z, y\rangle \in S\}$$

Remark 5.1. To some, it may seem backward to write $S \circ R$ instead of $R \circ S$. In fact, both conventions do appear in the mathematical literature – though the convention adopted here is the most common one – it is not the only one. The reason adopting this convention might be more clear when we get to functions.

Example 5.6. Suppose we had a relation (say R) that paired names with social security numbers and another relation that paired social security numbers with the state they were issued in (call this relation S), then $(S \circ R)$ is the relation pairing names with the states where their social security numbers were assigned.

Composition can very naturally be iterated. Before we describe the iterative version we introduce the diagonal relation.

Definition 5.6. The *diagonal relation* over a set A is the relation

$$\Delta_A = \{\langle x, y\rangle \in A \times A | x = y\}$$

This relation is called the "diagonal" in analogy with the matrix presentation of a relation R, where $\langle x, y\rangle$ in the matrix is labeled with a 0 if $\langle x, y\rangle \notin R$ and $\langle x, y\rangle = 1$ if $\langle x, y\rangle \in R$.

Example 5.7. Suppose $A = \{0, 1, 2\}$ and $R = \{\langle 0, 1\rangle, \langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle\}$ then the matrix representation appears as follows:

R	0	1	2
0	1	0	0
1	0	1	1
2	0	1	0

Under this matrix representation, the so-called diagonal relation Δ_A appears as the follows:

Δ_A	0	1	2
0	1	0	0
1	0	1	0
2	0	0	1

Notice that is it the matrix consisting of a diagonal of ones [1]

Theorem 5.1. If R is any relation on A, then $(R \circ \Delta_A) = R$.

Proof: The theorem says Δ_A is a right identity for composition. To see that the relations (sets of pairs) $(R \circ \Delta_A)$ and R are equal, we apply Thm 4.4.2, *i.e.* we show ($\subseteq$): $R \circ \Delta_A \subseteq R$ and ($\supseteq$): $R \subseteq R \circ \Delta_A$.
($\subseteq$): Assume $\langle x, y\rangle \in (R \circ \Delta_A)$. Then, by the definition of composition, there exists a $z \in A$ such that $\langle x, z\rangle \in R$ and $\langle z, y\rangle \in \Delta_A$. But by the definition of Δ_A, $z = y$ and so, replacing z by y we get $\langle x, y\rangle \in R$ which is what we are to show.

($\supseteq$): Assume $\langle x, y\rangle \in R$. Then, to see that $\langle x, y\rangle \in R \circ Delta_A$ we must show there exists a $z \in A$ such that $\langle x, z\rangle \in R$ and $\langle z, y\rangle \in R$. Let z be y. Clearly, $\langle y, y\rangle \in \Delta_A$ and also, by our assumption $\langle x, y\rangle \in R$.
□

Exercise 5.2. Prove the following:
For every relation $R \subset A \times B$, $\Delta_B \subseteq R \circ R^{-1} =$

Definition 5.7 (iterated composition) We define the *iterated composition* of a relation R on a set A with itself as follows.

$$R^0 = \Delta_A$$
$$R^{k+1} = R \circ R^k$$

Corollary 5.1. For all relations R on a set A, $R^1 = R$, since, by the definition of iterated composition and by theorem 5.1 we have: $R^1 = R \circ R^0 = R \circ \Delta_A = R$.

Typically, we only consider the case where $R \subseteq A \times A$, but the definition is still sensible if the relation R is a binary relation on $A \times B$, so long as $B \subseteq A$.

Example 5.8. Suppose R is the relation on natural number associating each number with its successor, $R = \{\langle x, y\rangle \in \mathbb{N} \times \mathbb{N} \mid x = y + 1\}$. Then R^k is the relation associating each number with its k^{th} successor; $R^k = \{\langle x, y\rangle \in \mathbb{N} \times \mathbb{N} \mid x = y + k\}$.

5.2.2 Properties of Relations

A relation may satisfy certain structural properties. The properties all say something about the "shape" of the relation.

Definition 5.8. A relation $R \subset A \times A$ is

[1] Students with some familiarity with linear algebra will know that this is the identity matrix.

1.)	reflexive	$\forall a : A.\ aRa$
2.)	irreflexive	$\forall a : A.\ \neg(aRa)$
3.)	symmetric	$\forall a, b : A.\ aRb \Rightarrow bRa$
4.)	antisymmetric	$\forall a, b : A.\ (aRb \wedge bRa) \Rightarrow a = b$
5.)	transitive	$\forall a, b, c : A.\ (aRb \wedge bRc) \Rightarrow (aRc)$
6.)	connected	$\forall a, b : A.\ a \neq b \Rightarrow (aRb \vee bRa)$

Remark 5.2. Note that a relation can fail to be both reflexive and irreflexive. Let $A = \{0, 1, 2\}$ and $R = \{\langle 0, 1\rangle, \langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 0\rangle\}$ Then, R is not reflexive because $\langle 0, 0\rangle \notin R$. But it also fails to be irreflexive since $\langle 1, 1\rangle \in R$.

5.2.3 Closures of Relations

The idea of "closing" a relation with respect to a certain property is the idea of *adding just enough* to the relation to make it satisfy the property (if it doesn't already.)

Example 5.9. Consider A and R as just presented in remark 5.2. We can "close" R under the reflexive property by unioning the set $E = \{\langle 0, 0\rangle, \langle 2, 2\rangle\}$ with R. This is the minimal extension of R that makes it reflexive. Adding, for example, $\langle 2, 1\rangle$ does not contribute to the reflexivity of R and so it is not added. Also note that even though $E \neq \Delta_A$, $R \cup E = R \cup \Delta_A$ since $\langle 1, 1\rangle \in R$.

Thus, the closure is the minimal extension to make a relation satisfy a property. For some properties (like irreflexivity) there may be no way add to the relation to make it satisfy the property – in which case we say the closure "does not exist". To make R satisfy irreflexivity, we would have to *remove* $\langle 1, 1\rangle$.

Definition 5.9. Given a relation $R \subseteq A \times B$ and a property P of the relation, the *closure* of R with respect to P is the relation S such that $P(S)$ and $R \subseteq S$ and S is the smallest such relation,

$$\begin{aligned} &closure(R, P) = S \\ &\quad \text{iff } (P(S) \wedge R \subseteq S) \wedge \forall T : T \subseteq A \times B \Rightarrow ((P(T) \wedge R \subseteq T) \Rightarrow S \subseteq T) \end{aligned}$$

If we close a relation with respect to a property P that the relation already enjoys, the result is just the relation R itself. The reader is invited to verify this fact by proving the following lemma.

Lemma 5.1. Given a relation $R \subseteq A \times B$ and a property P of the relation, if $P(R)$ holds, then $closure(R, P) = R$.

Definition 5.10. The predicate $Refl_A(R)$ means R is reflexive with respect to the set A.

$$Refl_A(R) \stackrel{\text{def}}{=} \forall x : A.xRx$$

Theorem 5.2. If $R \subseteq A \times A$ then the reflexive closure of R is the relation $R \cup \Delta_A$

Proof: More formally, the theorem says

$$closure(R, Refl_A) = R \cup \Delta_A$$

Thus, to show that the $R \cup \Delta_A$ is the reflevive closure, (by the definition of *closure*) we must show three things:

i.) $Refl_A(R \cup \Delta_A)$
ii.) $R \subseteq (R \cup \Delta_A)$
iii.) $\forall T : T \subseteq A \times A \Rightarrow ((Refl_A(T) \wedge R \subseteq T) \Rightarrow (R \cup \Delta_A) \subseteq T)$

i.) More particularly, we must show that $\forall x : A.\ \langle x, x\rangle \in (R \cup \Delta_A)$. Choose an arbitrary $x \in A$. Then, by the membership property of unions, we must show that $\langle x, x\rangle \in R$ or $\langle x, x\rangle \in \Delta_A$. But by the definition of membership in a comprehension, $\langle x, x\rangle \in \Delta_A$ iff $\langle x, x\rangle \in A \times A$ (which is obviously true since x was arbitrarily chosen from the set A) and if $x = x$. So, we conclude that (i) holds.

ii.) We must show that $R \subseteq (R \cup \Delta_A)$. But this is true by Thm 4.9 from Chapter 4.
iii.) Finally, we must show that $R \cup \Delta_A$ is the least such set, *i.e.* that

$$\forall T : T \subseteq A \times A \Rightarrow ((R \subseteq T \wedge Refl_A(T)) \Rightarrow (R \cup \Delta_A) \subseteq T)$$

To see this, choose an arbitrary relation $T \subseteq A \times A$. Assume $R \subseteq T$ and $Refl_A(T)$. We must show that $(R \cup \Delta_A) \subseteq T$. Let x be an arbitrary element of $(R \cup \Delta_A)$. Then, there are two cases: $x \in R$ or $x \in \Delta_A$. If $x \in R$, since we have assumed $R \subseteq T$, we know $x \in T$. In the other case, $x \in \Delta_A$, that is, x is of the form $\langle y, y\rangle$ for some y in A. But since we assumed $Refl_A(T)$, we know that $\forall z : A.\langle z, z\rangle \in T$ so, in particular, $\langle y, y\rangle \in T$, *i.e.* $x \in T$.
□

Definition 5.11. The predicate $Sym(R)$ means $R \subseteq A \times A$ is symmetric.

$$Sym(R) \stackrel{\text{def}}{=} \forall x, y : A.\ xRy \Rightarrow yRx$$

Note that unlike reflexivity, symmetry does not require us to know what the full set A is, it only requires us to know what pairs are in the relation R.

Example 5.10. For any set A, the empty relation is symmetric, though the empty relation is reflexive if and only if $A = \emptyset$.

Theorem 5.3. If $R \subseteq A \times A$ then the symmetric closure of R is the relation $R \cup R^{-1}$

Example 5.11. Let $A = \{0, 1, 2, 3\}$ and $R = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 0\rangle\}$. Then

$$\begin{aligned}
R^1 &= R = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 0\rangle\} \\
R^2 &= R \circ R = \{\langle 0, 2\rangle, \langle 1, 3\rangle, \langle 2, 0\rangle, \langle 3, 1\rangle\} \\
R^3 &= R \circ R^2 = \{\langle 0, 3\rangle, \langle 1, 0\rangle, \langle 2, 1\rangle, \langle 3, 2\rangle\} \\
R^4 &= R \circ R^3 = \{\langle 0, 0\rangle, \langle 1, 1\rangle, \langle 2, 2\rangle, \langle 3, 3\rangle\} \\
R^5 &= R \circ R^4 = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 0\rangle\}
\end{aligned}$$

Note that $R^5 = R$. The transitive closure of R is the union

$$R \cup R^2 \cup R^3 \cup R^4$$

Definition 5.12. The predicate $Trans(R)$ means $R \subseteq A \times A$ is transitive.

$$Trans(R) \stackrel{\text{def}}{=} \forall x, y, z : A.\ (xRy \wedge yRz) \Rightarrow xRz$$

Theorem 5.4. If $R \subseteq A \times A$ then the transitive closure of R is the relation

$$R^+ = \bigcup_{i>0} R^i$$

Definition 5.13. The *reflexive transitive closure* of a relation $R \subseteq A \times A$ is the relation

$$R^* = \bigcup_{i \in \mathbb{N}} R^i$$

Remark 5.3. The notation R^* to denote the reflexive transitive closure of a relation R is borrowed from the theory of strings and regular languages. It is borrowed from S. C. Kleene (1909-1994) and is sometimes (especially in the context of regular expressions) called the *Kleene Closure* of R.

5.3 Equivalence Relations and Partitions

Equivalence relations are a generalization of the idea of equality.

Definition 5.14 (equivalence relation) A relation on a set A that is reflexive (on A) , symmetric and transitive is called an *equivalence relation on* A. We will sometimes write $Equiv_A(R)$ to mean R is an equivalence relation on A.

$$Equiv_A(R) \stackrel{\text{def}}{=} Refl_A(R) \wedge Sym(R) \wedge Trans(R)$$

Example 5.12. Ordinarily equality on numbers is an equivalence relation.

Example 5.13. In propositional logic, the if-and-only-if connective [Def. 2.2.3] is an equivalence on propositions. To see this we must show three things:

i.)	$\forall P.P \Leftrightarrow P$	(Reflexive)
ii.)	$\forall P, Q.(P \Leftrightarrow Q) \Rightarrow (Q \Leftrightarrow P)$	(Symmetric)
iii.)	$\forall P, Q, Q.(P \Leftrightarrow Q) \wedge (Q \Leftrightarrow R) \Rightarrow (P \Leftrightarrow R)$	(Transitive)

But these theorems have all been proved previously as exercises, so $\Leftrightarrow$ is an equivalence on propositions.

Example 5.14. The reflexive closure of the sibling relation is an equivalence relation. To see this, By the reflexive closure, everyone is related to him or herself by this relation, because we explicitly stated it is closed under reflection.

If A is the sibling of B, the B is the sibling of A so the relation is symmetric. And finally, if A is the sibling of B and B is the sibling of C, then A is the sibling of C so the relation is transitive.

Note that, under this relation, if an individual has no brothers or sisters, there is no other person (except herself by virtue of the reflexive closure) related to her.

Lemma 5.2. For any set A,, the diagonal relation Δ_A is an equivalence relation on A.

This is the so-called "finest" equivalence on any set A and is defined by real equality on the elements of the set A. To see this recall that $\Delta_A = \{\langle x, y\rangle \mid x = y\}$

Lemma 5.3. For any set A,, the complete relation $A \times A$ is an equivalence relation on A.

This equivalence is rather uninteresting, it says every element in A is equivalent to every other element in A. It is the "coarsest" equivalence relation on the set A.

Equivalence Classes

It is often useful to define the set of all the elements from a set A equivalent to some particular element under an equivalence relation $R \subseteq A \times A$.

Definition 5.15 (equivalence class) If A is a set, R is an equivalence relation on A and and x is an element of A, then the *equivalence class of x modulo R* (we write $[x]_R$) is defined as follows.

$$[x]_R = \{y \in A \mid xRy\}$$

Example 5.15. If S is the reflexive closure of the sibling relation, then for any individual x, $[x]_S$ is the set consisting of x and of his or her brothers and sisters.

Theorem 5.5. If A is a set, R is an equivalence relation on A, and x and y are elements of A, then the following statements are equivalent.

i. xRy

ii. $[x]_R \cap [y]_R \neq \emptyset$

iii. $[x]_R = [y]_R$

Proof: To prover these statements are equivalent, we will show, $(i) \Rightarrow (ii)$ and $(ii) \Rightarrow (iii)$ and finally, $(iii) \Rightarrow (i)$.

$[(i) \Rightarrow (ii)]$ Assume xRy. Then, by the definition of $[x]_R$, $y \in [x]_R$. Also, by reflexivity of R (recall it is an equivalence relation) yRy and so $y \in [y]_R$. But then, y is in both $[x]_r$ and y is in $[y]_R$, hence the intersection is not empty and we have shown $[x]_R \cap [y]_R \neq \emptyset$.

$[(ii) \Rightarrow (iii)]$ Assume $[x]_R \cap [y]_R \neq \emptyset$. Then, thre must be some element (say z)(such that $z \in [x]_R$ and $z \in [y]_R$. We show $[x]_R = [y]_R$, *i.e.* we show that $\forall w.w \in [x]_R \Leftrightarrow x \in [y]_R$. Choose an arbitrary w. But then $w \in [x]_R \Leftrightarrow xRw$. By the symmetry of R, we know wRx. Now, since $z \in [x]_R$, xRz and by transitivity of R, wRz holds as well. Now, since $z \in [y]_R$, yRz and by symmetry we have zRy and by transitivity we get wRy. Finally, another application of symmetry allows us to conclude yRw and we have shown that $w \in [x]_R \Leftrightarrow w \in [y]_R$ for arbitrary w, thus $[x]_R = [y]_R$ if their intersection is non-empty.

$[(iii) \Rightarrow (i)]$ Assume $[x]_R = [y]_R$. Then, every element of $[x]_R$ is in $[y]_R$ and vice-versa. But, because R is reflexive, $x \in [x]_R$ and since $y \in [y]_R$, $y \in [x]_R$. But this is true only if xRy holds.

□

5.3.1 Partitions

Definition 5.16 (Partition) A *partition* of a set A is a set of subsets of A (we refer to these sets as A_i where $i \in I, I \subseteq \mathbb{N}$). Each A_i is called a *block* and a collection of such A_i is a partition if it satisfies the following two properties:

i.) the sets in the blocks are pairwise disjoint, *i.e.*

$$\forall i, j : I.\ \ i \neq j \Rightarrow (A_i \cap A_j = \emptyset)$$

and,

ii.) the union of the sets $A_i, i \in I$ is the set A itself:

$$\bigcup_{i \in I} A_i = A$$

Example 5.16. If $A = \{1, 2, 3\}$ then the following are all the partitions of A.

$$\{\{1, 2, 3\}\}$$
$$\{\{1\}, \{2, 3\}\}$$
$$\{\{1, 2\}, \{3\}\}$$
$$\{\{1, 3\}, \{2\}\}$$
$$\{\{1\}, \{2\}, \{3\}\}$$

Theorem 5.6. For any set A, $R \subseteq A \times A$ is an equivalence relation if and only if the set of its equivalence classes form a partition *i.e.*

$$Equiv_A(R) \Leftrightarrow Partition(\bigcup_{x \in A} \{[x]_R\})$$

Definition 5.17 (k-partition) A *k-partition* of a set A is a partition of A into k subsets.

So for example, $\{\{1, 2, 3\}\}$ is a 1-partition of $\{1, 2, 3\}$, $\{\{1\}, \{2, 3\}\}, \{\{1, 2\}, \{3\}\}$, and $\{\{1, 3\}, \{2\}\}$ are all 2-partitions while $\{\{1\}, \{2\}, \{3\}\}$ is a 3-partition.

5.3.2 Counting

Definition 5.18 (Counting k-partitions) The following recurrence relation[2] can be used to compute the number of k-partitions of a set of size n.

$$\begin{aligned} S(n,1) &= 1 \\ S(n,n) &= 1 \\ S(n,k) &= S(n-1,k-1) + k \cdot S(n-1,k) \end{aligned}$$

Definition 5.19. Counting Equivalence Relations There are as many equivalence relations on a set of size n as there are k-partitions $k \in \{1 \cdots n\}$.

$$\sum_{k=1}^{n} S(n,k)$$

[2]The numbers computed by this recurrence relation are called Stirling Numbers of the second kind.

Chapter 6

Functions

Some relations have the special property that they are functions. A relation $R \subseteq a \times B$ is a function if each element of the domain A gets mapped to one element and only one element of the codomain B.

6.1 Functions

A *function* from A to B is a relation ($f \subseteq A \times B$) satisfying the following properties,

$$\forall x : A.\exists y : B.\langle x, y\rangle \in f$$
$$\forall x : A.\forall y, z : B.\langle x, y\rangle \in f \wedge \langle x, z\rangle \in f \Rightarrow y = z$$

Since we usually write $f(x) = y$ instead of $\langle x, y\rangle \in f$, we can restate these properties in the more familiar notation as follows.

$$\forall x : A.\exists y : B.f(x) = y$$
$$\forall x : A, y, z : B.f(x) = y \wedge f(x) = z \Rightarrow y = z$$

The first property is called *totality* and the section is called *functionality*.

We write the set of all functions from A to B as $A \to B$, so if $f \subseteq A \times B$ is a function we write $f : A \to B$.

Definition 6.1 (domain, codomain, range) If $f : A \to B$, we call the set A the *domain* of f and the set B the *codomain* of f. The set $\{y \in B | \exists x : A.f(x) = y\}$ is called the *range* of f. We write $dom(f)$ to denote the set which is the domain of f, $codom(f)$ to denote the codomain and $rng(f)$ to denote it range.

6.1.1 Equivalence

We define function equality extensionally, as equality on the underlying sets, thus, for functions $f, g : A \to B$,

$$f = g \stackrel{\text{def}}{=} \forall x : A.f(x) = g(x)$$

Remarks on Extensional Equivalence

The definition of equality for functions is based on the so-called extensional view of functions *i.e.* functions as sets of pairs. Within computer science, we might be interested in notions of equivalence that take into account other properties besides simply the input-output behavior of functions. For example, programs (say $\mathcal{P}_1$ and $\mathcal{P}_2$) that sort lists of numbers are functions from lists to lists and though as sets $\mathcal{P}_1 = \mathcal{P}_2$ must be true if they both actually implement sorting correctly; the two programs may have significantly different run-time complexities. Program $\mathcal{P}_1$ may implement the merge sort algorithm which has $O(nlogn$ time complexity while program $\mathcal{P}_2$ may implement insertion sort which has time complexity $O(n^2)$. So, if we consider their run-time complexities, the two are clearly not equivalent. Many other proprieties are not accounted for by extensional equality; indeed, the only property that *is* accounted for is the input-output behavior.

6.1.2 Operations on Functions

Since functions are relations, and thus are sets of pairs, all the operations on sets and relations make sense as operations on functions.

Inverse

Given a relation $R \subseteq A \times A$, recall the definition of R inverse, $R^{-1} = \{\langle y, x\rangle \in B \times A \mid xRy\}$. Now, consider a function $f : A \to B$. Since f is a relation, the relation f^{-1} exists; but is it a function?

Example 6.1. Let $f : A \to B$ be a function. Suppose that for some $x, y \in A$, where $x \neq y$, that for some z, $f(x) = z$ and $f(y) = z$. But then, $\langle z, x\rangle \in f^{-1}$ and $\langle z, y\rangle \in F^{-1}$ so, in this case, f^{-1} is not a function since it violates the functionality condition. We conclude that if any two elements of A get mapped to the same element of B, then f^{-1} is not a function.

Example 6.2. Let $f : A \to B$ be a function. Suppose that for some $z \in B$, there is no x such that $f(x) = z$. But then, there is no pair in f^{-1} whose first element is z. This violates the totality condition and so f^{-1} is not a function if there is some element of B not mapped onto by f.

6.1.3 Composition of Functions

Consider functions $f : A \to B$ and $g : B \to C$, the following picture illustrates the situation.

$$A \xrightarrow{f} B \xrightarrow{g} C$$

There is a function $h : A \to C$ defined so that $h(x) = g(f(x))$. In this case, h is called the *composition* of g and f. In general, if the domain of g is the codomain

of f, $(dom(g) = codom(f))$ we write their composition as $g \circ f$. It is defined as follows:

$$(g \circ f)(x) \stackrel{\text{def}}{=} g(f(x))$$

Thus, function composition is a binary operator on pairs of functions analogous to the way addition is a binary operation on pairs of integers. The analogy goes deeper. Addition is associative *e.g.* if a, b and c are numbers, $a + (b + c) = (a + b) + c$. Function composition is associative as well.

Theorem 6.1. If $f : A \to B, g : B \to C$ and $h : C \to D$ then

$$h \circ (g \circ f) = (h \circ g) \circ f$$

Proof: To show two functions are equal, we must apply extensionality. To show

$$h \circ (g \circ f) = (h \circ g) \circ f$$

we must show that for every $x \in A$,

$$(h \circ (g \circ f))(x) = ((h \circ g) \circ f)(x)$$

Pick an arbitrary x. The following sequence of equalities (derived by unfolding the definition of composition and then folding it back up) shows the required equality holds.

$$\begin{aligned}
&(h \circ (g \circ f))(x) \\
&\quad = h((g \circ f)(x)) \\
&\quad = h(g(f(x))) \\
&\quad = (h \circ g)(f(x)) \\
&\quad = ((h \circ g) \circ f)(x))
\end{aligned}$$

□

Zero (0) is both a left and right identity for addition *i.e.* $0 + a = a$ and $a + 0 = a$. Similarly, the identity function $Id(x) = x$ is a left and right identity for the operation of function composition.

Theorem 6.2. If Id is the identity function $(Id(x) = x)$ then, for any sets A and B, and any function $f : A \to B$, Id is both a left and right identity.

$$f \circ Id = f \quad \text{and} \quad Id \circ f = f$$

Proof: To show two functions are equal, we apply extensionality, choosing an arbitrary x.

$$(f \circ Id)(x) = f(Id(x)) = f(x)$$

Thus Id is a left identity for $\circ$. Similarly,

$$(Id \circ f)(x) = Id(f(x)) = f(x)$$

Thus $(f \circ Id) = f$ and $Id \circ f = f$ and the theorem has been shown.
□

6.1.4 Injections, Surjections and Bijections

Functions may possess various properties.

Definition 6.2 (injection, one-to-one) A function $f : A \to B$ is an *injection* (or *one-to-one*) if f maps every element of A to exactly one element of B. Formally, we write:

$$\forall x, y : A.\ \ f(x) = f(y) \Rightarrow x = y$$

The definition says that if f maps x and y to the same element, then it must be that x and y are one in the same *i.e.* that $x = y$. Injections are also called *one-to-one* functions.

Definition 6.3 (surjection, onto) A function $f : A \to B$ is an *surjection* (or *onto*) if every element of B is mapped to by some element of A under f.

$$\forall y : B.\ \ \exists x : A.\ \ f(x) = y$$

Another way of saying this would be to say that $codom(f) = rng(f)$.

Definition 6.4 (bijection) A function $f : A \to B$ is a *bijection* if it is both an injection and a surjection. Bijections are sometimes called *one-to-one correspondences*.

Theorem 6.3. A function $f : A \to B$ has an inverse f^{-1} iff it is bijective.

6.2 Cardinality

The term *cardinality* refers to the relative "size" of a set.

Definition 6.5 (equal cardinality) Two sets A and B have the same cardinality iff there exists a bijection $f : A \to B$. In this case we write $|A| = |B|$ or $A \sim B$.

Although the usage is less common, sometimes sets of equal cardinality are said to be *equipollent* or *equipotenet*.

Definition 6.6 (even) $Even = \{x : \mathbb{N} | \exists y : \mathbb{N}.\ x = 2y\}$.

Theorem 6.4. $|\mathbb{N}| = |Even|$.

Proof: To show these sets have equal cardinality we must find a bijection between them. Let $f(n) = 2n$, we claim $f : \mathbb{N} \to Even$ is a bijection. To see this, we must show it is both: (i.) one-to-one and (ii.) onto.
(i.) f is one-to-one, *i.e.* we must show:

$$\forall x, y : \mathbb{N}, f(x) = f(y) \Rightarrow x = y$$

Choose arbitrary $x, y \in \mathbb{N}$. Assume $f(x) = f(y)$ and we show $x = y$. But by the definition of f, if $f(x) = f(y)$ then $2x = 2y$ and so $x = y$ as we were to show.

(i.) f is onto, *i.e.* we must show:

$$\forall x : Even.\, \exists y : \mathbb{N}.\, x = f(y)$$

Choose an arbitrary x and assume $x \in Even$. Then, $x \in \{x : \mathbb{N} | \exists y : \mathbb{N}.\, x = 2y\}$ is true so we know, $x \in \mathbb{N}$ and $\exists y : \mathbb{N}.\, x = 2y$. To see that $\exists y : \mathbb{N}.\, x = f(y)$, note that $f(y) = 2y$.
□

This theorem may be rather surprising, it says that the set of natural numbers is the "same size" as the set of even numbers. Clearly there are only half as many evens as there are naturals, but somehow these sets are the same size. This is one of the unintuitive aspects of infinite sets. This seeming paradox, that a proper subset of an infinite set can be the same size, was first noticed by Galileo [10] and is sometimes called *Galileo's paradox* [35] after the Italian scientist Galileo Galilei (1564 – 1642).

Definition 6.7. $Squares = \{x : \mathbb{N} \mid \exists y : \mathbb{N}.\, x = y^2\}$.

Exercise 6.1. Prove that $|\mathbb{N}| = |Squares|$.

Definition 6.8 (less equal cardinality) The cardinality of a set A is at most the cardinality of B iff there exists an injection $f : A \to B$. In this case we write $|A| \leq |B|$.

Definition 6.9 (strictly smaller cardinality) The cardinality of a set A is less than the cardinality of B iff there exists an injection $f : A \to B$ and there is no bijection from A to B. In this case we write $|A| < |B|$. Formally,

$$|A| < |B| \stackrel{\text{def}}{=} |A| \leq |B| \wedge |A| \neq |B|$$

Typically, it is harder to prove a set A has strictly smaller cardinality that a set B becuase it is harder to prove that *no* function in $A \to B$ is a bijection. To prove this we usually assume there is a bijection and derive a contradiction.

Theorem 6.5 (Cantor's Theorem) For every set A, $|A| < |\rho(A)|$.

Proof: Let A be an arbitrary set. To show $|A| < |\rho(A)|$ we must show that (i.) there is an injection $A \to B$ and (ii) there is no bijection from A to B.
(i.) Let $f(x) = \{x\}$. We claim that this is a bijection form A to $\rho(A)$ (the set of all subsets of A. Clearly, for each $x \in A$, $f(x) \in \rho(A)$. To see that f is an injection we must show:

$$\forall x, y : A.\, f(x) = f(y) \Rightarrow x = y$$

Choose arbitrary x and y from A. Assume $f(x) = f(y)$, *i.e.* that $\{x\} = \{y\}$, we must show $x = y$. But, by Theorem 4.4.6, $\{x\} = \{y\} \Leftrightarrow x = y$, thus f is an injection as was claimed.

(ii.) To see that there is no bijection, we assume $f : A \to B$ is an arbitrary function and show that it can not be onto.

Now, if f is onto then every subset of A must be mapped to from some element of A. Consider the set

$$B = \{y : A \mid y \notin f(y)\}$$

Clearly $B \subseteq A$, so $B \in \rho(A)$. Now, if f is onto, there is some $z \in A$ such that $f(z) = B$. Also, it must be the case that $z \in B \vee z \notin B$.
(case 1.) Assume $z \in B$. Then, $z \in \{y : A \mid y \notin F(y)\}$, that is, $z \in A$ (as we assumed) and $z \notin f(z)$. Since we assumed $f(z) = B$, we have $z \notin B$. But we started by assuming $z \in B$ so this is absurd.
(case 2.) Assume $z \notin B$. Then, $\neg(z \in \{y : A \mid y \notin f(y)\}$. By the definition of membership in a set defined by comprehension, $\neg(z \in A \wedge z \notin f(z))$. By DeMorgan's law, $(z \notin A \vee \neg(z \notin f(z)))$. Since we know $z \in A$ it must be that $\neg(z \notin f(z))$, *i.e.* that $z \in f(z)$. Since $f(z) = B$ we have $z \in B$. But again, this is absurd because we started this argument by assuming $z \notin f(z)$.
Since we have arrived at a contradiction in both cases, we conclude that the function f can not be a bijection.
□

Cantor's theorem gives a way to take any set and use it to construct a set of strictly larger cardinality. Thus, we can construct a hierarchy of non-equivalent infinities. Start with the set $\mathbb{N}$ and take the power set. By Cantor's theorem, $|\mathbb{N}| < |\rho(\mathbb{N})|$. Similarly, $|\rho(\mathbb{N})| < |\rho(\rho(\mathbb{N}))|$ and so on.

6.3 Infinite and Finite

The following definition of infinite is sometimes called Dedekind infinite after the mathematician Richard Dedekind (1831-1916) who first formulated it. It is somwhat surprising because it does not mention natural numbers or the notion of finiteness.

Definition 6.10 (infinite) A set A is *infinite* iff there exists a function $f : A \to A$ that is one-to-one but not onto.

Theorem 6.6. $\mathbb{N}$ is infinite.

Proof: Consider the function $f(n) = n + 1$. Clearly, f is one-to-one since if $f(x) = f(y)$ for arbitary x and y in $\mathbb{N}$, then $x+1 = y+1$ and so $x = y$. However, f is not onto since there is no element of $\mathbb{N}$ that is mapped to 0 by f.
□

Definition 6.11 (finite) A set A is finite iff there exists a natural number n such that $|A| = |\{k : \mathbb{N} \mid 0 \leq k < n\}|$. More formally, we write

$$finite(A) \stackrel{\text{def}}{=} \exists n : \mathbb{N}.\ |A| = |\{k : \mathbb{N} \mid 0 \leq k < n\}|$$

.

Theorem 6.7. A set A if infinite iff A is not finite.

6.4 Exercises

1. Write down the formal definitions of injection, surjection and bijection using the notation $\langle x, y\rangle \in f$ instead of the abbreviated form $f(x) = y$. Note that you will need to include a new variable (say z) to account for $f(x) = f(y)$ in this more primitive notation.

Part III

Induction and Recursion

Chapter 7

Natural Numbers

The German mathematician and logician Leopold Kronecker (1823 - 1891) famously remarked:

God made the natural numbers; all else is the work of man.

Kronecker was saying the natural numbers are absolutely primitive and that other structures have been defined by men. Similarly, Immanual Kant (1742 – 1804) and Luitzen Egbertus Jan Brouwer (1881 - 1966) both believed that understanding of natural numbers is somehow innate and that it arises from intuition about the human experience of time as a sequence of moments. [1] In any case, it would be hard to argue against the primacy of the natual numbers among mathematical structures.

7.1 Peano Arithmetic

The *Peano axioms* are named for Giuseppe Peano (1858–1932), an Italian mathematician and philosopher. Peano first presented his axioms [26] of arithmetic in 1889, though in a later paper Peano credited Dedekind [5] with the first presentation of the axioms. We still know them as Peano's axioms.

Peano's axioms are stated as follows:

$$
\begin{aligned}
&i.) && 0 \in \mathbb{N} \\
&ii.) && \forall k : \mathbb{N}.\ sk \in \mathbb{N} \\
&iii.) && \forall k : \mathbb{N}.\ 0 \neq sk \\
&iv.) && \forall, j, k : \mathbb{N}.\ j = k \Leftrightarrow sj = sk \\
&v.) && (P[0] \wedge \forall k : \mathbb{N}.\ P[k] \Rightarrow P[sk]) \Rightarrow \forall n : \mathbb{N}.\ P[n]
\end{aligned}
$$

So, natural numbers are either of the form 0 or sk where k is some previously constructed natural number. We call s the *sucessor function* and sk is the

[1] Interestingly, Kant also believed that geometry was similarly primitive and outr intuition of it arises from our experience of three dimensional space. The discovery in the late 19th century of non-Euclidean geometries [**?**] makes this idea seem quaint by modern standards.

successor of k. Axiom *iii* says 0 is not the successor of any natural number and *iv* says the successor function preserves equality. Property v is the induction principle which is the main topic of discussion of this chapter.

Equivalently, as we did in Chapter 1, Definition 1.1, we can define the natural numbers inductively as follows. The idea of an inductive definition is a generalization of Peano's axioms for arithmetic.

$$\mathbb{N} ::= \mathbf{0} \mid \mathtt{s}\, n$$

where the alphabet consists of the symbols $\{\mathbf{0}, s\}$ and n is a variable denoting some previously constructed element of the set $\mathbb{N}$.

Under either definition, the set of natural number $\mathbb{N}$ is the set

$$\{0, s0, ss0, sss0, ssss0, \cdots\}$$

The Peano axioms *iii*, *iv*, and v all follow from the fact that this set is inductively defined; in particular, every inductively defined set admits an (structural) induction principle (axiom *v.*) that can be constructed by examining the definition.

7.2 Definition by Recursion

We have defined functions by recursion earlier (*e.g.* the *val* function given in Chapter 2 Def. 2.10). The idea of defining functions by recurion on the structure of one of its arguments here is the same.

Definition 7.1 (Addition)

$$\begin{aligned} k + 0 &= k \\ k + sn &= s(k_n) \end{aligned}$$

Example 7.1. So, to add $3 + 3$,

$$sss0{+}sss0 = s(sss0{+}ss0) = ss(sss0{+}s0) = sss(sss0{+}0) = sss(sss0) = ssssss0$$

Or to add $0 + 3$

$$0 + sss0 = s(0 + ss0) = ss(0 + s0) = sss(0 + 0) = sss0$$

Definition 7.2 (multiplication)

$$\begin{aligned} k \cdot 0 &= 0 \\ k \cdot sn &= (k \cdot n) + k \end{aligned}$$

Example 7.2. So, to multipley $3 \cdot 2$,

$$sss0 \cdot ss0 = (sss0 \cdot s0) + sss0 = ((sss0 \cdot 0) + sss0) + sss0 = (0 + sss0) + sss0 = sss0 + sss0 = ssssss0$$

7.3 Mathematical Induction

Suppose you wished to justify the principle of mathematical induction.

$$(P[0] \wedge \forall k : \mathbb{N}.\ P[k] \Rightarrow P[sk]) \Rightarrow \forall n : \mathbb{N}.\ P[n]$$

It says, to prove a property P is true for every natural number n, it is enough to know two things:

i. $P[0]$

ii. $\forall k : \mathbb{N}. P[k] \Rightarrow P[sk]$

So, suppose you accept i and ii. Now, previously, to prove $\forall n : \mathbb{N}.\ P[n]$, choose an arbitrary n and assume $n \in \mathbb{N}$. Now show $P[n]$. You can do it as follows,

1.	$P[0]$	you accepted this as i
2.	$P[0] \Rightarrow P[s0]$	instantiate ii with 0
3.	$P[s0]$	modus ponens of 1,2
4.	$P[s0] \Rightarrow P[ss0]$	instantiate ii with $s0$
5.	$P[ss0]$	modus ponens of 3,4
$\vdots$	$\vdots$	
2n.	$P[s^{(n-1)}] \Rightarrow P[s^n 0]$	instantiate ii with $s^{(n-1)}$
2n+1.	$P[s^n 0]$	modus ponens of 2n and 2n+1

So, no matter what n is choosen, we can prove $P[n]$ holds in $2n + 1$ steps using the base case (i) and the induction step (ii), both of which we assumed.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Jonathan Barnes. *Logic and the Imperial Stoa*, volume LXXV of *Philosophia Antiqua*. Brill, Leiden · New York · Koln, 1997.

[3] Noam Chomsky. *Syntactic Structures*. Number 4 in Janua Linguarum, Minor. Mouton, The Hague, 1957.

[4] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951.

[5] Richard Dedekind. *Was sind and was sollen die Zahlen?* 1888. English translation in [6].

[6] Richard Dedekind. *Essays on the Theory of Numbers*. Dover, 1963.

[7] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[8] E. Engeler. *Formal Languages: Automata and Structures.* Markham, Chicago, 1968.

[9] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[10] Galileo Galilei. *Two New Sciences*. University of Wisconsin Press, 1974. Translated by Stillman Drake.

[11] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[12] Nelson Goodman. *The Structure of Appearance, Third ed.*, volume 107 of *Synthese Library*. D. Reidel, Dordrecht, 1977.

[13] Nelson Goodman and W. V. Quine. Steps toward a constructive nominalism. *Jounral of Symbolic Logic*, 12:105 – 122, 1947.

[14] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, New York, 1993.

[15] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. MIT Press, 1992.

[16] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming, Types, Semantics and Language Design*. Types, Semantics, and Language Design. MIT Press, Cambridge, MA, 1994.

[17] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633–674. North-Holland, 1990.

[18] Paul R. Halmos. *Naive Sert Theory*. Springer Verlag, New York, Heidelberg, Berlin, 1974.

[19] Herodotus. *The History*. University of Chicago, 1987. Translated by David Green.

[20] D. Hilbert and W. Ackermann. *Mathematical Logic*. Chelsea Publishing, New York, 1950.

[21] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, Massachusetts, 1969.

[22] Xavier Leroy. *The Objective Caml system release 1.07*. INRIA, France, May 1997.

[23] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1991.

[24] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[25] L. C. Paulson. *Standard ML for the Working Programmer*. Cambridge University Press, 1991.

[26] Giuseppe Peano. *Arithmetices principia, nova methodo exposita*, 1899. English translation in [34], pp. 83–97.

[27] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Dept., Denmark, 1981.

[28] W.V.O. Quine. *Methods of Logic*. Holt, Rinehart and Winston, 1950.

[29] Arto Salomaa. *Formal Languages*. ACM Monograph Series. Academic Press, 1973.

[30] S. A. Schmidt. *Denotational Semantics*. W. C. Brown, Dubuque, Iowa, 1986.

[31] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings Symposium on Computers and Automata*, pages 19–46. Polytechnic Inst. of Brooklyn Press, New York, 1971.

[32] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[33] R. D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, London, 1991.

[34] Jan van Heijenoort, editor. *From Frege to Gödel: A sourcebook in mathematical logic, 1879 – 1931.* Harvard University Press, 1967.

[35] Wikipedia. Galileo's paradox — wikipedia, the free encyclopedia, 2005.

[36] G. Winskel. *Formal Semantics of Programming Languages.* MIT Press, Cambridge, 1993.

[37] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus.* Routledge and Kegan Paul Ltd., London, 1955.