Logic and Discrete Mathematics for Computer Scientists

James Caldwell

Department of Computer Science
University of Wyoming
Laramie, Wyoming

Draft of
August 26, 2011

[1]This material is based upon work partially 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.

The text which perhaps most closely embodies the approach taken here may be Gries and Schneider's [21]. 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 [18].

Another text that is close in style to this one is *The Haskell Road to Logic, Maths and Programming'* by Doets and van Eijck [9]. In that textbook, the authors use the functional programming language Haskell as a computational basis for much of the text and present proofs in an informal natural deduction style.

Manna an Waldinger's text *The Logical Basis for Computer Programming* [36] is an excellent presentation of the material presented here and much more.

Unfortunately, that two volume work is now out of print.

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 seeing 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.

Acknowledgments: Thanks go to Eric Berg, Andrew Blair, John Dumkee and other anonymous students in COSC 2300 at the University of Wyoming for being careful readers and for providing feedback on the text.

Chapter 1

Syntax and Semantics*

In this chapter we give a brief overview of syntax and semantics. We describe, without delving to deeply into the all the formal details, how to specify abstract syntax using inductive definitions, 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 a program is the result of executing it. 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 [5]. 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 complexity of the devices that can be used to generate the terms in the language.

The study of formal languages has an extensive literature of it's own [**?**, 12, 29, 45, **?**]. Similarly, the study of mathematical semantics of programming languages is a rich research area in its own right [47, 48, 42, 46, 50, 23, 24, 22, 53, 1, 38]

1.3 Syntax

We can finitely describe abstract syntax in a number of ways. A common way is to describe the terms of the language inductively by giving a formal grammar describing how terms of the language can be constructed. 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.

Definition 1.1 (grammar) 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

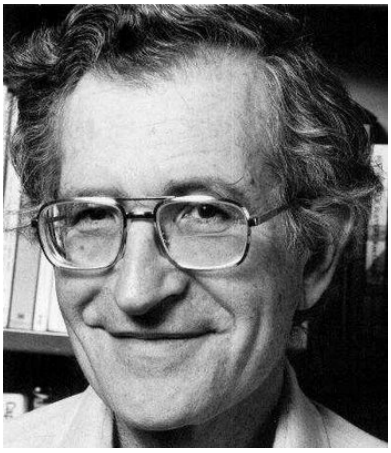
class: is the name of the syntactic class being defined, and

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 rules separated by the vertical bar are alternatives. 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 describe the allowable forms of the structures included in the language. The constructors are either constants from the alphabet, are elements from some collection of sets, or describe how to construct new complex constructs consisting of symbols from the alphabet, elements from the parameter sets, and possibly from 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 syntactic structures in the language defined by the grammar are finite. These non-recursive alternatives (the ones that do not have subparts which are of the type of structure being defined) are sometimes called the *base cases*.

Two syntactic elements are equal if they are constructed using identical constructors applied to equal arguments. It is never the case that $c_1 x = c_2 x$ if c_1 and c_2 are different constructors.



Noam Chomsky

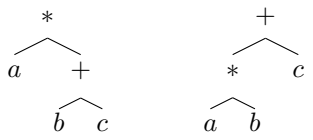
Noam Chomsky (1928-)is the father of modern linguistics. In 1959 he characterized formal languages in terms of their generative power and laid the mathematical foundation for the study of formal languages. He is also known as political activist.

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 above we 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. **??** and the formula $(a * b) + c$ is displayed by the tree on the right of Fig. **??**. 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.

[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 [27] 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. I theorize that notions of right and left may have first been understood only in relation to the linear form introduced by writing. In that case, if right means "the side of the papyrus you start on when writing a new line" then the Egyptian interpretation of right and left coincide with the Greeks.

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 $\mathbf{T}$ and $\mathbf{F}$. Although this is enough, *i.e.* it is enough to say that a Boolean is either the symbol $\mathbf{T}$ or is the symbol $\mathbf{F}$, 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* $\mathbf{T}$ *or the symbol* $\mathbf{F}$*.*

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 $\mathbf{T}$ or $\mathbf{F}$. There are no other abstract syntax trees 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:

Definition 1.2.

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

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

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

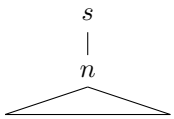
[3] "Boolean" is eponymous for George Boole, the English mathematician who first formulated symbolic logic in symbolic algebraic form.

Figure 1.1: Syntax trees for terms $s\mathbf{0}$ and $sss\mathbf{0}$

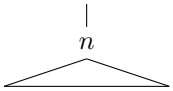
the set $\{\mathbf{0}, s\mathbf{0}, ss\mathbf{0}, \cdots\}$. Such place-holders are called *meta-variables* and are required if the language has inductive structure, *i.e.* if we define the elements of the language using previously constructed elements of the language.

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 above The subtree for a previously constructed $\mathbb{N}$ labeled n is displayed by the following 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. 1.1.

Syntax of a simple computational language

We define a simple language PLB (Programming Language Booleans) of Boolean expressions with a semantics allowing us to compute 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, *i.e.* the Booleans defined above serve as the *base case* for the language.

Definition 1.3 (PLB)

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

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:

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

Definition 1.4 (T List)

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

where

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

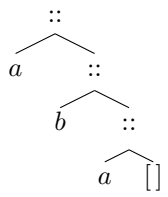


Figure 1.2: Syntax tree for the list $[a, b, a]$ constructed as $a::(b::(a::[\,]))$

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*.

Example 1.1. 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:

1.3.3 Definitions

A definition is a way to extend a language to possibly include new symbols but to describe them in terms of the existing language. Adding a definition does not allow anything new to be said that could not already have been; though definitions can be extraordinarily useful in making things clear. The key idea behind defined terms is that they can be completely eliminated by just replacing them by its definition.

Definition 1.5 (definitions) A *definition* is a template or schematic form that introduces new symbols into an existing language as an abbreviation for another (possibly more complicated) term. Definitions have the form

$$\mathbf{A}[x_1, \cdots, x_k] \stackrel{\text{def}}{=} \mathbf{B}[x_1, \cdots, x_k]$$

where $x_i, 1 \leq i \leq k$ are variables standing for terms of the language (defined so-far). An instance of the defined term is the form $\mathbf{A}[t_1, \cdots, t_k]$ where the x_i's are instantiated by terms t_i. This term is an abbreviation (possibly parametrized if $k > 0$) for the schematic formula $\mathbf{B}[t_1, \cdots, t_k]$ *i.e.* for the term having the shape of $\mathbf{B}$ but where each of the variables x_i is replaced by the term t_i. $\mathbf{A}$ may introduce new symbols not in the language while $\mathbf{B}$ must be a formula of the language defined up to the point of its introduction, this includes those formulas given by the syntax as well as formulas that may include previously defined symbols.

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.

Instances of defined terms can be replaced by their definitions replacing the arguments in the left side of the definition into the right side. The process of "replacement" is fundamental and is called *substitution*. In following chapters, we will carefully define substitution (as an algorithm) for propositional and then predicate logic.

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.

Example 1.2. In mathematics, we define functions all the time by introducing their names and arguments and giving the right hand side "definition." *e.g.*

$$f(k) = 2^k + k$$

$$exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} \quad x \in \mathbb{R}$$

The second one is interesting since the variable n is "bound" by the summation operator ($\sum$). We call operators that "bind" variables *binding operators*. In the

definition of *exp*, we must be careful when we substitute in something for x – if the term being replaced for x contains the variable n in it, we first must rename n's on the right side of the definition and then go ahead with substitution. This process is called *capture avoiding substitution* and is covered in some detail in Chapter **??**.

1.4 Semantics

Semantics associates meaning with syntax. Formal semantics (the kind we are interested in here) is given by defining a mathematical mapping from syntax (think of syntax as a kind of data-structure) to some other mathematical structure. This mapping is called the *semantic function* or *interpretation* ; we will use these terms interchangeably. When possible, formal languages are given *compositional semantics*. The meaning of a syntactic structure depends on the meanings of its parts.

Before a semantics is given, 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.

1.4.1 Definition by Recursion

In practice, semantic functions are defined by recursion on the structure of the syntax being interpreted. We expect students have already encountered recursion. To define an interpretation, an equation must be given for possible construct(or) in the grammar; one equation for each alternative. So, it is easy enough to tell if the definition is complete; it must have as many cases as there are constructors. The alternatives that do not contain references to the syntactic class defined by the grammar are *base cases* and these cases in the definition of the semantic function are not recursive. The alternatives that do contain subparts of the same type as the syntactic class being defined are *inductive alternatives* . The semantic equations for these cases of the semantic function typically call the semantic function (recursively) on the inductive parts; this is where the recursion comes in. It may appear that a recursive definition is circular, but because we restrict the inductive parts of the structure to be "previously constructed", we guarantee that eventually we will have reduced

the complex parts down to one of the base cases. In this way, computation by recursion is guaranteed to terminate.

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 as $\mathbf{T}$ and 0 is interpreted as $\mathbf{F}$ 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].

[4] Perhaps interestingly, in the logic of CMOS circuit technology, this seemingly backward 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.

[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.

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 $\mathbf{0}$ 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.

$$\begin{aligned} &[\![s\mathbf{0}]\!] \\ &= [\![\mathbf{0}]\!] + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

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

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

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).

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

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

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 calculations (equational style derivations) that show how the equations can be used to compute meanings.

$[\![\texttt{if } \mathbf{T} \texttt{ then } \mathbf{F} \texttt{ else } \mathbf{T} \texttt{ fi}]\!]$
$\langle\langle$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 calculations, 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.

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

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 Chapter **??** on propositional 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 by recursion on the (syntactic) structure of the first argument as follows:

Definition 1.6 (Append)

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

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 the result is obtained by consing 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*.

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

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

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

Remark 1.1 (Infix Notation for Append) The append operation is so commonly used that many functional programming languages include special infix notation. In the Haskell programming language [**?**] the infix notation is ++, in the ML family of programming languages append is written @. We will write $m{+}{+}n$ for $\mathit{append}(m,n)$.

Using this infix notation, we rewrite the computation above as follows:

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

We will use the more succinct notation for lists from now on and the infix notation, 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.

$$\begin{aligned} &[\,]{+}{+}[a,b] \\ &= [a,b] \end{aligned}$$

We will discuss lists and operations on lists as well as ways to prove properties about lists in some depth in Chapter 11. For example, the rules for append immediately give $\mathit{append}((,[\,]),M) = M$, but the following equation is a theorem as well $\mathit{append}((,M),[\,]) = M$. 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 Possibilities for 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 [37, 33, 39] and the language Haskell [**?**]. Scheme is also useful in this way [16]. All three, ML, Haskell and Scheme are languages in the family of functional programming languages. Of course we can define term structures in any modern programming language, but the functional 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 [16]. The excellent book [9]presents most of the material presented here in the context of the Haskell programming language.

Part I

Logic

Kurt Godel

Kurt Gödel (1906 – 1978) was one of the greatest minds of the 20th century. His famous incompleteness theorem changed, in a deep way, the conception of mathematics.

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[1] Discourses[2] II xxv.

Propositional logic is the most basic form of logic. It takes as primitive, 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 [55] definition of propositions as truth functional.

In the following sections of this chapter we define the syntax of propositional formulas, we describe the semantics, present a sequent proof system an proofs and finally discuss equational style reasoning in propositional logic.

2.1 Syntax of Propositional Logic

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. We think of the syntactic forms (formulas) as trees (syntax trees) whose possible shapes are given by a *grammar* for propositional formulas. Even though familiar symbols appear in formulas (whose meaning you may already know) they are

[1] Epictetus was an ancient philosopher (50–130 A.D.) of the Stoic school in Rome.
[2] Translated by Jonathan Barnes in his book [3].

uninterpreted here; they are simply tree-like structures waiting for a semantics to be applied.

2.1.1 Formulas

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

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

Note that the fact that the set $\mathcal{V}$ is infinite is unimportant since no individual 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. There 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 $\wedge, \vee, \Rightarrow$ which we read as *and*, *or* and *implies* respectively.

Definition 2.2 (Propositional Logic syntax) 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.

Another view of syntax* Readers familiar with a programming language that supports high-level datatypes will be familiar with constructor functions.

If we were to implement a datatype representing formulas we would provide a set of constructor functions, one for each kind of formula.

They can be described by giving their type signatures[3]. If P is the type of propositional formulas and V is the type of variables, 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 labeled 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 labeled 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 connectives*. Their arguments are pairs of previously constructed propositional formulas and so they all have the signature $(\mathcal{P} \times \mathcal{P}) \to \mathcal{P}$.

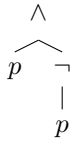
Here are some formulas represented in different ways.

No.	Linear Form	Constructor Form
i.	$\bot$	mk_bot
ii.	p	$mk\_var(p)$
iii.	$\neg p$	$mk\_not(mk\_var(p))$
iv.	$p \wedge \neg p$	$mk\_and(mk\_var(p), mk\_not(mk\_var(p)))$
v.	$\neg\bot \Rightarrow p$	$mk\_implies(mk\_not(mk\_bot), mk\_var(p))$
vi.	$((p \wedge \bot) \Rightarrow (p \vee \neg q))$	$mk\_implies(mk\_and(mk\_var(p), mk\_bot),$ $mk\_or(mk\_var(p),$ $mk\_not(mk\_var(q))))$

The syntax trees for the last four of these examples are drawn as follows:

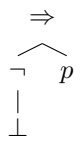
iii.)

iv.)

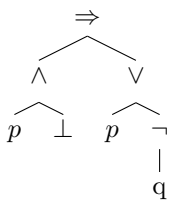


[3]A signature $f : A \to B$ says f is a function for type A to type B. A signature of the form $p : A \times B$ says p is a tuple whose first element is of type A and whose second element is type B.

v.)



vi.)



2.1.2 Definitions: Extending the Language

As discussed in Sect. 1.1.3.3, definitions allow for the introduction of new symbols into the language by describing them in terms of the existing language. Adding new symbols can be a significant notional convenience but it does not extend the expressiveness of the language since definitions are given in terms of the existing language.

A useful connective we have not included in our base syntax for propositional logic is for the if-and-only-if connective.

If-and-only-if

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

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

True $\top$ The syntax includes a constant $\bot$ which, when we do the semantics, will turn out to denote *false*; but we do not have a constant corresponding to *true*. We define it here.

Definition 2.4 (Top) We define a new constant "$\top$" (say *top*) as follows: $\top \stackrel{\text{def}}{=} \neg\bot$.

2.1.3 Substitutions*

A *substitution* is a means to map formulas to other formulas by uniformly replacing all occurrences individual variables with a formula. For example, given the formula $(p \wedge q) \Rightarrow p$ we could substitute any formula for p or any formula for q. Say we wanted to substitute $(r \vee q)$ for p then we write the following:

$$subst(p, \lceil r \vee q \rceil)\lceil (p \wedge q) \Rightarrow p \rceil$$

Recalling that syntax is best

2.1.4 Exercises

2.2 Semantics

Semantics gives meaning to syntax. The style of semantics presented here was first presented by Alfred Tarski in his paper[49] on truth in formalized languages which was first published in Polish in 1933.

If I asked you to tell me whether the expression $x + 11 > 42$ it true, you'd probably tell me that you need to know what the value of x is. So, the meaning of $x + 11 > 42$ depends on the values assigned to x. Similarly, if I asked you if a formula was true (say $p \wedge q$) you'd tell me you need to know what the values of p and q are. The meanings of a formula depend on the values assigned to variables in the formula.

In the following sections we introduce the set of Boolean values and we formalize the notion of an assignment. We present the semantics for propositional logic in the form of a valuation function that, given an assignment and a formula returns **T**or **F**. The valuation function is then used as the basis to describe the method of truth tables. Truth tables characterize *all* the possible meanings of a formula. This gives us a semantics for propositional logic.

2.2.1 Boolean values and Assignments

Definition 2.5 (Booleans) The two element set

$$\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$$

is called the Boolean[4] set, and its elements (**T** and **F**) are called Boolean values.

Note that any two element set would do, as long as we could distinguish the elements from one another.

When we ask if a formula is true, we are asking whether it evaluates to **T** when it is interpreted with respect to some kind of structure. For an arithmetic expression like the one in the example give above ($x + 11 > 42$) – the structure would have to (at least) indicate the integer value associated with the variable x. For propositional logic, the structure binding Boolean values to variables is called an assignment.

Definition 2.6 (assignment) An *assignment* is a function that maps propositional variables to one of the Boolean values **T** or **F**. Assignments have the following type:

$$\mathcal{V} \to \{\mathbf{T}, \mathbf{F}\}$$

We will use the symbols $\alpha, \alpha', \hat{\alpha}, \alpha_1, \alpha_2, \cdots$ to denote assignments.

Remember that the set of propositional variables $\mathcal{V}$ is infinite but any individual formula contains a finite number of them. We don't need to know the

[4]The Booleans are eponymous for George Bool (1815 – 1864) the English mathematician who first presented logic in symbolic or algebraic form.

value of of every variable to determine the value of a formula, we simply need to know the values of the variables occurring in the formula. This will allow us to specify assignments by enumerating the cases for each variable that does occur. We "don't care" what values the assignment gives to variables not in the formula under consideration, they will have no effect on the outcome of the valuation.

Counting 2.1 (Assignments) For a formula ϕ containing k distinct variables where $k > 0$, there are 2^k possible assignments (disregarding all the variables in $\mathcal{V}$ that are not among the k variables that do occur.)

Example 2.1. For the formula consisting of the single variable p there are $2^1 = 2$ possible assignments.

	p
α_0	**F**
α_1	**T**

To read the table, the assignment name is in the left column and the variables are listed across the top. Thus, $\alpha_0(p) = \mathbf{F}$ and $\alpha_1(p) = \mathbf{T}$.

For the formula $p \vee \bot$ having one occurrence of the variable p, $k = 1$ and there are $2^1 = 2$ possible assignments which were the ones just given.

The formula $(p \vee q) \Rightarrow p$ has two distinct variables p and q and so has $2^2 = 4$ different assignments.

	p	q
α_0	**F**	**F**
α_1	**F**	**T**
α_2	**T**	**F**
α_3	**T**	**T**

2.2.2 The Valuation Function

Definition 2.7 (valuation) 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 2.8 (valuation function)

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

$$\begin{array}{lll} val(\alpha, \phi \Rightarrow \psi) & = & \textbf{not}(val(\alpha, \phi)) \ \textbf{or} \ val(\alpha, \psi) \\ val(\alpha, \phi \Leftrightarrow \psi) & = & val(\alpha, \phi \Rightarrow \psi) \ \textbf{and} \ val(\alpha, \psi \Rightarrow \phi) \end{array}$$

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 $\mathcal{P}$..

Example 2.2. As an example, suppose we define an assignment α as follows:

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

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

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

Consider the valuation of another formula under same assignment.

Example 2.3.

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

Definition 2.9 (satisfies) 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 2.10 (failsifies) 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 2.11 (valid) 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.*

Example 2.2 shows that $\alpha \models ((p \Rightarrow q) \vee r)$ and Example 2.3 shows $\alpha \not\models ((p \Rightarrow q) \vee q)$.

2.2.3 Truth Table Semantics

The semantics for propositional logic maps a formula (a syntax tree) to its meaning. The meaning of a propositional formula depends only on the meaning of its parts. Based on this, we say the semantics of propositional logic are *compositional*. 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.

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

Definition 2.13 (falsifiable) A propositional formula is *falsifiable* if the column under the principal connective is false in any row of the truth table.

Definition 2.14 (valid) 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 2.15 (contradiction) 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 did not include a constant in the base syntax for the language of propositional logic whose meaning is **T**; however, we defined the constant $\top$ (see Definition 2.4). The following truth table shows that this defined formula always has the 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 $\mathcal{P}$.

2.2.4 Exercises

2.3 Proof Theory

An alternative to using semantics to determining whether a formula is valid is to build a proof. In this section we present a formal[5] system of proofs. It is fair to think of a proof as an argument that proceeds in steps. At each step there is information that has been been accrued in the process of building the proof to that point and there are the goals left to be shown before the proof is complete. Together, the information accrued so far and the outstanding goals determine the *state* of the proof. In the proof system presented here, this state is represented in a structure called a *sequent*. Although a narrative describing a proof is necessarily linear, it turns out that proofs are formally represented by tree structures. Nodes of the tree are sequents recording the state of proof at that point and the number of edges from a node and the form of the children nodes depend on which proof rule was applied at that node.

A key idea is that formal proofs are a kind of tree-like data-structure and we can determine whether a tree really is a proof bu checking local conditions on the nodes. In this section, we make this idea concrete.

[5] By formal, we mean that we give a detailed mathematical presentation with enough detail for a software implementation.

2.3.1 Sequents

Gerhard Gentzen

Gerhard Gentzen (1909–1945) was a German logician who, in his short years, made astounding contributions to the foundations of mathematics, logic and proof theory. In the same paper [18] he invented both natural deduction proof system as well as the sequent proof systems we use here.

Sequents are pairs of lists of formulas 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 [18].

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 $List_{\mathcal{P}}$[6].

Definition 2.16 (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.

Remark 2.1 (On Sequent notation) A standard notational convention is to write the sequent $\langle \Gamma, \Delta \rangle$ in the form $\Gamma \vdash \Delta$. The symbol "$\vdash$" is called *turnstile*.

If $\Gamma = [p, p \Rightarrow q]$ and $\Delta = [q, r]$ we will omit the left and right brackets and simply write $p, p \Rightarrow q \vdash q, r$ instead of $[p, p \Rightarrow q] \vdash [q, r]$. In the specifications of proof rules we will often want to focus on some formula in the antecedent (left side) or consequent (right side) of a sequent. We do this by introducing some notation for destructuring lists. Recall the definition of append (Def. 11.2) from Chapter 1 and the infix notation. We write $m{+}{+}n$ to denote the list constructed by appending (concatenating) the lists m and n together *e.g.* $[3,4]{+}{+}[1,2,3] = [3,4,1,2,3]$. If Γ_1, Γ_2 are formula lists and ϕ is a formula, we write Γ_1, ϕ, Γ_2 to denote the list $\Gamma_1{+}{+}([\phi]{+}{+}\Gamma_2)$. By these conventions, Γ_1, ϕ, Γ_2 is a list of formulas in which the formulas in Γ_1 occur in order and to the left of ϕ and the formulas in Γ_2 occur in order and to the right of ϕ.

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

2.3.2 Semantics of Sequents

Informally, a sequent $\Gamma \vdash \Delta$ is valid if, assuming all the formulas in Γ are true, then at least one formula in Δ is. Validity for propositional formulas was defined in terms of satisfiability under all assignments and likewise for sequents.

Definition 2.17 (Sequent satisfiability) A sequent $\Gamma \vdash \Delta$ is *satisfiable* under assignment α if α make all the formulas in Γ true then α makes at least one formula in Δ true. In this case we write $\alpha \models \Gamma \vdash \Delta$.

Given an assignment mapping propositional variables to Boolean values, we can compute the valuation of a sequent under that assignment by translating the sequent into a formula of propositional logic and then using the ordinary valuation function given in Def. 2.7.

The translation of the sequent $\Gamma \vdash \Delta$ into a formula is based on the following ideas:

i.) All the formulas in the antecedent Γ are true if and only if their conjunction is true as well.

ii.) At least one formula in the consequent Δ is true if and only if the disjunction of the formulas in Δ is true.

iii.) Implication models the notion *if-then*.

We will formalize the translation once we have defined operations mapping lists of formulas to their conjunctions and disjunctions.

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:

Definition 2.18 (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.

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

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

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

Exercise 2.1. Verify that $((\phi \wedge \neg\bot) \Leftrightarrow \phi)$ is a tautology.

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 2.19 (Disjunction over a list) The function which creates a disjunction of all the elements in a list is defined by recursion on the structure of the list and is given by the following two equations.

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

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

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

Exercise 2.2. Verify that $((\phi \vee \bot) \Leftrightarrow \phi)$ is a tautology.

An informal semantic argument for this choice to represent the disjunction of 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.

Formal Semantics for 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 2.20 (Formula interpretation of a sequent)

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

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 2.21 (Sequent Valuation) Given an assignment α and a sequent $\Gamma \vdash \Delta$ we say α satisfies $\Gamma \vdash \Delta$ if the following holds:

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

In this case we write $\alpha \models \Gamma \vdash \Delta$.

This gives the following

Definition 2.22 (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

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

To exercise these definitions we now consider the cases 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**

2.3.3 Sequent Schemas and Matching

So far we have used sequents as *schemas* that can be filled in with lists of actual formulas taking the place of the meta-variables Γ and Δ. You can think of them as templates waiting to be filled in. A way to fill in the template is by a substitution function that maps the meta-variables $(\phi, \psi, \Gamma, \Delta)$ in the syntax of a schematic sequent to the appropriate kind of things (formulas or formula lists) depending on the type of meta-variable.

Example 2.4. Consider the following schematic sequent:

$$\mathcal{S}: \quad \Gamma_1, \phi \Rightarrow \psi, \Gamma_2 \vdash \Delta$$

The following substitution (call it σ) specifies how to map meta-variables occurring in $\mathcal{S}$ to lists of formulas and formulas.

$$\begin{array}{l}\sigma(\Gamma_1) = [p] \\ \sigma(\Gamma_2) = [] \\ \sigma(\Delta) = [r] \\ \sigma(\phi) = p \vee q \\ \sigma(\psi) = r\end{array}$$

We apply a substitution to a schema $\mathcal{S}$ by decomposing the schema into its syntactic parts and recursively applying the substitution to those parts.

$$\begin{array}{l}\sigma(\Gamma_1, \phi \Rightarrow \psi, \Gamma_2 \vdash \Delta) \\ \quad = \sigma(\Gamma_1), \sigma(\phi \Rightarrow \psi), \sigma(\Gamma_2) \vdash \sigma(\Delta) \\ \quad = [p], \sigma(\phi) \Rightarrow \sigma(\psi), [] \vdash [r] \\ \quad = [p], (p \vee q) \Rightarrow r, [] \vdash [r]\end{array}$$

By our notational conventions for sequents the resulting sequent is written as follows:

$$p, (p \vee q) \Rightarrow r \vdash r$$

Definition 2.23 (matching) 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.

2.3.4 Proof Rules

Definition 2.24 (Proof Rule Schemata) 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*.

Definition 2.25 (Admissable Rules) A proof rule is *admissible* if, whenever the premises of the rule are valid the conclusion is valid.

Proof rules are schemas (templates) 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 (or concrete) sequent might (or might not) match. By an actual sequent, we mean a sequent that contains no meta-variables (*e.g.* it contains no Γs or Δs, or ϕs or ψs but is composed of formulas int he language of propositional logic.)

Structural Rules*

The semantics of sequents (given in Def. 2.20) gives them lots of structure. There are some non-logical rules that sequents obey. These rules are admissible based simply on the semantics, regardless of the formula instances occurring in the antecedent and consequent. They essentially express the ideas that the order of the formulas does not affect validity and neither does the number of times a formula occurs in the antecedent or the consequent.

It turns out that in the propositional case, it is never required that a structural proof rule be used to find a proof. Once the quantifiers have been added in Chapter 4, some proofs will require the use of these rules.

Weakening Weakening says that if $\Gamma \vdash \Delta$ is valid, then adding formulas to the left or right side does not affect validity.

Proof Rule 2.1 (WL)

$$\frac{\Gamma \vdash \Delta}{\phi, \Gamma \vdash \Delta} \text{ (WL)}$$

Proof Rule 2.2 (WR)

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta} \text{ (WR)}$$

Contraction The contraction rules essentially say that, if a formula occurs on the left or right side of a sequent, the number of times it occurs matter. You are free to make as many copies of a formula as you wish.

Proof Rule 2.3 (CL)

$$\frac{\Gamma_1, \phi, \Gamma_2, \phi, \Gamma_3 \vdash \Delta}{\Gamma_1, \phi, \Gamma_2, \Gamma_3 \vdash \Delta} \text{ (CL)}$$

Proof Rule 2.4 (CR)

$$\frac{\Gamma \vdash \Delta_1, \phi, \Delta_2, \phi, \Delta_3}{\Gamma \vdash \Delta_1, \phi, \Delta_2, \Delta_3} \text{ (CR)}$$

Permutation The permutation rules say that reordering of formulas in the antecedent and consequent do not affect validity. The reordering is expressed in terms of locally swapping the order of formulas in the antecedent and consequent of the sequent.

Proof Rule 2.5 (PL)

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

Proof Rule 2.6 (PermR)

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

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:

Proof Rule 2.7 (Ax)

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

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

Proof Rule 2.8 ($\bot$Ax)

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

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:

Proof Rule 2.9 ($\wedge$R)

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

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.

Proof Rule 2.10 ($\wedge$L)

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

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.

Proof Rule 2.11 ($\vee$R)

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

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.

Proof Rule 2.12 ($\vee$L)

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

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.

Proof Rule 2.13 ($\Rightarrow$R)

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

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.

Proof Rule 2.14 ($\Rightarrow$L)

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

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.

Negation Rules

The negation $\neg\phi$ can be viewed as an abbreviation for the formula $\phi \Rightarrow \bot$; this claim can be checked by writing out the truth table. Based on this relationship, the proof rule for negation is related to that of implication (see above.)

Proof Rule 2.15 ($\neg$R)

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

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.

Proof Rule 2.16 ($\neg$L)

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

2.3.5 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 2.26 (proof tree) 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

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

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

$$\frac{S_1 \qquad S_2}{S}\ (N)$$

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

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

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.

As characterized in [43], the goal directed process of building proofs, *i.e.* working backward from the goal, is a reductive process as opposed to the deductive process which proceeds forward from the axioms.

We present some examples.

Example 2.5. 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 2.6. 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:

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

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.

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

In this case, both incomplete branches are instances of the axiom rule. The matches for the left and right branches are, respectively:

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

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

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

2.3.6 Some Useful Tautologies

Recall that the symbol $\top$ is an abbreviation for the true formula $\neg\bot$.

Theorem 2.1.

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

xviii. $(p \Rightarrow q) \vee (q \Rightarrow p)$

Exercise 2.3. Give sequent proofs showing that the tautologies in Thm 2.1 hold.

2.3.7 Exercises

2.4 Metamathematical Considerations*

Metamathematics is the application of mathematical methods to study the mathematics itself. In this chapter, we have presented syntax, semantics and a sequent proof system for propositional logic. We have mentioned that proofs provide an alternative path to determining validity. But we have not said how we know that the proof system presented here coincides with the semantics. The relationship of the proof system to the semantics is given by *soundness* and *completeness* results. Also, from a computational stance, we'd like to know if there are algorithms and what (their computational complexity might be) for deciding whether a formula is valid or not; such an algorithm is called a *decision procedure*.

2.4.1 Soundness and Completeness

The properties that relate a proof systems to the corresponding semantic notion are soundness and completeness. Recall that the semantic notion of validity for a formula ϕ is denoted $\models \phi$. It is traditional to assert that a formula ϕ is provable by writing $\vdash \phi$, since we have used $\vdash$ in our notation for sequents, we will write $\|\!-\phi$ to mean there is a proof of ϕ.

Soundness Soundness is the property that claims that every provable formula is semantically valid. An unsound proof system would not be of much use, if we could prove any theorem which was not valid, we could prove all theorems because $\bot \vdash \phi$ for an arbitrary formula ϕ.

Theorem 2.2 (Soundness) For every propositional formula ϕ, if $\|\!-\phi$ then $\models \phi$.

We do not have the tools or methods yet to prove this theorem. We have informally argued for the admissibility of the individual proof rules and these individual facts can be combined to show soundness. The proof method used to prove soundness is based on an induction principle that follows the structure of a formula. These methods will be introduced in a Chapter **??**.

Completeness Completeness is the property that all valid formulas are provable. If a proof system is complete, it captures all of the valid formulas. It turns out that there are mathematical theories for which there is no complete proof system, propositional logic is not one of them.

Theorem 2.3 (Completness) For every propositional formula ϕ, if $\models \phi$ then $\|\!-\phi$.

Again, we do not yet have a proof method that will allow us to prove completeness; by the end of the book we will.

2.4.2 Decidability

A set is *decidable* if there is an algorithm to decide if an element is in the set. To talk about the decidability of a logic, we first have to describe it as a set or collection of formulas.

Definition 2.27 (Theory) Given a language $\mathcal{L}$ and a semantic notion of validity on $\mathcal{L}$ (say $\models$), the *theory* of $\langle \mathcal{P}, \models \rangle$ is the collection of all valid formulas in $\mathcal{P}$. We write this as follows:

$$\mathbf{Th}(\mathcal{L}, \models) = \{\phi \mid \models \phi\}$$

Thus, the theory of propositional logic $\mathbf{Th}\langle \mathcal{P}, \models \rangle$ is the collection of all formulas in the language of propositional logic ($\mathcal{P}$) that are semantically valid.

Definition 2.28 (Decidability) Given a set S, a subset T of S is *decidable* if there is an algorithm if to determine if an arbitrary element of S is in T. More formally, if there is an algorithm $d : S \to \mathbb{B}$ such that $d(x) = \mathbf{T}$ if and only if $x \in T$.

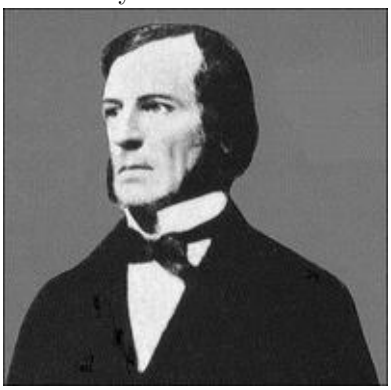
Theorem 2.4 (Propositonal Logic is Decidable) The theory of $\mathbf{Th}(\mathcal{P}, \models)$ is decidable.

2.4.3 Exercises

Chapter 3

Boolean Algebra and Equational Reasoning*

Propositional logic has an algebraic form first investigated by George Boole in 1840's. Arguably, Boole's algebraic and symbolic approach to logic was the first truly significant step forward in the development of logic since Aristotle; Boole's algebra was symbolic.



George Boole

George Boole (1815-1864) was an English mathematician and logician. His father was a shoemaker and George was largely self-taught. He became a schoolmaster at age 16 and published his first mathematical paper at age 23. See [**?**] in [10] for more on Boole's life.

3.1 Boolean Algebra

In the previous chapter 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.

3.1.1 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 3.1. 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 3.1. 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 3.1.

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

Theorem 3.1. The following three properties hold[1].

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 3.2. 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 3.3. 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 3.2. Since $5 \equiv 3(\mathrm{mod}\, 2)$ and $10 \equiv 98(\mathrm{mod}\, 2)$

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

[1] We will see later in Chapter 7 that relations having these properties are called equivalence relations

To see this note the following:

$$5 + 10 = 15 \text{ and } 15 = 7 \cdot 2 + 1, \text{ so } 5 + 10 \equiv 1 (\mathrm{mod}\, 2)$$
$$3 + 98 = 101 \text{ and } 101 = 50 \cdot 2 + 1, \text{ so } 3 + 98 \equiv 1 (\mathrm{mod}\, 2)$$
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 3.2. 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 3.4. The following identities hold.

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

3.1.2 Translation from Propositional Logic

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

We define the translation (denoted $\mathcal{M}[\![\phi]\!]$) which maps propositional formulas to algebraic formulas by recursion on the structure of the formula ϕ.

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.

3.1.3 Falsity

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

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

3.1.4 Variables

Propositional variables are just mapped to variables in the algebraic language

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

3.1.5 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.

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

3.1.6 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:

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

3.1.7 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

3.1.8 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 3.1. Verify this identity by using a truth table.

By the translation we have so far

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

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.

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

3.1.9 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

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

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,

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

3.1.10 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 3.3. 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:

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

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.

3.1.11 Notes

In modern times, the Boolean algebras have been investigated abstractly [25].

3.2 Equational Reasoning

The connective $\Leftrightarrow$ turns out to have the following properties.

Theorem 3.5. If ϕ and ψ and *varpsi* are propositional meta-variables, the following theorems hold.

$$\begin{array}{ll} 1.) & \phi \Leftrightarrow \phi \\ 2.) & (\phi \Leftrightarrow \psi) \Leftrightarrow (\psi \Leftrightarrow \phi) \\ 3.) & ((\phi \Leftrightarrow \psi) \wedge (\psi \Leftrightarrow R)) \Rightarrow (\phi \Leftrightarrow R) \end{array}$$

Exercise 3.2. Prove Thm. 3.5.

We shall see in Chap 7 that operations like $\Leftrightarrow$ that have properties (1), (2) and (3) behave like an equality. If you interpret "$\Leftrightarrow$" as "$=$" and ϕ, ψ and R as numbers you will see this. Property (1) shows $\Leftrightarrow$ is *reflexive*, property (2) shows it is *symmetric* and property (3) shows it is *transitive*.

3.2.1 Complete Sets of Connectives

Definition 3.3 (Complete set) A set of connectives, $\mathcal{C}$,

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

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

Example 3.4. 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.

1.) $\bot \overset{\langle i \rangle}{\Longleftrightarrow} \neg\neg\bot \overset{\langle \top\, def \rangle}{\Longleftrightarrow} \neg\top \overset{\langle xiv \rangle}{\Longleftrightarrow} \neg(\phi \vee \neg\phi)$

2.) $(\phi \wedge \psi) \overset{\langle i \rangle}{\Longleftrightarrow} \neg\neg(\phi \wedge \psi) \overset{\langle iv \rangle}{\Longleftrightarrow} \neg(\neg\phi \vee \neg\psi)$

3.) $(\phi \Rightarrow \psi) \overset{\langle iii \rangle}{\Longleftrightarrow} \neg\phi \vee \psi$

4.) $(\phi \Leftrightarrow \psi) \overset{\langle \Leftrightarrow\, def. \rangle}{\Longleftrightarrow} (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$
$\overset{\langle iii \rangle}{\Longleftrightarrow} (\neg\phi \vee \psi) \wedge (\psi \Rightarrow \phi)$
$\overset{\langle iii \rangle}{\Longleftrightarrow} (\neg\phi \vee \psi) \wedge (\neg\psi \vee \phi)$
$\overset{\langle 2 \rangle}{\Longleftrightarrow} \neg(\neg(\neg\phi \vee \psi) \vee \neg(\neg\psi \vee \phi))$

Exercise 3.3. Prove that the set $\{\neg, \wedge\}$ is complete for $\{\bot, \neg, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$. You'll need to give definitions for $\bot$, *vee*, $\Rightarrow$ and $\Leftrightarrow$ in terms of $\neg$ and $\wedge$ and then prove that your definitions are correct.

Exercise 3.4. Prove that the set $\{\bot, \Rightarrow\}$ is complete for $\{\bot, \neg, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$.

Chapter 4

Predicate Logic

> *Since by the aid of speech and such communication as you receive here you must advance to perfection, and purge your will and correct the faculty which makes use of the appearances of things; and since it is necessary also for the teaching (delivery) of theorems to be effected by a certain mode of expression and with a certain variety and sharpness, some persons captivated by these very things abide in them, one captivated by the expression, another by syllogisms, another by sophisms, and still another by some other inn (πανδοκεον) of this kind; and there they stay and waste away as if they were among the Sirens.* Epictetus Discourses [13] II xxiii

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 quantification*, they are different ways of asserting that everything satisfies some specified property. The second two forms are called *existential quantification*, 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 $(\exists x. \cdots)$.

4.1 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 4.1 (arity) A function is called *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 4.2 (Boolean valued function) A function is *Boolean-valued* if its range is the set $\mathbb{B}$.

Definition 4.3 (predicate) A *predicate* is a n-ary Boolean-valued function over some domain of input.

Example 4.1. 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:

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

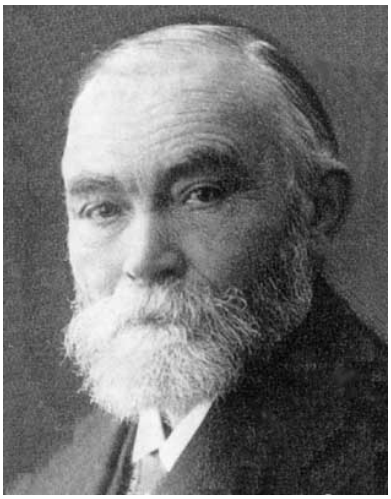
We could define a unary predicate which is true when its argument is equal to 0 and is false otherwise:

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

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.



Gottlob Frege

Gottlob Frege (1948 – 1925) was a German mathematician, logician and philosopher. He made the largest advance in logic since Aristotle by his discovery of the notion of and formalization of quantified variables. Frege was also a founder of analytic philosophy and the philosophy of language. A fundamental contribution there is that words obtain meanings in the context of their usages.

4.2 The Syntax of Predicate Logic

Predicate logic formulas are constructed from two sorts of components: terms and formulas which may contain terms.

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.

4.2.1 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 parameters of the definitions that use it.)

4.2.2 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.

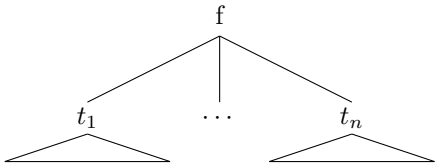
Definition 4.4 (Terms) *Terms* are defined over a set of function symbols $\mathcal{F}$ are given by the following grammar:

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

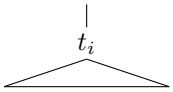
where:

$\mathcal{F}$ is a 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 parametrized 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 4.2. 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:

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

4.2.3 Formulas

Definition 4.5 (Predicate Logic Formula) *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 parametrized 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 4.3. 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 4.4. 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 4.5. A version of the symmetric property for le on numbers can be stated 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 4.6. 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 4.1. 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.

4.3 Substitution

Substitution is the process of replacing a variable by some more complex piece of syntax such as a term or a formula. Readers are already familiar with this process, though there are is some added complexity that results from notations that bind variables *e.g.* summation ($\Sigma_{i=j}^{k} f(i)$), product ($\Pi_i = j^k f(i)$), integral ($\int_a^b f(x)dx$), and the quantifiers of predicate logic ($\forall x.\phi(x)$ and $\exists x.\phi(x)$).

As an example of a simple substitution (without considerations related to bindings) consider the following example.

Example 4.7. If we consider the polynomial $2x^2-3x-1$ and say $x = y^2+2y+1$ then, by substitution we know

$$2x^2 - 3x - 1 = 2(y^2 + 2y + 1)^2 - 3(y^2 + 2y + 1) - 1$$

Here, we simply replaced *all* the occurrences of x in the polynomial $2x^2 - 3x - 1$ by the polynomial $y^2 + 2y + 1$. Of course, the rules of algebra would allow us to simplify the resulting expression further; but the process of substitution is one of replacing all the x's by $y^2 + 2y + 1$.

Summation is

4.3.1 Bindings and Variable Occurrences

Variable Occurrences

Definition 4.6 (occurs (in a term)) A variable x *occurs in term* t if and only if $x \in \texttt{occurs}(t)$ where `occurs` is defined by recursion on the structure of the term 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 4.7 (occurs (in a formula)) 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 4.8 (binding operator) 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. Different 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 [11].

Example 4.8 () 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 $Q(x, y)$.

4.3.2 Free Variables

Definition 4.9 (free occurrence (in a term)) 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 4.10 (free occurrence (in a formula)) 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

Bound variables can only occur in formulas; this is because there are no binding operators in the language of terms.

Definition 4.11 (bound occurrence) A variable x *occurs bound in formula* ϕ if and only if $x \in \mathtt{BV}(\phi)$ where $\mathtt{BV}$ 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 ψ iff it contains a sub-formula of the form $\forall x.\phi$ or $\exists x.\phi$.

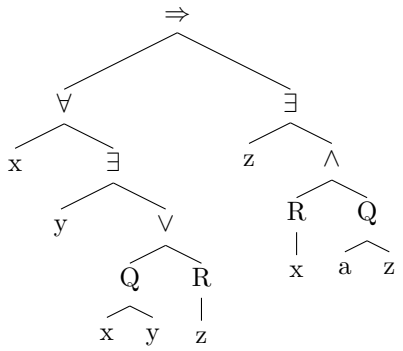
Discussion

The algorithms for computing the free variables and bound variables of a formula are given by recursion on the structure of the formula. By drawing a syntax tree, it is easy to see which variables are free and which are bound. Choose a variable in the tree. It is bound if it is the left child of a quantifier or if, traversing the tree to its root a quantifier is encountered having the same variable as a left child. A variable is free if it is not the left child of a quantifier or if the path from the variable to the root of the syntax tree does not include a quantifier whose left child matches the variable.

Example 4.9. Consider the formula

$$(\forall x.\exists y.Q(x,y) \vee R(z)) \Rightarrow \forall z.R(x) \wedge Q(a,z)$$

The syntax tree appears as follows:



We can refer to variables by their left to right position in the formula. The leftmost x in the formula is bound because, in the syntax tree, it is a left child of the quantifier $\forall$. Similarly, the same holds for the leftmost y. The second occurrence of x in the formula is bound because on the path to the root of the syntax tree passes a $\forall$ quantifier whose left child is also x. The second y in the formula is bound because the path to the root passes an $\exists$ quantifier whose left child is a y. The first occurrence of the variable z in the formula is free because no quantifier on the path to the root has a z as a left child. The second z occurring in the formula is bound because it is the left child of an $\exists$ quantifier. The third x is free. The constant symbol a is not a variable and so is neither free nor bound. The last z in the formula is bound by the $\exists$ quantifier above it in the syntax tree.

Remark 4.2. Note that there are formulas where a variable may occur both free *and* occur bound, *e.g.* x in the example above. As another example where this happens, 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.

4.3.3 Capture Avoiding Substitution*

Substitution is perhaps the most basic operation in mathematics and it is often 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 [28], Quine got it wrong in the first edition of his logic book [44], 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[2] lambda calculus [6] 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 – *i.e.* we just search for the variable to be replaced by a term and when one that matches is found, it is replaced.

Definition 4.12 (substitution (for terms)) *Substitution* is defined as follows:

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

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 4.13 (Capture Avoiding Substitution) *Capture avoiding sub-*

[2]Alonzo Church was an American mathematician and logician who taught at Princeton University. Among other things, he is known for his development of λ-calculus, a notation for functions that serves as a theoretical basis for modern programming languages.

stitution for formulas is defined as follows:

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

4.4 Proofs

4.4.1 Proof Rules for Quantifiers

4.4.2 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 [3].

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

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.

4.4.3 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 4.14 (existential witness) The term t substituted for the bound variables in an $\exists$R-rule is called the *existential 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.

4.4.4 Some Proofs

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

[3] 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 4.10. 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.

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

Example 4.11. In the case of propositional logic we did not need to apply any of the structural rules however, they may be required in the case of the quantifier rules. Consider the following theorem.

$$\exists x.P(x) \Rightarrow \forall x.P(x)$$

Here is a sequent proof whose first step is to copy the formula using the rule for contraction on the right.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(a), P(y) \vdash P(y), \forall x.P(x)}\,(\text{Ax})}{P(a) \vdash P(y), P(y) \Rightarrow \forall x.P(x)}\,(\Rightarrow\text{R})}{P(a) \vdash P(y), \exists x.P(x) \Rightarrow \forall x.P(x)}\,(\exists\text{R})}{P(a) \vdash \forall x.P(x), \exists x.P(x) \Rightarrow \forall x.P(x)}\,(\forall\text{R})}{\vdash P(a) \Rightarrow \forall x.P(x), \exists x.P(x) \Rightarrow \forall x.P(x)}\,(\Rightarrow\text{R})}{\vdash \exists x.P(x) \Rightarrow \forall x.P(x), \exists x.P(x) \Rightarrow \forall x.P(x)}\,(\exists\text{R})}{\vdash \exists x.P(x) \Rightarrow \forall x.P(x)}\,(\text{CR})$$

4.4.5 Translating Sequent Proofs into English

Gentzen[4] 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:

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

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:

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

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:

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

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.*."

[4] Gentzen, Gerhard

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:

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

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 $\phi[x := y]$ [5]. *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.*"

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*

[5] 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.

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:

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

We say: "Since we know $\exists x.\phi$, pick an arbitrary element of the domain of discourse, call it y, and assume $\phi[x := y]$. *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 5

Set Theory

Georg Cantor

Georg Cantor (1845-1918) was a German mathematician and logician who created set theory. See [15]. To many, set theory is is the universal language of mathematics. Using set theory Cantor was able to characterize much of the known mathematics at the time and proved many fundamental theorems about set theoretic structures.

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 [38]. A classic presentation can be found in Halmos' [26] *Naive Set Theory*.

[1] We say it is "a foundation" since alternative approaches to the foundations of mathematics exist *e.g.* category theory [32, 41] or type theory [**?**] can also serve as the foundations of mathematics. Set theory is the foundational theory accepted by most working mathematicians.

5.1 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. This is in contrast to other structures like lists or sequences, where both the order of the elements and the number of times they occur (multiplicity) are taken into account when determining if two are equal. For equality on sets, all that matters is membership. Two sets are considered equal if they have the same elements.

5.1.1 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:

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

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 intentionally. 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.

5.1.2 Membership is primitive

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].

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

5.2 Equality and Subsets

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 or when they are subsets.

5.2.1 Extensionality

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*.

[2]Indeed there are some serious philosophers who reject it as senseless [20].

Definition 5.1.

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

We write $A \neq B$ for $\neg(A = B)$.
Consider the following sets.

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

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 $2 \notin \{1, \{2\}\}$.

5.2.2 Subsets

Definition 5.2 (Subset) 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 5.1. For every set A, $A \subseteq A$

Proof:

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

□

Differences between the membership and subset relations.

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

[3]It may be interesting to note that some philosophers with *nominalist* tendencies[19, 20] deny the distinction commonly made between these sets, they say that the "don't understand" the distinction being made between these sets.

Equality via subsets

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

Theorem 5.2 (subset extensionality) 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 ($\Leftarrow$). — def. of $\Leftrightarrow$ and ($\wedge$R)

($\Rightarrow$) 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.

($\Rightarrow$R) and ($\wedge$R)

($\forall$R)

($\forall$L) and ($\wedge$L)

($\Leftarrow$) 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.

($\wedge$L)

($\forall$L)

def. of $\Leftrightarrow$ and ($\wedge$R)

($\forall$L) twice

□

This theorem gives us another way to prove sets are equal; instead of proving that every element in both, show that the two sets are subsets of one another. In the same way we labeled the cases of the proof of an ($\Leftrightarrow$) by ($\Rightarrow$) and ($\Leftarrow$) we sometimes label the cases of a proof that two set are equal by ($\subseteq$) and ($\supseteq$).

5.3 Set Constructors

Although we have been discussing sets as if they exist, we need to provide various means to construct them.

5.3.1 The Empty Set

There exists a set containing no elements. We can write this fact as follows:

Axiom 5.1 (Empty Set)

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

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

Definition 5.3 (Uniqueness) 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 5.3. 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 restate the empty set axiom in a simpler form as follows.

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

Corollary 5.1 (Empty Set)

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

With this fact, we can easily prove the following theorem:

Theorem 5.4 (Empty Subset) For every set A, $\emptyset \subseteq A$.

Exercise 5.1. Prove Theorem 5.4.

5.3.2 Unordered Pairs and Singletons

Definition 5.4 (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 5.2 (Pairing)

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

Note that although we might believe such a set exists, (*i.e.* the set A containing just the elements x and y) without recourse to the pairing axiom, we would have no justification for asserting such a set exists.

Note that the the set constructed by the pairing axiom for particular x and y is unique *i.e.* if we fix x and y, and we claim that A and B are sets having the property claimed for sets whose existence is asserted by the pairing axiom, then $A = B$. This is made precise by the following lemma.

Lemma 5.1 (Pairing Unique)

$$\forall x.\forall y.\ unique(P)$$

where P is the property of sets defined as follows:

$$P(C) \stackrel{\text{def}}{=} \forall z.\ z \in C \Leftrightarrow (z = x \vee z = y)$$

Proof: Choose arbitrary x, y, and show that P is unique. Specifically, show that

$$\forall A.\forall B.P(A) \wedge P(B) \Rightarrow A = B$$

Choose arbitrary sets A and B and assume $P(A)$ and $P(B)$.

$$\begin{array}{ll} P(A): & \forall z.\ z \in A \Leftrightarrow (z = x \vee z = y) \\ P(B): & \forall z.\ z \in B \Leftrightarrow (z = x \vee z = y) \end{array}$$

To show $A = B$ we show that $w \in A \Leftrightarrow w \in B$ for arbitrary w.
case 1: Assume $w \in A$ and show $w \in B$. But by $P(A)$ (using w for z) if $w \in A$ then we know $(w = x \vee w = y)$. Now, using w for z in $P(B)$ this gives us the fact that $w \in B$ which is what we were to show.
case 2: Assume $w \in B$ and show $w \in A$. Use $P(B)$ (using w for z) similarly to we did in case 1 and this case holds as well.

□

Now, since any unordered pair composed of elements x and y is unique, we will write $\{x, y\}$ (or $\{y, x\}$) to denote this set. As a corollary, we restate the pairing axiom as follows:

Corollary 5.2 (Pairing)

$$\forall x.\forall y.\forall z.\ z \in \{x, y\} \Leftrightarrow (z = x \vee z = y)$$

From now on we will use this form of the pairing axiom instead of the form having the existential quantifier in its statement.

Singletons

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

Lemma 5.2 (Singleton Exists)

$$\forall x.\exists A.\forall z.\ z \in A \Leftrightarrow z = x$$

Proof: To prove the theorem, choose an arbitrary element (call it w) and show

$$(*)\quad \exists A.\forall z.z \in A \Leftrightarrow z = w$$

Now, by the pairing axiom, we know

$$\forall x.\forall y.\forall z.\ z \in \{x, y\} \Leftrightarrow (z = x \vee z = y)$$

Let both x and y be w, then we know, $\forall z.z \in \{w, w\} \Leftrightarrow (z = w \vee z = w)$. Since $(P \vee P) \Leftrightarrow P$ we can simplify this as $\forall z.\ z \in \{w, w\} \Leftrightarrow (z = w)$. Use $\{w, w\}$ as the witness for A in (*) giving $\forall z.\ z \in \{w, w\} \Leftrightarrow (z = w)$ which we have just shown to be true.
□

Like pairs, singletons are unique.

Corollary 5.3 (Singleton Unique)

$$\forall x.\ \mathit{unique}(P)$$

where P is the property of sets defined as follows:

$$P(C) \stackrel{\text{def}}{=} \forall z.\ z \in C \Leftrightarrow (z = x)$$

Proof: Singletons are just pairs where the elements are not distinct. Note that the proof uniqueness for pairs (Lemma 5.1) does not depend in any way on distinctness of the elements in the pair and so singletons are also unique.

□

Note that by extensionality, $\{x, x\} = \{x\}$, and since singletons are unique, we will write $\{x\}$ for the singleton containing x. Note that the singleton set $\{x\}$ is distinguished from its element x, *i.e.* $x \neq \{x\}$. Because the set that is claimed to exist in Lemma 5.2 is unique, we can restate that lemma more simply as follows.

Corollary 5.4 (Simplified Singleton Exists)

$$\forall x.\forall z.\ z \in \{x\} \Leftrightarrow z = x$$

Corollary 5.5 (Singleton Member)

$$\forall w.w \in \{w\}$$

Proof: To prove this, choose an arbitrary w and show $w \in \{w\}$. By Corollary 5.4, we know $\forall x.\forall z.z \in \{x\} \Leftrightarrow z = x$. In this formula, choose both x and z to be w, yielding the fact $w \in \{w\} \Leftrightarrow w = w$. Since $w = w$ is always true, we have shown $w \in \{w\}$.
□

Lemma 5.3 (Singleton Equality)

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

Exercise 5.2. Prove the singleton equality lemma.

5.3.3 Ordered Pairs

Kazimierz Kuratowski

Kazimierz Kuratowski (1896 – 1980) was a Polish mathematician who was active in the early development of topology and axiomatic set theory.

The pair $\{a, b\}$ and the pair $\{b, a\}$ are identical as far as we can tell using set equality. They are indistinguishable if we only consider their members. What if we want to be able to distinguish pairs by the order in which their elements are listed, is it possible using only sets? The following encoding of ordered pairs was first given by Kuratowski.

Definition 5.5 (ordered pair)

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

Note that the angled brackets ("$\langle$" and "$\rangle$") are used here to denote ordered pairs.

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 5.5 (characteristic property of ordered pairs) For sets A and B and for every $a, a' \in A$ and $b, b' \in B$,

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

Exercise 5.3. Prove theorem 5.5.

Definition 5.6 (projections) We define the *projection* functions[5] which map pairs (say p) to their first and second components.

$$\begin{array}{lcl} \pi_1\, p = x & \stackrel{\text{def}}{=} & \{x\} \in p \\ \pi_2\, p = b & \stackrel{\text{def}}{=} & \{\pi_1\, p, b\} \in p \end{array}$$

Thus, to prove that for a pair p, $\pi_1\, p = x$, it is enough to show that $\{x\} \in p$. Similarly, to show that $\pi_2\, p = b$ show that $\{\pi_1\, p, b\} \in p$.

Lemma 5.4. For every a and b the identities following hold.

i.) $\pi_1 \langle a, b\rangle = a$

ii.) $\pi_2 \langle a, b\rangle = b$

Proof: Choose arbitrary a and b.
i.) By definition $\pi_1 \langle a, b\rangle = a$ if and only if $\{a\} \in \langle a, b\rangle$. By definition, $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$, and $\{a\}$ is in this set so this case holds.
ii.) By definition $\pi_2 \langle a, b\rangle = a$ if and only if $\{\pi_1 \langle a, b\rangle, b\} \in \langle a, b\rangle$. We just saw (in the proof of i.) that $\pi_1 \langle a, b\rangle = a$, so we must check whether $\{a, b\} \in \{\{a\}, \{a, b\}\}$, which is true and so this case holds as well.

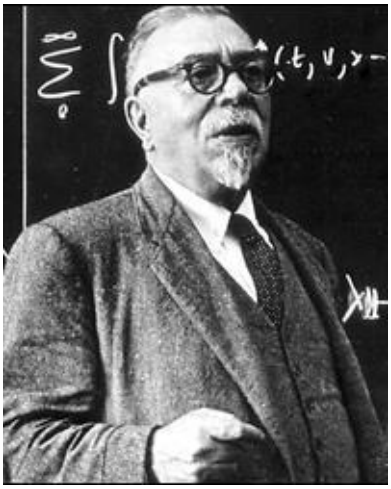
[5] After we introduce relations and functions below, it will be possible to prove that the projections, which technically are defined here as relations between a pair and its first or second element, really are functions.

□

Lemma 5.5.

$$\forall a.\pi_1 \langle a, a\rangle = \pi_2\langle a, a\rangle$$

Proof: Choose arbitrary a. By definition, $\langle a, a\rangle = \{\{a\}, \{a, a\}\}$, thus $\pi_1\langle a, a\rangle = a$ iff $\{a\} \in \{\{a\}, \{a, a\}\}$ which is true. Similarly, $\pi_2\langle a, a\rangle = a$ iff $\{a, a\} \in \{\{a\}, \{a, a\}\}$ which is also true and so the theorem holds.
□



Norbert Weiner

Norbert Wiener (1894 – 1964) was a U.S. mathematician who taught at MIT and founded the field of cybernetics.

Exercise 5.4. An alternative definition of ordered pairs (this was the first definition) was given by Norbert Wiener in 1914.

$$\langle x, y\rangle \stackrel{\text{def}}{=} \{\{\{x\}, \emptyset\}, \{\{y\}\}\}$$

Prove that this definition satisfies the characteristic property of ordered pairs as stated in Thm.5.5..

5.3.4 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.

Definition 5.7 (union) 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.

Axiom 5.3 (union membership axiom) 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 5.6. For every set A, $A \cup \emptyset = A$.

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

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

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

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

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

□

The following theorem asserts that the order of arguments to a union operation do not matter.

Theorem 5.7 (union commutes) 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 5.6 yields the following corollary.

Corollary 5.6. For every set A, $\emptyset \cup A = A$.

Theorem 5.8. 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 **??**, we have the following:

Corollary 5.7. For all sets A and B, $A \subseteq (B \cup A)$.

5.3.5 Set Intersection

Another way of constructing new sets form existing ones is to take their intersection.

Definition 5.8 (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:

Axiom 5.4 (intersection membership axiom)

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

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

Theorem 5.9. 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 empty set 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 empty set axiom, this is false so the right to left implication holds vacuously.
□

Theorem 5.10 (intersection commutes) For every pair of sets A and B, $A \cap B = b \cap A$.

Exercise 5.5. Prove Theorem 5.10.

5.3.6 Power Set

Definition 5.9 (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 $\rho(\mathcal{S})$.

Axiom 5.5 (power set) The axiom characterizing membership in the power-set says:

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

Consider the following examples:

$$\begin{array}{ll} A_0 = \{\} & \rho A_0 = \{\{\}\} \\ A_1 = \{1\} & \rho A_1 = \{\{\}, \{1\}\} \\ A_2 = \{1, 2\} & \rho A_2 = \{\{\}, \{1\}, \{2\}, \{1, 2\}\} \\ A_3 = \{1, 2, 3\} & \rho A_1 = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\} \end{array}$$

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

5.3.7 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 5.6 (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 powerful mechanism for defining new sets.

Note that we characterize membership in sets 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 5.1. 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 5.2. 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:

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

Substitution into Comprehensions*

Just like the quantifiers $\forall$ and $\exists$, the notation for a set defined by comprehension binds a variable. This makes substitutions into sets defined by comprehension interesting in the same way substitutions into quantified formulas can be. In the comprehension $\{x : \mathcal{S} \mid \phi\}$, free occurrences of x in the formula ϕ are bound by the declaration of x on the left side of "|." The following definition extends Def. 4.13 from chapter 4 to include sets defined by comprehension.

Definition 5.10 (capture avoiding substitution over a comprehension)

$$\begin{aligned}
\{x : \mathcal{S} \mid \phi\}[x := t] &= \{x : \mathcal{S} \mid \phi\} \\
\{y : \mathcal{S}[x := t] \mid \phi\}[x := t] &= \{y : \mathcal{S} \mid \phi[x := t]\} \\
&\quad \text{if } (x \neq y, y \notin FV(t)) \\
\{y : \mathcal{S} \mid \phi\}[x := t] &= \{z : \mathcal{S}[y := z][x := t] \mid \phi[y := z][x := t]\} \\
&\quad \text{if } (x \neq y, y \in FV(t), z \notin (FV(\mathcal{S}) \cup FV(t) \cup FV(\phi) \cup \{x\}))
\end{aligned}$$

Bertrand Russell

Bertrand Russell (1888 – 1972) was an English born philosopher and logician.

Remark 5.1 (Russell's Paradox) In our definition of comprehension we insisted that elements of sets defined by comprehension come from some preexisting set. In Cantor's original theory, this constraint was not stipulated and in (1900??) Bertrand Russell noticed the following problem.

Consider the set S consisting of those sets that do not contain themselves as elements. Without the constraint on comprehensions, this set is easily defined as follows.

$$S = \{x | x \notin x\}$$

Now, by the law of excluded middle of propositional logic, we know either $S \in S$ or $S \notin S$.
(Case 1) $S \in S$ iff $S \in \{x | x \notin x\}$. By the rule for membership in comprehensions, this is true iff $S \notin S$. But then we have shown that $S \in S \Leftrightarrow \neg(S \in S)$ which is a contradiction.
(Case 2) $S \notin S$ iff $S \notin \{x | x \notin x\}$ *i.e.* $S \notin \{x | x \notin x\}$ which is true iff $\neg(S \notin S)$ But the definition of $x \notin y$ simply is $\neg(x \in y)$ so we have $\neg\neg(S \in S)$. By double negation elimination, this is true iff $S \in S$. Thus, we have show that $S \notin S \Leftrightarrow S \in S$ which again is a contradiction.

5.3.8 Set Difference

Definition 5.11 (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 5.3. If *Even* is the set of even natural numbers, $\mathbb{N} - Even = Odd$.

Theorem 5.11. For every set A, $A - \emptyset = A$.

Theorem 5.12. For every set A, $A - A = \emptyset$.

Theorem 5.13. For all sets A and B, $A - B = \emptyset \Leftrightarrow A \subseteq B$.

Definition 5.12 (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 5.14. For all sets A and B, A and B are disjoint sets iff $A - B = A$.

5.3.9 Cartesian Products and Tuples

Definition 5.13 (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 5.5. $z = \langle a, b\rangle$ means z is a set of the form $\{\{a\}, \{a, b\}\}$. Evidently, the Cartesian product of two sets is a set of pairs.

Example 5.4. 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 5.15. 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 empty set axiom there is no such b so this case holds vacuously.
($\Leftarrow$): We assume $x \in \emptyset$ which, by the empty set axiom, is a contradiction so this case holds vacuously as well.
□

So theorem 5.15 says that $\emptyset$ is both a left and right identity for Cartesian product.

Lemma 5.6 (pair lemma) If A and B are arbitrary sets,

$$\forall x\colon A \times B.\, \exists y\colon A.\, \exists z\colon B.\, x = \langle y, z\rangle$$

Remark 5.2 (Membership in Comprehensions defined over 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 :

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

So,

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

By lemma 5.6, 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 $P[x, y]$.

5.4 Properties of Operations on Sets

A *set operator* is a mapping from sets to a set. For example, the union operator maps two sets to the set whose elements are those coming from either set. The number of set arguments (inputs) an operator takes is called the *arity* of the operator. A *unary operator* maps a single set to a new set. A *binary operator* maps two sets to set. In general, if an operator take k arguments, we say is it a k-ary operation..

5.4.1 Idempotency

Definition 5.14 (Idempotence) Given a binary operation $\star$, its *idempotent* elements are those elements x for which $x \star x = x$.

Example 5.5. For the operation of ordinary multiplication, 0 and 1 are (the only) idempotent elements. For the operation of addition, 0 (but not 1) is an idempotent element.

The following lemma shows that every set is an idempotent element for intersections and unions.

Lemma 5.7 (Intersection Idempotent) $\forall A.\ A \cap A = A$.

Lemma 5.8 (*cup* Idempotent) $\forall A.\ A \cup A = A$.

5.4.2 Monotonicity

Monotonicity is a property of a unary operators.

Definition 5.15 (Monotone) A unary set operator (say X) is *monotone* if for all sets A and B, $A \subseteq B \Rightarrow X(A) \subseteq X(B)$.

Example 5.6. For an arbitrary sets A and B, the powerset operation is monotone *i.e.*

$$A \subseteq B \rightarrow \rho(A) \subseteq \rho(B)$$

5.4.3 Commutativity

Definition 5.16 (Commutative) A binary set operator (say $\circ$) is *commutative* if for all sets A, B

$$(A \circ B) = (B \circ A)$$

Lemma 5.9 (Intersection commutative) Set intersection is commutative:

$$A \cap B = B \cap A$$

Lemma 5.10 (Union commutative) Set union is commutative.

$$A \cup B = B \cup A$$

5.4.4 Associativity

Definition 5.17 (Associative) A binary set operator (say $\circ$) is *associative* if for all sets A, B, and C,

$$A \circ (B \circ C) = (A \circ B) \circ C)$$

Lemma 5.11 (Intersection associative) Set intersection is associative.

$$A \cap (B \cap C) = (A \cap B) \cap C$$

Lemma 5.12 (Union associative) Set union is associative.

$$A \cup (B \cup C) = (A \cup B) \cup C$$

5.4.5 Distributivity

The distributive property relates pairs of operators.

Definition 5.18 (Distributive) For binary set operators (say $\circ$ and $\square$), we say$\circ$ *distributes over* $\square$ if all sets A, B, and C,

$$A \circ (B \square C) = (A \square B) \circ (a \square C)$$

Lemma 5.13 (Union distributes over intersection)

$$\forall A, B, C.\ A \cup (B \cap C) = (A \cap B) \cup (A \cap C)$$

Lemma 5.14 (Intersection distributes over union)

$$\forall A, B, C.\ A \cap (B \cup C) = (A \cup B) \cap (A \cup C)$$

Chapter 6

Relations

Alfred Tarski

Alfred Tarski (1902–1983) was born in Poland and came to the US at the outbreak of WWII. Tarski was the youngest person to ever earn a Ph.D. for the University of Warsaw and throughout his career he made many many contributions to logic and mathematics – though he may be best know for his work in semantics and model theory. Tarski and his students developed the theory of relations as we know it. See [14] for a complete and and rather personal biography of Tarski's life and work.

6.1 Introduction

Relations establish a correspondence between the elements of sets thereby imposing structure on the elements. 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 together $\langle x, y\rangle \in R$ and $\langle x, z\rangle \in R$ imply $y = z$. 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 $yF^{-1}x \stackrel{\text{def}}{=} xFy$. We discuss functions further in Chapter 8.

Relations and functions play a crucial role 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.

6.2 Relations

6.2.1 Binary Relations

Definition 6.1 (Binary Relation)

> 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.

Definition 6.2 (domain and co-doman)

> If $R \subseteq A \times B$ then we say A is the *domain* of R and B is the *codomain* of R.

Example 6.1. 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 6.2. 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 6.3. Less-than ($<$) is a relation on $(\mathbb{Z} \times \mathbb{Z})$.

$$< \; = \{\langle x, y\rangle \in \mathbb{Z} \times \mathbb{Z} \mid x \neq y \wedge \exists k{:}\mathbb{N}.\; y = x + k\}$$

To aid readability, binary 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 6.3. If $R \subseteq A \times A$ we say *R is a relation on A.*

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

6.2.2 n-ary Relations

6.2.3 Some Particular Relations

Definition 6.4 (Diagonal Relation (Identity Relation)) The *diagonal relation* over a set A is the relation

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

Exercise 6.1. Prove that $\Delta_A^{-1} = \Delta_A$.

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 6.4. Suppose $A = \{0, 1, 2\}$ and $R = \{\langle 0, 0\rangle, \langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle\}$ 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]. One problem with the matrix representation is that you must order the elements of the underlying set to be able to write them down across the top and bottom of the matrix – this choice may have to be arbitrary if there is no natural order for the elements.

6.3 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 sets.

Definition 6.5 (subrelation) If $R, S \subseteq A \times B$ and $R \subseteq S$ then we say R is a *subrelation* of S.

We define a number of operations that are specifically defined on relations – mainly owing to the fact that they are are sets but also have additional structure in that they are sets of pairs.

6.3.1 Inverse

Definition 6.6 (inverse) If $R \subseteq A \times B$, then the *inverse* of R is

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

Thus, to construct the inverse of a relation, just turn every pair around *i.e.* swap the elements in the pairs of R by making the first element of each pair the second and the second element of each pair the first. The following useful lemma says that swapping the order of the elements from a pair in a relation R puts the new pair into the relation R^{-1}.

Remark 6.1. In linear algebra, the inverse, as defined here, is called the *transpose.*

Lemma 6.1. If $R \subseteq A \times B$, then $aRb \Leftrightarrow bR^{-1}a$.
Proof: ($\Rightarrow$) Assume aRb, *i.e.* $\langle a, b\rangle \in R$. We must show that $\langle b, a\rangle \in R^{-1}$. By the definition of inverse, we must show that $\langle b, a\rangle \in \{\langle y, x\rangle \in B \times A | \langle x, y\rangle \in R\}$. Now, since $\langle a, b\rangle \in R$ we know (since $R \subseteq A \times B$) that $\langle b, a\rangle \in B \times A$ and we assumed $\langle a, b\rangle \in R$ so this case is proved.
($\Leftarrow$) Assume $bR^{-1}a$ and show aRb. If $bR^{-1}a$, then $\langle b, a\rangle \in \{\langle y, x\rangle \in B \times A | \langle x, y\rangle \in R\}$, *i.e.* $b \in B$, $a \in A$ and $\langle a, b\rangle \in R$ which is what we were to show.

[1] Students familiar with linear algebra will know that this is the identity matrix and might consider the relationship between composition of relations and matrix multiplication – in particular – if you consider ordinary matrix multiplication where addition become $\vee$ and multiplication is , we can show the isomorphism between composition of relations and this Boolean matrix multiplication.

□

Example 6.5. The inverse of the less-than relation ($<$) is greater-than ($>$).

6.3.2 Complement of a Relation

Definition 6.7 (complement) If $R \subseteq A \times B$, then the *complement* of R is the relation

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

Corollary 6.1. For every relation $R \subseteq A \times B$ and for every $a \in A$ and $b \in B$,

$$a\overline{R}b \Leftrightarrow \neg(aRb)$$

Exercise 6.2. Prove that if $R \subseteq A \times B$, then $\overline{R} = (A \times B) - R$

Example 6.6. The complement of the less-than relation ($<$) is the greater-than-or-equal-to relation ($\geq$).

6.3.3 Composition of Relations

Given relations whose codomain and domains match-up in the proper way, we can construct a new relation which is the composition of the two.

Definition 6.8 (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:

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

Remark 6.2. 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 for adopting this convention might be more clear when we get to functions.

Example 6.7. 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.

Theorem 6.1 (Composition is associative) For arbitrary sets A, B, C and D if $R \subseteq A \times B$, $S \subseteq B \times C$ and $T \subseteq C \times D$ then

$$T \circ (S \circ R) = (T \circ S) \circ R$$

Proof: First, note by the definition of composition that

$$(T \circ (S \circ R)) \subseteq A \times D \text{ and } ((T \circ S) \circ R) \subseteq A \times D$$

For arbitrary $\langle x, y\rangle \in A \times D$ we show

$$\langle x, y\rangle \in (T \circ (S \circ R)) \Leftrightarrow \langle x, y\rangle \in ((T \circ S) \circ R)$$

Starting on the left, by the definition of composition we know the following.

$$\langle x, y\rangle \in (T \circ (S \circ R)) \Leftrightarrow \exists z : C.\, \langle x, z\rangle \in (S \circ R) \wedge \langle z, y\rangle \in T$$

So we assume there is such a z, *i.e* so far we know

$$\begin{array}{ll} i.) & \langle x, y\rangle \in (T \circ (S \circ R)) \\ ii.) & \langle x, z\rangle \in (S \circ R) \\ iii.) & \langle z, y\rangle \in T \end{array}$$

Then (by the definition of composition) and $(ii.)$ we obtain two more facts which hold for some arbitrary element $w \in B$.

$$\begin{array}{ll} iv.) & \langle x, w\rangle \in R \\ v.) & \langle w, z\rangle \in S \end{array}$$

From $(v.)$ and $(iii.)$ and the definition of composition we obtain the following.

$$vi.) \quad \langle w, y\rangle \in (T \circ S)$$

But then $(iv.)$ and $(vi.)$ together mean $\langle x, y\rangle \in ((T \circ S) \circ R)$ which completes the proof.
□

Theorem 6.2 (Composition inverse lemma) For all relations $R \subseteq A \times B$ and $S \subseteq B \times C$, the following identity holds.

$$(S \circ R)^{-1} = R^{-1} \circ S^{-1}$$

Proof: Let A , B and C be arbitrary sets and $R \subseteq A \times B$ and $S \subseteq B \times C$ be arbitrary relations. Note that $(S \circ R) \subseteq A \times C$ and so the inverse relation $(S \circ R)^{-1} \subseteq C \times A$. So, by extensionality, we must show for arbitrary $a \in A$ and $c \in C$ that $c(S \circ R)^{-1}a \Leftrightarrow c(R^{-1} \circ S^{-1})a$.
We reason equationally. Starting on the left side with $c(S \circ R)^{-1}a$ and show the right side $c(R^{-1} \circ S^{-1})a$ holds. By Lemma 6.1,

$$c(S \circ R)^{-1}a \Leftrightarrow a(S \circ R)c$$

Now, by definition of composition, $a(S \circ R)c$ if and only if there is some $b \in B$ such that both aRb and bSc hold. But then (by two applications of Lemma 6.1) we know $cS^{-1}b$ and $bR^{-1}a$ also hold. By the definition of composition this means
$c(R^{-1} \circ S^{-1})a$, which was to be shown.

□

Recall the definition of the diagonal relation Δ_A (Def. 6.4).

Lemma 6.2. 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 5.5.2, *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 6.3. Prove the following lemma.

Lemma 6.3. If R is any relation on $A \times B$, then $(\Delta_B \circ R) = R$.

Note that in rational arithmetic the reciprocal of $\frac{1}{x}$ of x is the multiplicative inverse: $x * \frac{1}{x} = 1$. So, the multiplication of a number with it's inverse gives the identity element for the operation of multiplication. We have just shown that the identity for composition of relations is Δ_A (where A depends on the domain and codomain of the relation.) Based on this we might make the following *false* conjecture.

Conjecture 6.1 (False) For relations $R \subseteq A \times B$:

$$R^{-1} \circ R = \Delta_A$$

Exercise 6.4. For $A = \{a, b, c\}$ and $R = \{\langle a, b\rangle, \langle a, c\rangle, \langle b, a\rangle, \langle c, b\rangle\}$ show that the conjecture is false.

Exercise 6.5. For $A = \{a, b\}$ and $R = \{\langle a, b\rangle\}$ show that the conjecture is false.

We will see in chapter 8 that the conjecture is true when we consider functions which are a restricted form of relations.

Definition 6.9 (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 6.2. For all relations R on a set A, $R^1 = R$, since, by the definition of iterated composition and by Lemma 6.2 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 6.8. 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; $R^k = \{\langle x, y\rangle \in \mathbb{N} \times \mathbb{N} \mid x = y + k\}$.

Exercise 6.6. Let R be the successor relation as defined in example 6.8. Prove that

$$(\leq \circ R) =<$$

For each $k \in \mathbb{N}$, R^k is the relation that takes you directly to the places reachable in R by following k steps. The following two definitions collect together $R^{k'}s$ where k ranges over $\mathbb{N}^+$ (the strictly positive natural numbers[2]) and $\mathbb{N}$.

Definition 6.10 (reachability in R)

$$R^+ = \bigcup_{i \in \mathbb{N}^+} R^i$$

Definition 6.11 (connectivity of R)

$$R^* = \bigcup_{i \in \mathbb{N}} R^i$$

So, R^+ contains all the pairs $\langle x, y\rangle \in A \times A$ such that y is reachable from x by following *one or more* edges of R. Similarly, R^* contains all the pairs $\langle x, y\rangle \in A \times A$ such that y is reachable from x by following *zero or more* edges of R. As we will see below; R^+ is the so-called transitive closure of R (Thm 6.6) and R^* is the reflexive transitive closure of R (Thm 6.7).

6.4 Properties of Relations

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

A relation $R \subset A \times A$ is

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.)	asymmetric	$\forall a, b : A.\ (aRb \Rightarrow b \not R a$
6.)	transitive	$\forall a, b, c : A.\ (aRb \wedge bRc) \Rightarrow (aRc)$
7.)	connected	$\forall a, b : A.\ a \neq b \Rightarrow (aRb \vee bRa)$

We discuss each of these properties individually below.

[2] $\mathbb{N}^+ \stackrel{\text{def}}{=} \mathbb{N} - \{0\}$.

6.4.1 Reflexivity

Definition 6.12.

$$Relf_A(R) \stackrel{\text{def}}{=} \forall a : A.\ aRa$$

If we think of xRy as meaning we can get from x to y by following one edge in R, then saying R is a reflexive relation means that there is an edge (or really a loop) in R from every point to itself.

Lemma 6.4. For all relations $R \subseteq A \times A$, if R is reflexive, and Δ_A is the diagonal relation on A, then $\Delta_A \subseteq R$.

Lemma 6.5. For all relations $R, S \subseteq A \times A$, if R and S are reflexive, then $R \cap S$ is reflexive.

Examples of reflexive relations include equality ($=$), less-than-or-equal ($\leq$) and greater-than-or-equal ($\geq$), the proper subset relation ($\subset$) and, for propositions, iff ($\Leftrightarrow$).

6.4.2 Irreflexivity

Definition 6.13.

$$Irref_A(R) \stackrel{\text{def}}{=} \forall a : A.\ a \not R a$$

Irreflexivity means than no element of the set is connected directly to itself.

Remark 6.3. 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 $\langle 1, 1\rangle \in R$.

Lemma 6.6. For all relations $R, S \subseteq A \times A$, if R and S are irreflexive, then $R \cap S$ and $R \cup R$ are both irreflexive.

Examples of irreflexive relations include inequality ($\neq$), less-than ($<$), greater-than ($>$) and the proper subset relation ($\subset$).

6.4.3 Symmetry

Definition 6.14.

$$Sym(R) \stackrel{\text{def}}{=} \forall a, b : A.\ aRb \Rightarrow bRa$$

A relation R is symmetric if every point reachable in one step in R can be returned to by taking a single step also in R.

Lemma 6.7. The diagonal relation Δ_A is symmetric.

Lemma 6.8. If $R \subseteq A \times A$ then $R = R^{-1} \Leftrightarrow Sym(R)$.

Lemma 6.9. If $R \subseteq A \times A$ then $R = R^{-1} \Leftrightarrow R \subseteq R^{-1}$.

6.4.4 Antisymmetry

Definition 6.15.

$$AntiSym(R) \stackrel{\text{def}}{=} \forall a, b : A.\ (aRb \wedge bRa) \Rightarrow a = b$$

Antisymmetry means that if you can get from one point to another and back in one step, then those points must have been equal.

Lemma 6.10. The diagonal relation Δ_A is antisymmetric.

6.4.5 Asymmetry

Definition 6.16.

$$Asym(R) \stackrel{\text{def}}{=} \forall a, b : A.\ (aRb \Rightarrow b \not R a$$

Asymmetry means that there is no way to get from any point to itself in two steps.

Lemma 6.11. For all relations $R \subseteq A \times A$, if R is asymmetric then R is irreflexive.

6.4.6 Transitivity

$$Trans(R) \stackrel{\text{def}}{=} \forall a, b, c : A.\ (aRb \wedge bRc) \Rightarrow (aRc)$$

A relation is transitive if every place you can get in two steps, you can get by taking a single step.

6.4.7 Connectedness

$$Connected(R) \stackrel{\text{def}}{=} \forall a, b : A.\ a \neq b \Rightarrow (aRb \vee bRa)$$

A relation is connected if there is an edge between every pair of points (going one direction or the other.)

6.5 Closures

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) and to get the "smallest" such extension.

Example 6.9. Consider A and R as just presented in remark 6.3. 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 6.17 (Closure) 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 set of relations S such that $P(S)$ and $R \subseteq S$ and S is the smallest such relation,

$$S \in \mathit{closure}(R, P)$$
$$\text{iff } (P(S) \wedge R \subseteq S) \wedge \forall T : T \subseteq A \times B \Rightarrow ((P(T) \wedge R \subseteq T) \Rightarrow S \subseteq T)$$

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

We now show that membership in a closure is unique.

Theorem 6.3 (Uniqueness of Closures) If $R \subseteq A \times A$ and P is a property of relations then, the property of being a member in $\mathit{closure}(R, P)$ is unique.

Proof: Let R and P be arbitrary. The property (call it M) that we are showing is unique is membership in a closure *i.e.*

$$M(S) = S \in \mathit{closure}(R, P)$$

We recall the definition of uniqueness (Def. 5.5.3) which says

$$\mathit{unique}(M) \stackrel{\text{def}}{=} \forall R_1, R_2.(M(R_1) \wedge M(R_2)) \Rightarrow (R_1 = R_2)$$

To show this, we assume $M(R_1)$ and $M(R_2)$ for arbitrary relations $R_1, R_2 \subseteq A \times A$ and show $R_1 = R_2$. By our assumption, we know:

$$\begin{array}{ll} M(R_1): & R_1 \in \mathit{closure}(R, P) \\ M(R_2): & R_2 \in \mathit{closure}(R, P) \end{array}$$

Since R_1 and R_2 are in the closure of R by P, we know

$$\begin{array}{rl} i.) & R \subseteq R_1 \\ ii.) & P(R_1) \\ iii.) & \forall T \subseteq A \times A.\, (R \subseteq T \wedge P(T)) \Rightarrow R_1 \subseteq T \\ iv.) & R \subseteq R_2 \\ v.) & P(R_2) \\ vi.) & \forall T \subseteq A \times A.\, (R \subseteq T \wedge P(T)) \Rightarrow R_2 \subseteq T \end{array}$$

Using R_2 for T in (*iii.*) yields the following.

$$(R \subseteq R_2 \wedge P(R_2)) \Rightarrow R_1 \subseteq R_2$$

By (*iv.*) and (*v.*) We get that $R_1 \subseteq R_2$. Similarly, using R_1 for T in (*vi.*) yields the following.

$$(R \subseteq R_1 \wedge P(R_1)) \Rightarrow R_2 \subseteq R_1$$

By (*i.*) and (*ii.*) We get that $R_2 \subseteq R_1$. But then by subset extensionality (Thm. 5.5.2) $R_1 = R_2$.
□

Since closures are unique, from now on we will simply write $S = closure(R, P)$ instead of $S \in closure(R, P)$. Also, since closures are unique, any relation which has the property of being a closure must be the only one that is the closure *i.e.* to prove $S = closure(R, P)$ we simply need to show that S has the three properties that make it the closure of R by P.

Theorem 6.4. 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 reflexive 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.) $\forall T : T \subseteq A \times A \Rightarrow ((R \subseteq T \subseteq Refl_A(T)) \Rightarrow (R \cup \Delta_A) \subseteq T)$

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 5.8 from Chapter 5.
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 6.18 (Symmetric) 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 6.10. For any set A, the empty relation is symmetric, though the empty relation is reflexive if and only if $A = \emptyset$.

Theorem 6.5 (Symmetric Closure) If $R \subseteq A \times A$ then the symmetric closure of R is the relation $R \cup R^{-1}$

Example 6.11. 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

$$R \cup R^2 \cup R^3 \cup R^4$$

Theorem 6.6 (Transitive Closure) If $R \subseteq A \times A$ then the transitive closure of R is the relation

$$R^+ = \bigcup_{i>0} R^i$$

Theorem 6.7. The *reflexive transitive closure* of a relation $R \subseteq A \times A$ is the relation

$$R^* = \bigcup_{i \in \mathbb{N}} R^i$$

Remark 6.4. The notation R^* to denote the reflexive transitive closure of a relation R is borrowed from the theory of strings and regular languages. It was first introduced by Stephen Cole Kleene (1909-1994) and, in the context of regular languages, is called the *Kleene Closure* of R.

6.6 Properties of Operations on Relations

Just as we had properties that may or may not hold for relations, we can consider properties of the operations on relations. This idea of properties of operations is a "higher order" concept.

Definition 6.19 (Involution) A unary operator $\_' : A \to A$ is an involution if it is its own inverse, *i.e.* if $x'' = x$ for all $x \in A$.

Lemma 6.13 (complement involutive) For every pair of sets A and B and every relation R, $R \subseteq A \times B$ the following identity holds.

$$\overline{\overline{R}} = R$$

.

Proof: By extensionality. Choose arbitrary $a \in A$ and $b \in B$ and show $a\overline{\overline{R}}b \Leftrightarrow aRb$. We reason equationally.

$$a\overline{\overline{R}}b \Leftrightarrow \neg(a\overline{R}b) \Leftrightarrow \neg\neg(aRb) \Leftrightarrow aRb$$

□

Theorem 6.8 (Inverse involutive) For every pair of sets A and B, and for every $R \subseteq A \times B$

$$R = (R^{-1})^{-1}$$

Proof: Note that since $R \subseteq A \times B$ is a set, we must show (by extensionality) that for arbitrary z, that $z \in R \Leftrightarrow z \in (R^{-1})^{-1}$. Since $R \subseteq A \times B$, z is of the form $\langle a, b\rangle$ for some $a \in A$ and some $b \in B$, thus, we must show $aRb \Leftrightarrow a(R^{-1})^{-1}b$. Two applications of Lemma 6.1 give the following.

$$aRb \Leftrightarrow bR^{-1}a \Leftrightarrow a(R^{-1})^{-1}b$$

□

Chapter 7

Equivalence and Order

7.1 Equivalence Relations

Equivalence relations generalize of the notion of what it means for two elements of a set to be equal.

Definition 7.1 (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 7.1. Ordinary equality on numbers is an equivalence relation.

Example 7.2. 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 7.3. 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, then 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 7.1. For any set A,, the diagonal relation Δ_A is an equivalence relation on A.

This is the so-called "finest" equivalence (see Definition 7.3 below) 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 7.2. 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.

7.1.1 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 7.2 (equivalence class) If A is a set, R is an equivalence relation on A 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 7.4. 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 7.1. 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.

1. xRy
2. $[x]_R \cap [y]_R \neq \emptyset$
3. $[x]_R = [y]_R$

Proof: To prove 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, there 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.

□

Definition 7.3 (Fineness of an Equivalence) An equivalence relation $\equiv_1 \subseteq A \times A$ is *finer* than the equivalence relation $\equiv_2 \subseteq A \times A$ if,

$$\forall x : A.\ [x]_{\equiv_1} \subseteq [x]_{\equiv_2}$$

7.1.2 The Quotient Construction*

In higher mathematics a *quotient* is a structure induced by an equivalence relation.

Definition 7.4 (Quotient) If $\equiv$ is an equivalence relation on the set A, then we define

$$A/\equiv \ \stackrel{\text{def}}{=}\ \{[x]_{\equiv} | x \in A\}$$

This set of sets is called the the *quotient of A modulo $\equiv$*.

Lemma 7.3. For every set A, $A = A/\Delta_A$.

Lemma 7.4. For every set A, $A/A^2 = \{A\}$.

7.1.3 $\mathbb{Q}$ is a Quotient

Consider the fractions $\mathcal{F}$ defined as follows.

Definition 7.5.

$$\mathcal{F} = \mathbb{Z} \times \mathbb{Z}^{\{\neq 0\}}$$

where $\mathbb{Z}^{\{\neq 0\}}$ is the set of non-zero integers.

You may recognize $\mathcal{F}$ as the fractions, *e.g.* we can think of the first number as the numerator and the second as the denominator, so $\langle a, b\rangle$ is the fraction $\frac{a}{b}$.

Now, note that the equality on fractions (*i.e.* the equality on pairs – $\langle a, b\rangle = \langle c, d\rangle \Leftrightarrow a = c \wedge b = d$) is not the equality for rational numbers (usually denoted $\mathbb{Q}$.) Notice that, for example,

$$\langle 1, 2\rangle \neq \langle 2, 4\rangle$$

but of course, for rational numbers

$$\frac{1}{2} = \frac{2}{4}$$

We define an equivalence relation on pairs of fractions that does reflect equality on rationals as follows:

Definition 7.6.

$$\equiv_{\mathbb{Q}} \stackrel{\text{def}}{=} \{\langle\langle x, y\rangle, \langle z, w\rangle\rangle \in \mathcal{F} \times \mathcal{F} \mid \ xw = yz\}$$

Less pedantically we might write

$$\langle x, y\rangle \equiv_{\mathbb{Q}} \langle z, w\rangle \stackrel{\text{def}}{=} xw = yz$$

This is the ordinary cross-multiplication rule you learned in grade school for determining if two rational numbers are equal.

Exercise 7.1. Prove that $\equiv_{\mathbb{Q}}$ is indeed an equivalence relation on fractions *i.e.* you must show that it is (i.) reflexive, (ii.) symmetric and (iii.) transitive.

1. $\forall \langle a, b\rangle \in \mathcal{F}.\ \langle a, b\rangle \equiv_{\mathbb{Q}} \langle a, b\rangle$
2. $\forall \langle a, b\rangle, \langle c, d\rangle \in \mathcal{F}.\ \langle a, b\rangle \equiv_{\mathbb{Q}} \langle c, d\rangle \Rightarrow \langle c, d\rangle \equiv_{\mathbb{Q}} \langle a, d\rangle$
3. $\forall \langle a, b\rangle, \langle c, d\rangle, \langle e, f\rangle \in \mathcal{F}.$
   $(\langle a, b\rangle \equiv_{\mathbb{Q}} \langle c, d\rangle \ \wedge \ \langle c, d\rangle \equiv_{\mathbb{Q}} \langle e, f\rangle) \Rightarrow \langle a, b\rangle \equiv_{\mathbb{Q}} \langle e, f\rangle$

Exercise 7.2. Describe the equivalence class $[\langle 2, 4\rangle]_{\equiv_{\mathbb{Q}}}$

Exercise 7.3. Describe the equivalence class $[\langle x, y\rangle]_{\equiv_{\mathbb{Q}}}$

The rational numbers are defined by a quotient construction.

Definition 7.7 (Rational Numbers)

$$\mathbb{Q} \stackrel{\text{def}}{=} \mathcal{F}/\equiv_{\mathbb{Q}}$$

This conception of the rational numbers is perhaps confusing. It leads to the following dialog.

Question: "What is a rational number?"
Answer: "A rational number is an equivalence class of fractions."
Question: "But then what does it mean to add two rational numbers."
Answer: "Addition is an operation that maps a pair of rational numbers (equivalence classes of fractions) to a rational number (another equivalence class of fractions.)."

7.1.4 Partitions

Definition 7.8 (Partition) A *partition* of a set A is a set of non-empty 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* (or a *component*) and a collection of such A_i is a partition if it satisfies the following two properties:

i.) For all $i \in I$, $A_i \neq \emptyset$

ii.) 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,

iii.) the union of the sets $A_i, i \in I$ is the set A itself:

$$\bigcup_{i \in I} A_i = A$$

Example 7.5. 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 7.2. 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.*

$$Equiv_A(R) \Leftrightarrow Partition(\bigcup_{x \in A} \{[x]_R\})$$

Exercise 7.4. Prove Thm. 7.2.

Counting Partitions

Definition 7.9 (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.

Definition 7.10 (Counting k-partitions) The numbers computed by the following recurrence relation are called Stirling Numbers of the second kind. They compute the number of k-partitions of a set of size n.

$$S(n, 1) = 1$$
$$S(n, n) = 1$$
$$S(n, k) = S(n - 1, k - 1) + k \cdot S(n - 1, k)$$

Definition 7.11 (Counting Equivalence Relations) There are as many equivalence relations on a set of size n as there are k-partitions for each $k \in \{1 \cdots n\}$.

$$\sum_{k=1}^{n} S(n, k)$$

7.1.5 Congruence Relations*

It is all well and good to define equivalence relations on a set, but usually we consider sets together with operations on them, for many applications, we expect the equivalence to, in some sense, respect those operations. This is the idea of the *congruence relation* – a congruence is an equivalence that respects operators or is *compatible* with one or more operators. In ordinary situations with equality, we expect the substitution of "equals" for "equals" to not change anything. So, if $x = y$, then x can be replaced with y in any context *e.g.* if x and y are equal, we expect $f(x) = f(y)$.

The idea of congruence is to ensure that substitution of equivalent for equivalent in an operator results in equivalent elements.

Definition 7.12. If f is an k-ary operation on the set A and $\equiv$ is an equivalence relation on A then $\equiv$ is a *congruence* for f if

$$\forall x_1, \cdots, x_k, y_1, \cdots, y_k : A.(\forall i : \{1..k\}.x_i \equiv y_i) \Rightarrow f(x_1, \cdots, x_k) \equiv f(y_1, \cdots, y_k)$$

This may look pretty complicated, but is the general form for an arbitrary k-ary operation. Here's the restatement for a binary operator.

Definition 7.13. If f is an binary operation on the set A and $\equiv$ is an equivalence relation on A then $\equiv$ is a *congruence* for f if

$$\forall x_1, x_2, y_1, y_2 : A.(x_1 \equiv y_1 \wedge x_2 \equiv y_2) \Rightarrow f(x_1, x_2) \equiv f(y_1, y_2)$$

We will sometimes write $Cong(\equiv, f)$ to indicate that the equivalence relation $\equiv$ is compatible with the operator f.

Operations on Rational Numbers

Consider the following operations on fractions (Def. 7.5).

Definition 7.14 (Multiplication of fractions) We define multiplication of fractions pointwise as follows:

$$\langle a, b\rangle *_{\mathcal{F}} \langle c, d\rangle \stackrel{\text{def}}{=} \langle ac, bd\rangle$$

where ac and bd denote ordinary multiplication of integers.

Definition 7.15 (Addition of fractions) We define addition of fractions as follows.

$$\langle a, b\rangle +_{\mathcal{F}} \langle c, d\rangle \stackrel{\text{def}}{=} \langle ad + bc, bd\rangle$$

Exercise 7.5. Prove that the multiplication and addition of of fractions both result in fractions *i.e.*

$$\begin{array}{ll} i.) & \forall \langle a, b\rangle, \langle c, d\rangle : \mathcal{F}.\, (\langle a, b\rangle *_{\mathcal{F}} \langle c, d\rangle) \in \mathcal{F} \\ ii.) & \forall \langle a, b\rangle, \langle c, d\rangle : \mathcal{F}.\, (\langle a, b\rangle +_{\mathcal{F}} \langle c, d\rangle) \in \mathcal{F} \end{array}$$

Lemma 7.5. The relation $\equiv_{\mathbb{Q}}$ is compatible with the operation $*_{\mathcal{F}}$ *i.e.* $\equiv_{\mathbb{Q}}$ is congruent with respect to $*_{\mathcal{F}}$.

Proof: We must show that

$$\begin{array}{l}\forall\langle a,b\rangle,\langle c,d\rangle,\langle e,f\rangle,\langle g,h\rangle : \mathcal{F}.\\ \quad(\langle a,b\rangle \equiv_{\mathbb{Q}} \langle e,f\rangle \wedge \langle c,d\rangle \equiv_{\mathbb{Q}} \langle g,h\rangle)\\ \qquad\Rightarrow\ \ \langle a,b\rangle *_{\mathcal{F}} \langle e,f\rangle \equiv_{\mathbb{Q}} \langle c,d\rangle *_{\mathcal{F}} \langle g,h\rangle\end{array}$$

Assume that $\langle a,b\rangle,\langle c,d\rangle,\langle e,f\rangle,\langle g,h\rangle \in \mathcal{F}$ are arbitrary. Then, since these pairs are fractions, we know that $b \neq 0$, $d \neq 0$, $f \neq 0$ and $h \neq 0$. Also, assume $\langle a,b\rangle \equiv_{\mathbb{Q}} \langle c,d\rangle$ and $\langle e,f\rangle \equiv_{\mathbb{Q}} \langle g,h\rangle$. Then, by the definition of $\equiv_{\mathbb{Q}}$ we know $ad = bc$ and $eh = fg$. We must show

$$\langle a,b\rangle *_{\mathcal{F}} \langle e,f\rangle \equiv_{\mathbb{Q}} \langle c,d\rangle *_{\mathcal{F}} \langle g,h\rangle$$

By definition of $*_{\mathcal{F}}$ we know $\langle a,b\rangle *_{\mathcal{F}} \langle e,f\rangle$ is the pair $\langle ae,bf\rangle$ and $\langle c,d\rangle *_{\mathcal{F}} \langle g,h\rangle$ is the pair $\langle cg,dh\rangle$. We must show that these results are equivalent *i.e.* $(\langle ae,bf\rangle \equiv_{\mathbb{Q}} \langle cg,dh\rangle)$. To show this, we must show that $aedh = bfcg$. Now, since $ad = bc$ and since $d \neq 0$ we can divide both sides by d yielding the equality $a = \frac{bc}{d}$. Using this fact together with $eh = fg$ we show $aedh = bfcg$ as follows.

$$aedh = \frac{bc}{d}(edh) = \frac{bcedh}{d} = bceh = bcfg = bfcg$$

□

The significance of the lemma is that the operation $*_{\mathcal{F}}$ respects the equivalence $\equiv_{\mathbb{Q}}$ *i.e.* even though it is defined as an operation on fractions, substitution of $\equiv_{\mathbb{Q}}$-equivalent elements yield $\equiv_{\mathbb{Q}}$-equivalent results.

Lemma 7.6. The relation $\equiv_{\mathbb{Q}}$ is compatible with the operation $+_{\mathcal{F}}$ *i.e.* $\equiv_{\mathbb{Q}}$ is congruent with respect to $+_{\mathcal{F}}$.

Exercise 7.6. Prove Lemma 7.6

7.2 Order Relations

Equivalence relations abstract the notion of equality while order relations abstract the notion of order. We are all familiar with orderings on the integers, less-than ($<$) and less-than-or-equal ($\leq$).

7.2.1 Partial Orders

Definition 7.16 (Partial Order)

> If $R \subset A \times B$ is a relation that is reflexive, antisymmetric and transitive we call it a *partial order*.

Partial order relations are usually denoted by symbols of the form $\leq$, $\subseteq$, $\sqsubseteq$ or $\preceq$ and are written in infix notation.

Example 7.6. The subset relation (Def. 5.2) is a partial order. To see this, we must show that the subset relation is: (i.) reflexive, (ii.) antisymmetric and (iii.) transitive.

i.) For every set A, $A \subseteq A$ (see Thm. 5.5.1) so $\subseteq$ is a reflexive relation.
ii.) Antisymmetry holds by Thm. 5.5.2.
iii.) To see that transitivity holds for the $\subseteq$ relation, assume that $A \subseteq B$ and $B \subseteq C$ and show that $A \subseteq C$. Clearly, since every element of A is in B and every element of B is in C, every element of A is in C *i.e.* $A \subseteq C$.

Definition 7.17 (Strict Partial Order)

> If $R \subset A \times B$ is a relation that is reflexive, antisymmetric and transitive we call it a *partial order*.

7.2.2 Products and Sums of Orders

We can construct new partial orders from existing ones by a various kinds of compositions.

Cartesian Product

Definition 7.18 (Odered product) If $\langle P_1, \sqsubseteq_1 \rangle$ and $\langle P_2, \sqsubseteq_2 \rangle$ are posets, then $\langle P_1 \times P_2, \sqsubseteq \rangle$ is a poset, where

$$\langle x, y \rangle \sqsubseteq \langle z, w \rangle \stackrel{\text{def}}{=} x \sqsubseteq_1 z \wedge y \sqsubseteq_2 w$$

This is a pointwise ordering.

Lexicographic Product

Definition 7.19. If
If $\langle A, \sqsubseteq_1 \rangle$ and $\langle B, \sqsubseteq_2 \rangle$ are posets, then $\langle A \times B, \sqsubseteq \rangle$ is a poset where

$$\langle x, y \rangle \sqsubseteq \langle z, w \rangle \stackrel{\text{def}}{=} x \sqsubset_1 z \vee x = z \wedge y \sqsubseteq_2 w$$

Chapter 8

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.

8.1 Functions

Definition 8.1 (function) A *function* from A to B is a relation ($f \subseteq A \times B$) satisfying the following properties,

$$\begin{array}{rl} i.) & \forall x : A.\exists y : B.\langle x, y\rangle \in f \\ ii.) & \forall x : A.\forall y, z : B.(\langle x, y\rangle \in f \wedge \langle x, z\rangle \in f) \Rightarrow y = z \end{array}$$

Relations having the first property are said to be *total* and relations satisfying the second property are said to be *functional* or to satisfy the *functionality* property .

Remark 8.1. Since we usually write $f(x) = y$ for functions instead of $\langle x, y\rangle \in f$, we can restate these properties in the more familiar notation as follows.

$$\begin{array}{rl} i.) & \forall x : A.\exists y : B.f(x) = y \\ ii.) & \forall x : A, y, z : B.(f(x) = y \wedge f(x) = z) \Rightarrow y = z \end{array}$$

There is some danger in using the notation $f(x) = y$ if we do not know that f is a function.

We denote 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 $f : A \to B$ or $f \in A \to B$.

Definition 8.2 (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

$$rng(f, A, B) \stackrel{\text{def}}{=} \{y \in B | \exists x : A.f(x) = y\}$$

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 simply $rng(f)$ to denote it range if A and B are clear from the context.

It is worth considering what it means if the domain and or codomain of a function are empty.

Lemma 8.1. [Empty Domain] For every set A, $\forall f. f \in \emptyset \to A \Leftrightarrow f = \emptyset$
Proof: Choose an arbitrary set A and show both directions:
($\Rightarrow$) Suppose f is a function in $emptyset \to A$, then $f \subseteq A \times \emptyset$, so $f = \emptyset$.
($\Leftarrow$) Assume $f = \emptyset$ and show $f \in \emptyset \to A$ We must show three things: i.) $f \subseteq \emptyset$ but $f = \emptyset$ so this is trivially true; ii.) f is functional – which is vacuously true since the domain is empty; and iii.) that f is total, which is also vacuously true.
□

Lemma 8.2 (Empty Codomain) For every set A

$$\forall f : A \to \emptyset.\ A = \emptyset$$

Proof: Suppose f is a function in $A \to \emptyset$, then $f \subseteq A \times \emptyset$ and since $A \times \emptyset = \emptyset$, $f \subseteq \emptyset$, *i.e.* $f = \emptyset$. But also, f is a function so it is both functional and total. The emptyset is trivially functional. But notice, that for f to be total to following property must hold.

$$\forall x : A. \exists y : \emptyset. \langle x, y \rangle \in f$$

More specifically, since $f = \emptyset$ we must show

$$\forall x : A. \exists y : \emptyset. \langle x, y \rangle \in \emptyset$$

This is vacuously true if $A = \emptyset$ and is false otherwise, thus it must be the case that $A = \emptyset$.
□

Corollary 8.1. $\forall f :\in \emptyset \to \emptyset.\ f = \emptyset$

8.2 Extensionality (equivalence for functions)

We define function equality extensionally, as equality on the underlying sets as follows:

Definition 8.3 (extensionality) For functions $f, g : A \to B$,

$$f = g \stackrel{\text{def}}{=} \forall x : A. f(x) = g(x)$$

Thus, functions are equal if they are the same set of pairs. Since they are, by definition functional, this amounts to checking that they agree on every input.

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. When we think of $\mathcal{P}_a$ and $\mathcal{P}_2$ as sets $\mathcal{P}_1 = \mathcal{P}_2$ must be true if they both actually implement sorting correctly. So for example $\langle [2;1;4], [1;2;4] \rangle$ is a pair in both $\mathcal{P}_1$ and in $\mathcal{P}_2$. However, the two programs may have significantly different run-time complexities, so this is a way in which they are not equal. Program $\mathcal{P}_1$ may implement the merge sort algorithm which has $O(n\ log\, n)$ 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. Properties such as run time or length of a program that make distinctions other than the one made by extensionality are called *intensional properties*.

8.3 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.

8.3.1 Restrictions and Extensions

Definition 8.4. If $f \in A \to B$ and $A' \subseteq A$, then $f \cap (A' \times B)$ is called the *restriction of f to A'* and is sometimes written f/A' or $f \downarrow A'$.

Exercise 8.1. Prove functions are closed under restrictions, *i.e.* if $f \downarrow A'$ is a restriction of $f : A \to B$ to A' where $A' \subseteq A$, then $f \downarrow A'$ is a function.

Definition 8.5. If $g \in A' \to B$ is the restriction of $f \in A \to B$ then we say f *is the extension of g*.

Lemma 8.3. $f \in A \to B$ is an extension of $g \in A' \to B$ iff $g \subseteq f$.

8.3.2 Composition of Functions

Recall that functions are simply relations that are both functional and total. This means the operation of composition for relations (Def. 6.6.8) can be applied to functions as well. Given functions $f : A \to B$ and $g : b \to C$, their composition is the function $g \circ f$ which is simply calculated by applying f and then applying g.

Right away we must ask ourselves whether the relation obtained by composing functions results in a relation that is also a function.

Lemma 8.4 (function composition) Consider functions $f : A \to B$ and $g : B \to C$, the following picture illustrates the situation.

$$A \xrightarrow{f} B \xrightarrow{g} C$$

By composition of relations, we know there is a relation $(g \circ f) \subseteq A \times C$ which we claim is also a function *i.e.*

$$(g \circ f) \in A \to C$$

Proof: To show $(g \circ f) \in A \to C$, we must show

$$\begin{array}{ll} i.) & \forall x : A.\, \forall y, z : C.\, \langle x, y\rangle \in (g \circ f) \wedge \langle x, z\rangle \in (g \circ f) \Rightarrow y = z \\ ii.) & \forall x : A.\, \exists y : C.\, \langle x, y\rangle \in (g \circ f) \end{array}$$

$(i.)$ Assume $x \in A$ and $y, z \in C$ for arbitrary x, y and z. Also, assume $\langle x, y\rangle \in (g \circ f)$ and $\langle x, z\rangle \in (g \circ f)$ and show $y = z$. By the definition of composition, we know there is a $w \in B$ such that $f(x) = w$ and $g(w) = y$. Similarly we know there is a $\hat{w} \in B$ such that $f(x) = \hat{w}$ and $g(\hat{w}) = z$. Now, since f is a function we know $w = \hat{w}$ and similarly, since g is a function $g(w) = g(\hat{w})$ so $x = y$ and we have shown that $(g \circ f)$ is functional.
$(ii.)$ Assume $x \in A$ and show $(*)$ $\exists y : C.\, \langle x, y\rangle \in (g \circ f)$. Now, since f is a function in $A \to B$ it is total and so there is a $w \in B$ such that $f(x) = w$. Similarly, since g is a function in $B \to C$ there is a $z \in C$ such that $g(w) = z$. But then $\langle x, z\rangle \in (g \circ f)$ and so use z as the witness for y in $(*)$.
□

Remark 8.2. Having proved this theorem we say, *functions are closed under composition* . That is, we preserve the property of being a function when we compose two functions.

In general, this question is rather fundamental and can be asked in many contexts.

> *When is a mathematical structure closed with respect to some operation on it?*

By "closed with respect to", we mean that applying the operation preserves the property of having a particular structure. In the case of functions and the composition operation, the property we are considering is whether a relation is a function and the question "When is the composition of functions also a function?" is answered by the previous lemma, *Always.*

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.

Remark 8.3. Note that because the composition of relations is associative (see Thm. 6.6.1.) the associativity of function composition is obtained for free. We have included a direct proof of the associativity for function composition here to illustrate the difference in the proofs. This is a case where the proof for the special case (function composition) is easier than the more general case (relation composition.)

Theorem 8.1 (Associativity of function composition) 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.

Lemma 8.5 (Identity function) If A is a set, Δ_A is the identity function on A *i.e.* Δ_A is a function and $\forall x : A.\Delta_A(x) = x$.

Exercise 8.2. Prove Lemma 8.5.

Remark 8.4. We will sometimes write Id_A for Δ_A when we are thinking of the relation as the identity function on A.

Theorem 8.2 (Left Right identity lemma) For any sets A and B, and any function $f : A \to B$, Id_A is a left identity for f and Id_B is a right identity for f.

$$Id_B \circ f = f \quad \text{and} \quad f \circ Id_A = f$$

Proof: To show two functions are equal, we apply extensionality, choosing an arbitrary x.

$$(f \circ Id_A)(x) = f(Id_A(x)) = f(x)$$

Thus Id_A is a right identity for $\circ$. Similarly,

$$(Id_B \circ f)(x) = Id_B(f(x)) = f(x)$$

Thus $(f \circ Id_A) = f$ and $Id_B \circ f = f$ and the theorem has been shown.
□

8.3.3 Inverse

Given a relation $R \subseteq A \times A$, recall Definition 6.6.6 of the inverse relation

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

Now, consider a function $f : A \to B$. Since f is a relation, the relation f^{-1} exists; but is it a function?

We ask the question, "When are functions closed under the inverse operation? *i.e.* when is the inverse of a function still a function. We can try to begin to answer the question by considering cases where the inverse might fail to be a function.

Example 8.1. 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 8.2. 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.

Functions that rule out the behavior described in Example 8.1 are said to be *one-to-one* or are called *injections*. Functions that rule out the behavior described in Example 8.2 are the *onto* functions which are also called *surjections*.

8.4 Properties of Functions

In the previous section we have analyzed what conditions might rule out the possibility that a the inverse of a functions is itself a function. In this section, we formalize those conditions as logical properties. Functions that satisfy these properties have, in some sense, more structure than functions that don't.

8.4.1 Injections

Definition 8.6 (injection, one-to-one) A function $f : A \to B$ is an *injection* (or *one-to-one*) if every element of B is mapped to my at most one element of A. 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.

Theorem 8.3 (Composition of injections) For sets A, B and C, and functions $f : A \to B$ and $g : B \to C$, then

a.) if f and g are injections, then $g \circ f$ is an injection, and

b.) If $g \circ f$ is an injection then f is an injection.

Proof:

Proof of part (a): Since f and g are injections we know

$$\begin{array}{ll} i.) & \forall x, y : A. f(x) = f(y) \Rightarrow x = y \\ ii.) & \forall x, y : B. g(x) = g(y) \Rightarrow x = y \end{array}$$

We must show that $(g \circ f)$ is an injection, *i.e.* that:

$$\forall x, y : A.\ (g \circ f)(x) = (g \circ f)(y) \Rightarrow x = y$$

We choose arbitrary x and y from the set A and assume $(g \circ f)(x) = (g \circ f)(y)$ to show $x = y$. Now, by definition of function composition $(g \circ f)(x) = g(f(x))$ and $(g \circ f)(y) = g(f(y))$. Using $f(x)$ for x and $f(y)$ for y in $(ii.)$ we get that $f(x) = f(y)$, but then, by the fact that f is an injection $(i.)$ we know $x = y$.

Proof of part (b): Left as an exercise.
□

8.4.2 Surjections

Definition 8.7 (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.

$$\forall y : B.\ \ \exists x : A.\ \ f(x) = y$$

Corollary 8.2 (surjection characerization lemma) A function is a surjection if and only if $codom(f) = rng(f)$.

Exercise 8.3. Prove Corollary 8.2.

Theorem 8.4 (Composition of surjections) For sets A, B and C, and functions $f : A \to B$ and $g : B \to C$,

a.) if f and g are surjections, then $g \circ f$ is a surjection, and

b.) If $g \circ f$ is a surjection then g is a surjection.

Proof:

Proof of part (a): Since f and g are surjections we know

$$\begin{array}{ll} i.) & \forall y : B.\exists x : A.\ f(x) = y \\ ii.) & \forall y : C.\exists x : B.\ g(x) = y \end{array}$$

We must show that $(g \circ f)$ is an surjection, *i.e.* that:

$$\forall y : C.\exists x : A.\ (g \circ f)(x) = y$$

Choose an arbitrary $y \in C$ and show that there exists an $x \in A$ such that $(g \circ f)(x) = y$. By $(ii.)$, there is some $z \in B$ such that $g(z) = y$. Using z for y in $(i.)$, we get that there is an $x \in A$ such that $f(x) = z$. Now, we have that $f(x) = z$ and $g(z) = y$ so we know that $g(f(x)) = y$, in particular, we know that $(g \circ f)(x) = y$. Thus, we have shown that, if you choose an arbitrary $y \in C$ there exists an $x \in A$ such that $(g \circ f)(x) = y$.

Proof of part (b): Left as an exercise.
□

8.4.3 Bijections

Definition 8.8 (bijection) A function $f : A \to B$ is a *bijection* if it is both an injection and a surjection. Bijective functions are also said to be *one-to-one and onto.*

Now, going back to the question of when the inverse of a function is a function, we state the following theorem which perfectly characterizes the situation.

Lemma 8.6 (Inverse Characterization) For every function $f \in A \to B$, f is a bijection if and only if the inverse f^{-1} is a function.

Proof: Let f be an arbitrary function in $A \to B$. There are two cases:

($\Rightarrow$) Assume f is a bijection and show f^{-1} is a function.

If f is a bijection, then it is both an injection and a surjection, *i.e.* we assume the following about f.

$$\begin{array}{ll} \forall x, y : A.\ f(x) = f(y) \Rightarrow x = y & \text{(injection)} \\ \forall y : B.\ \exists x : A.\ f(x) = y & \text{(surjection)} \end{array}$$

Now, to show f^{-1} is a function from B to A, we must show that it is both total and functional[1].

i.) $\forall x{:}B.\, \exists y{:}A.\, \langle x, y\rangle \in f^{-1}$ (total)

ii.) $\forall x{:}B.\, \forall y, z{:}A.\, \langle x, y\rangle \in f^{-1} \wedge \langle x, z\rangle \in f^{-1} \Rightarrow y = z$ (functional)

(i.) We prove that f^{-1} is total as follows: Choose an arbitrary element of B, call it z and show that $\exists y : A.\langle z, y\rangle \in f^{-1}$. Now, since f is surjective, we know that $\exists x : A.f(x) = z$, so we assume there is an x such that $x \in A$ and such that that $f(x) = z$. Then, $\langle x, z\rangle \in f^{-1}$. Thus, let y in $\exists y : A.f^{-1}(z) = y$ be x. Since we have just argued that $\langle z, x\rangle \in f^{-1}$ and since $x \in A$ we have finished the proof that f^{-1} is total.

(ii.) To see that f^{-1} is functional, consider the following argument. Choose an arbitrary element of B, call it x and let y and z be arbitrary elements of A. We must show $\langle x, y\rangle \in f^{-1} \wedge \langle x, z\rangle \in f^{-1} \Rightarrow y = z$ so we assume $\langle x, y\rangle \in f^{-1}$ and $\langle x, z\rangle \in f^{-1}$ and show $y = z$. But since $\langle x, y\rangle \in f^{-1}$, we know (by the definition of f^{-1}) that $f(y) = x$ and similarly $f(z) = x$. Now, since we started by assuming f is an injection, it must be that $y = z$.

This completes the proof that if f is a bijection then f^{-1} is a function.

($\Leftarrow$) Assume f^{-1} is a function and show f is a bijection.

The proof is left as an exercise.

□

Remark 8.5. Note that we used the fact that f was surjective to prove that f^{-1} was total and we used the fact that f was injective to prove that f^{-1} was functional. Looking at the formulas for these properties above, we see the similarity of their forms – so it makes perfect sense that we can use them in this way.

Exercise 8.4. Prove the ($\Leftarrow$) direction of Theorem 8.6.

Lemma 8.7 (inverse-bijection) For every function $f \in A \to B$, if f is a bijection then f^{-1} is a bijection as well.

Exercise 8.5. Prove Lemma 8.7.

[1]Since we are trying to prove that f^{-1} is a function, and we do not know that it is yet, we use the relational notation to avoid confusion *e.g.* the notation $f^{-1}(x) = y$ suggests that there is a unique y such that $\langle x, y\rangle \in f^{-1}$; however, until we have shown f^{-1} is a function we do not know this to be true.

Exercise 8.6. Prove that Δ_A is a function and is bijective. We call this the identity function.

Lemma 8.8 (Inverse) For every function $f \in A \to B$, f is a bijection if and only if the inverse f^{-1} is a function.

Theorem 8.5 (Schröder-Bernstein) If A and B are sets and $f : A \to B$ and $g : B \to A$ are injections, then there exists a function $h \in A \to B$ that is a bijection.

See [4] for a proof.

Note that if f and g are bijections then both f and g are surjections and both are injections. Since composition preserves these properties (see Lemma 8.3and Lemma 8.4) we have the following.

Corollary 8.3 (Composition of bijections) For sets A, B and C, and functions $f : A \to B$ and $g : B \to C$, if f and g are bijections, then so is $g \circ f$.

The proof of following theorem shows how to lift bijections between pairs of sets to their Cartesian product.

Theorem 8.6. For arbitrary sets A, B, A' and B' the following holds:

$$A \sim A' \wedge B \sim B' \Rightarrow (A \times B) \sim (A' \times B')$$

Proof: Assume $A \sim A'$ and $B \sim B'$ are witnessed by the bijections $g : A \to A'$ and $h : B \to B'$. We must construct a bijection (say f) from $(A \times B) \to (A' \times B')$. We define f as follows:

$$f(\langle x, y\rangle) = \langle g(x), h(y)\rangle$$

f injective: Now, to see that f is an injection, we must show that for arbitrary pairs $\langle a, b\rangle, \langle c, d\rangle \in A \times B$ that:

$$f(\langle a, b\rangle) = f(\langle c, d\rangle) \Rightarrow \langle a, b\rangle = \langle c, d\rangle$$

Assume $f(\langle a, b\rangle) = f(\langle c, d\rangle)$ and show $\langle a, b\rangle = \langle c, d\rangle$. But by the definition of f we have assumed $\langle g(a), h(b)\rangle = \langle g(c), h(d)\rangle$. Thus, by equality on ordered pairs, we know $g(a) = g(c)$ and $h(b) = h(d)$. Now, since g and h are both injections we know $a = c$ and $b = d$ and so we have shown that $\langle a, b\rangle = \langle c, d\rangle$.

f surjective: To see that f is a surjection, we must show that

$$\forall \langle c, d\rangle : A' \times B'. \exists \langle a, b\rangle : (A \times B). f(\langle a, b\rangle) = \langle c, d\rangle$$

Choose an arbitrary pair $\langle c, d\rangle \in A' \times B'$. Then we claim the pair $\langle g^{-1}(c), h^{-1}(d)\rangle$ is the witness for the existential. To see that it is we must show that $f(\langle g^{-1}(c), h^{-1}(d)\rangle) = \langle c, d\rangle$. Here is the argument.

$$
\begin{aligned}
& f(\langle g^{-1}(c), h^{-1}(d)\rangle) \\
&= \langle g(g^{-1}(c)), h(h^{-1}(d))\rangle \\
&= \langle (g \circ g^{-1})(c), (h \circ h^{-1})(d)\rangle \\
&= \langle Id_{A'}(c), Id_{B'}(d)\rangle \\
&= \langle c, d\rangle
\end{aligned}
$$

□

8.5 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.

Chapter 9

Cardinality and Counting

9.1 Cardinality

The term *cardinality* refers to the relative "size" of a set.

Definition 9.1 (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 *equipotent*.

Exercise 9.1. Prove that the relation of equal cardinality is an equivalence relation. Specifically, show for arbitrary sets A, B, and C that the following hold:

i.) $|A| = |A|$

ii.) $|A| = |B| \Rightarrow |B| = |A|$

iii.) $(|A| = |B| \wedge |B| = |C|) \Rightarrow |B| = |A|$

This is easy if you study the theorems related to bijections their inverses and compositions.

Next, we use the theorem to show a (perhaps) rather surprising result, that $\mathbb{N}$ has the same cardinality as the set of *Even* numbers, even though half the numbers are not there in *Even*.

Definition 9.2 (even) $Even = \{x{:}\mathbb{N} | \exists y : \mathbb{N}.\ x = 2y\}$.

Theorem 9.1. $|\mathbb{N}| = |Even|$.

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 [17] and is sometimes called *Galileo's paradox* [52] after the Italian scientist Galileo Galilei (1564 – 1642).

Definition 9.3. $Squares = \{x : \mathbb{N} \mid \exists y : \mathbb{N}.\ x = y^2\}$.

Exercise 9.2. Prove that $|\mathbb{N}| = |Squares|$.

Definition 9.4 (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 $|A| \leq |B|$.

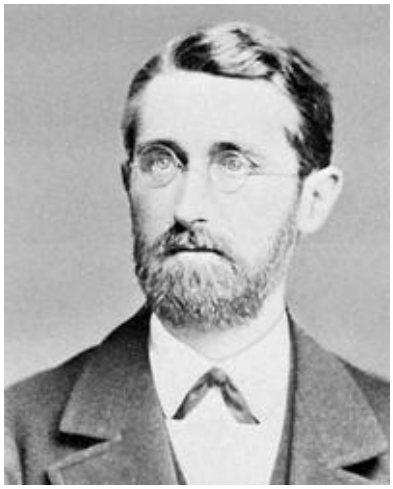
Definition 9.5 (strictly smaller cardinality) The cardinality of a set A is less than the cardinality of B iff there exists an injection $f : A \to B$ and there is no bijection from A to B. In this case we write $|A| < |B|$. Formally,

$$|A| < |B| \stackrel{\text{def}}{=} |A| \leq |B| \wedge |A| \neq |B|$$

The following theorem is a corollary of Thm. 8.8.5.

Theorem 9.2 (Schröder Bernstein) For all sets A and B, if $|A| \leq |B|$ and $|B| \leq |A|$ then $|A| = |B|$.

9.2 Infinite Sets



Richard Dedekind

Richard Dedekind (1831-1916) was a German mathematician who made numerous contributions in establishing the foundations of arithmetic and number theory

The following definition of infinite is sometimes called Dedekind infinite after the mathematician Richard Dedekind (1831-1916) who first formulated it. This characterization of infinite sets may be somewhat surprising because it does not mention natural numbers or the notion of finiteness.

Definition 9.6 (Dedekind infinite) A set A is *infinite* iff there exists a function $f : A \to A$ that is one-to-one but not onto.

Lemma 9.1. $\mathbb{N}$ is infinite.

Proof: Consider the function $f(n) = n + 1$. Clearly, f is one-to-one since if $f(x) = f(y)$ for arbitrary 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. □

Theorem 9.3. If a set A is infinite then, for any set B, if $A \sim B$, then B is infinite.

Proof: If A is infinite, then we know there is a function $f : A \to A$ that is one-to-one but no onto. Also, since $A \sim B$, there is a bijection $g : A \to B$. To show that B is infinite, we must show that there is a function $h : B \to B$ that is one-to-one but not onto. We claim $h = g \circ f \circ g^{-1}$ is such a function.

Now, to show that h is an injection (one-to-one) we recall (Theorem 8.8.3)that the composition of injections is an injection. We assumed that f and g were both injections and to see that g^{-1} is an injection as well, we cite Lemma 8.**??** that says that if g is a bijection then g^{-1} is a bijection as well. Since bijections are also injections, we have shown that h is an injection.

To show that h is not a surjection it is enough to show that

$$i.)\ existsy : B.\forall x : Bh(x) \neq y$$

We assumed f is not onto, thus,

$$\exists y' : A. \forall x : A. f(x) \neq y'$$

i.e. there is at least one y' in A such that

$$ii.) \forall x : A. f(x) \neq y'$$

Not that since g is a bijection we know the inverse g^{-1} is a function of type $B \to A$. Now, to show *i.)* use $g(y')$ as the witness. We must show:

$$\forall x : B h(x) \neq g(y)$$

Choose an arbitrary $x \in B$ and show $h(x) \neq g(y)$. But to show $\neg(h(x) = g(y)$ we assume $h(x) = g(y)$ and derive a contradiction.

$$h(x) = (g \circ f \circ g^{-1})(x) = g(f(g^{-1}(x))) = g(y')$$

Now, since g is an injection, we know $f(g^{-1}(x)) = y'$. But this is impossible because by *ii.)* we know $f(g^{-1}(x)) \neq y'$.
□

This last theorem provides us an alternative method of showing a set is infinite, specifically, show it has the same cardinality as some set already known to be infinite.

9.3 Finite Sets

A set is finite if we can, in a very explicit sense, count the elements in the set *i.e.* we can put the elements of the set in one-to-one correspondence with some initial prefix of the natural numbers.

Definition 9.7.

$$\{0..k\} = \{k : \mathbb{N} \mid 0 \leq k < n\}$$

Remark 9.1. Note that $\{0..0\} = \emptyset$ and so has 0 elements. $\{0..1\} = \{0\}$ and so has one element. In general $\{0..k\} = \{0, 1, \cdots, k-1\}$ and so has k elements.

Definition 9.8 (finite) A set A is finite iff there exists a natural number n such that $|A| = |\{0..k\}|$. In this case we write $|A| = k$.

$$finite(A) \stackrel{\text{def}}{=} \exists n : \mathbb{N}.\ |A| = |\{0..k\}|$$

The definition says that for a set A to be finite, there must be a natural number k and a bijection (call it f, from A to $\{0..k\}$. Since the mapping is a bijection, it has an inverse mapping $\{0..k\} \to A$ which is also a bijection. Then f^{-1} can be used to enumerate the elements of A.

In particular, note that if f is the bijection witnessing the finiteness of A (showing that it has k elements), the following identities hold:

$$f\, A = \{0\,..\,k-1\} \quad \text{and} \quad A = f^{-1}\{0\,..\,k-1\}$$

Lemma 9.2. For every set A, $|A| = 0 \Leftrightarrow A = \emptyset$

Proof:
($\Rightarrow$) Assume $|A| = 0$, then, there is a bijection $c : A \to \{0..0\}$. But by definition $\{0..=\}\emptyset$ since there are no natural numbers between 0 and -1. This means $c : A \to \emptyset$ which, by Thm. 8.8.2 $A = \emptyset$.
($\Leftarrow$) Assume $A = \emptyset$ and show $|A| = 0$. By definition, we must show a bijection from $A \to \{0..0\}$ *i.e.* a bijection $c : \emptyset \to \emptyset$. By Lemma 8.8.1 $c = \emptyset$ is a function in $\emptyset \to \emptyset$ and it is vacuously a bijection.
□

Now, the bijection f witnessing the fact that A is finite assigns to each element of A a number between 0 and k. Also, the inverse f^{-1} maps numbers between 0 and k to unique elements of A *i.e.* since f^{-1} is itself a bijection, and so is one-to-one, we know that no $i, j \in \{0..k-1\}$ such that $i \neq j$ map to the same element of A. We could enumerate (list) the elements of the set by the following bit of pseudo-code.

```
for i ∈ {0..k − 1} do
  Print (f^{-1}(i))
```

To make sure that there is nothing "between" the finite and infinite sets (*i.e.* that there is no set that is neither finite nor infinite) we would expect the following theorem holds.

Theorem 9.4. A set A is Dedekind infinite iff A is not finite.

Interestingly, it can be shown that proofs of this theorem require the use of the axiom of choice – whose use is beyond the scope of these notes.

9.3.1 Permutations

Definition 9.9. A bijection from a finite set A to itself is called a *permutation*.

If a set A is finite, there exists a $k \in \mathbb{N}$ such that there is a bijection $A \to \{0..k\}$. Call this function $\sharp$, then for each $x \in A$, $\sharp x = i$ for some $i \in \{0..k\}$.

9.4 Cantor's Theorem

Typically, it is harder to prove a set A has strictly smaller cardinality that a set B because 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. Cantor's[1] theorem which says that the power set of every set is strictly larger than the set itself.

Theorem 9.5 (Cantor's Theorem) For every set A, $|A| < |\rho(A)|$.

[1]Georg Cantor (1845–1918) was a German mathematician who developed set theory and established the importance of the ideas of injection, and bijection for counting.

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 injection 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 5.5.3, $\{x\} = \{y\} \Leftrightarrow x = y$, 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 $\mathbb{N}$ (which we proved was infinite) 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.

Corollary 9.1 (Infinity of infinities) There is an unbounded hierarchy of strictly distinct infinite sets.

9.5 Countable and Uncountable Sets

By the corollary 9.1, there are many different infinities and we distinguish infinite sets that are, in some sense, small by classifying them as countable sets and the large sets as being *uncountable*.

Definition 9.10 (Countable) A set A is *countable* iff A is finite or $A \sim \mathbb{N}$. Countable sets are also sometimes said to be *denumerable*.

Trivially, it follows that the set of natural numbers $\mathbb{N}$ is countable.
We have the following lemma characterizing countable sets.

Lemma 9.3. For all sets A, A is countable if and only if there exists a surjection $f : \mathbb{N} \to A$.

The proof in the ($\Rightarrow$) direction is trivial following almost directly from the definition of countable. The proof in the $\Leftarrow$) direction assumes the existence of the surjection and requires us to show it is a bijection, or to use it to construct a mapping from A onto an initial segment of $\mathbb{N}$.

The following theorems may be surprising.

Theorem 9.6 ($\mathbb{Q}$ countable) The set $\mathbb{Q}$ (of rational numbers) is countable.

Theorem 9.7 ($\mathbb{R}$ countable) The set $\mathbb{R}$ (of real numbers) is uncountable.

The proofs are originally due to Cantor.

9.6 Counting

We saw with the notion of cardinality that it is possible to compare the sizes of sets without actually *counting* them. By counting, we mean the process of sequentially assigning a number to each element of the set – *i.e* creating a bijection between the elements of the set some set $\{0..k\}$. This is precisely the purpose of the bijection that witnesses the finiteness of a set A – it counts the elements of the set A. Thus, counting is "finding a function of a certain kind."[2]

Lemma 9.4 (Counting Lemma)

$$\forall j, k : \mathbb{N}.\ (\{0..j\} \sim \{0..k\} \Rightarrow j \leq k$$

The following lemma shows that counting is unique, *i.e.* that it does not matter how you count a finite set, it always has the same cardinality.

Theorem 9.8.

$$\forall A. \forall n, m : \mathbb{N}.\ (|A| = n \wedge |A| = m) \Leftrightarrow n = m$$

A corollary is

Corollary 9.2.

$$\forall i, j : \mathbb{N}.\ \{0..i\} \sim \{0..j\} \Leftrightarrow i = j$$

[2]See Stuart Allen's formalization [2] of discrete math materials, the proofs here are the ones formalized by Allen.

9.6.1 The Pigeonhole Principle

A concept related to the one of uniqueness of counting is the *pigeonhole principle*. In formally, it says that if you have k pigeons and m boxes to put them into, if $m < k$ then at least one of the m boxes must contain at least two pigeons.

There are a few ways to state this theorem. A rather explicit statement is given by the following.

Theorem 9.9 (Pigeonhole Principle)

$$\forall m, n : \mathbb{N}. \forall f : \{0..m\} \to \{0..n\}. n < m \Rightarrow \exists i, j : \{0..m\}.\ i \neq j \wedge f(i) = f(j)$$

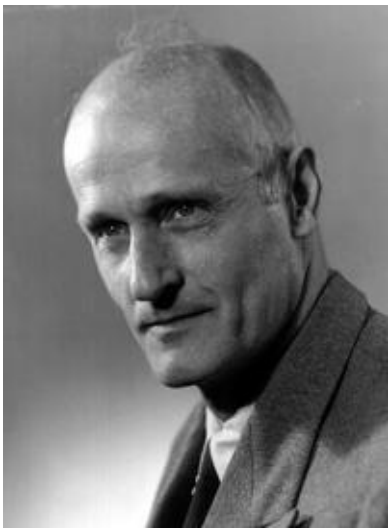
But this is merely saying that at least two elements from the domain must get mapped to the same element (pigeon hole) of the codomain. A more abstract statement is that if $i < j$ then there is no injection from $\{0..j\} \to \{0..i\}$ – *i.e.* if for every function there are always at least a pair of elements from the domain mapped to some single element in the codomain, then there certainly can be no injection.

Theorem 9.10 (Pigeonhole Principle 1)

$$\forall i, j : \mathbb{N}.\ i < j \Rightarrow \neg\exists f : \{0..j\} \to \{0..i\}. Inj(f, \{0..j\}, \{0..i\})$$

Part III

Induction and Recursion



Stephen Cole Kleene

Stephen Cole Kleene (1909–1994) was an American mathematic an and logician who made fundamental early developments in recursion theory, mathematical logic and metamathematics. He was a student of Alonzo Church's at Princeton in the 1930's and went on to the department of Mathematics at the University of Wisconsin in Madison where he stayed until his retirement.

Introduction

The mathematical structures studied in Part II (sets, relations and functions) are formed by building certain kinds of sets and then imposing constraints on those structures *e.g.* we have defined relations as subsets of Cartesian products, we defined functions as a constrained form of Cartesian products, they are relations that satisfy the properties of functionality and totality.

An alternative form of definition[3] definition are those given inductively, meaning that the structures are defined by giving base cases (instances of the structure that do not refer to substructures of the kind being defined, and by combining previously constructed instances of the structure being defined. Examples of mathematical structures having inductive structure are the natural numbers, lists, trees, formulas, and even programs.

Inductive structures are ubiquitous in Computer Science and they go hand in hand with definitions by recursion and inductive proofs. There are many forms of induction (mathematical induction, Notherian induction, well-founded induction [35] and structural induction [54]. In this part of the text, we concentrate on presenting inductive definitions in a style that allows students to define recursive functions and to generate a structural induction principle directly from the definition of the type. Ordinary mathematical induction is often the focus of in discrete mathematics courses but we see it here as simply a special case of structural induction.

In the following sections we present a number of individual inductively defined structures and then follow with a chapter giving the recipie for rolling

[3]In keeping with the foundational idea that all mathematical structures are definable as sets, there is, of course, a purely set theoretic form of definition for inductive structures.

your own inductive defintions, we show how to define functions by recursion on those definitions and show how to syunthesize an induction principle for the new inductive structure.

Readers who may have skipped Chapter 1 (which is labelled as optional) might read Section 1.3 about the form of inductive definitions used here.

Chapter 10

Natural Numbers

This memoir can be understood by any one possessing what is usually call good common sense; no technical philosophic, or mathematical, knowledge is in the least degree required. But I feel conscious that many a reader will scarcely recognize in the shadowy forms which I bring before him his numbers which all his life long have accompanied him as faithful and familiar friends; he will be frightened by the long series of simple inferences corresponding to our step-by-step understanding, by the matter-of-fact dissection of the chains of reasoning on which the laws of numbers depend, and will become impatient at being compelled to follow out proofs for truths which to his supposed inner consciousness seem at once evident and certain.

Richard Dedekind from the Preface to the First Edition of *The Nature and Meaning of Numbers*, translated by Wooster Woodruff Beman, in *Essays on the Theory of Numbers*, Open Court Publishing, Chicago, 1901.

Leopold Kronecker (1823 - 1891)

Leopold Kronecker

The German mathematician Leopold Kronecker 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 the other mathematical structures have been constructed by men. Similarly, the philosopher Immanuel Kant (1742 – 1804) and mathematician Luitzen Egbertus Jan Brouwer (1881 - 1966) both believed that understanding of natural numbers is somehow innate; that it arises from intuition about the human experience of time as a sequence of moments. [1] In any case, it would be difficult to argue against the primacy of the natural numbers among mathematical structures.

[1] Interestingly, Kant also believed that geometry was similarly primitive and our intuition of it arises from our experience of three dimensional space. The discovery in the 19th century of non-Euclidean geometries [**?**, 34] makes this idea seem quaint by modern standards.

10.1 Peano Axioms

Giuseppe Peano

Giuseppe Peano (1858–1932), an Italian mathematician and philosopher who, among other accomplishments, gave an axiomatic presentation of arithmetic.

The *Peano axioms* are named for Giuseppe Peano (1858–1932), an Italian mathematician and philosopher. Peano first presented his axioms [40] of arithmetic in 1889, though in a later paper Peano credited Dedekind [7] with the first presentation of the axioms. We still know them as Peano's axioms.

Definition 10.1 (Peano axioms) Let $\mathbb{N}$ be a set where $\mathbf{0}$ is a constant symbol, $\mathbf{s}$ be a function of type $\mathbb{N} \to \mathbb{N}$ (call it the successor function) and P is any property of natural numbers, then following are Peano's axioms.

$$
\begin{array}{rl}
i.) & \mathbf{0} \in \mathbb{N} \\
ii.) & \forall k : \mathbb{N}.\ \mathbf{s}k \in \mathbb{N} \\
iii.) & \forall k : \mathbb{N}.\ \mathbf{0} \neq \mathbf{s}k \\
iv.) & \forall j, k : \mathbb{N}.\ j = k \Leftrightarrow \mathbf{s}j = \mathbf{s}k \\
v.) & (P[\mathbf{0}] \wedge \forall k : \mathbb{N}.\ P[k] \Rightarrow P[\mathbf{s}k]) \Rightarrow \forall n : \mathbb{N}.\ P[n]
\end{array}
$$

Axioms $(i.)$ and $(ii.)$ say $\mathbf{0}$ is a natural number and if k is a natural number then so is $\mathbf{s}k$. We call $\mathbf{s}$ the *successor function* and $\mathbf{s}k$ is the successor of k. Axiom $(iii.)$ says that $\mathbf{0}$ is not the successor of any natural number and axiom iv is a kind of monotonicity property for successor, it says the successor function preserves equality. Axiom $(v.)$ is the induction principle which is the main topic of discussion of this chapter, see Section 10.3.

So, there are two ways to construct a natural number, either you write down the constant symbol $\mathbf{0}$ or, you write down a natural number (say k) and then you apply the successor function $\mathbf{s}$ which has type $\mathbb{N} \to \mathbb{N}$ to the k to get the number $\mathbf{s}k$.

Thus, $\mathbb{N} = \{\mathbf{0}, \mathbf{s0}, \mathbf{ss0}, \mathbf{sss0}, \cdots\}$ are the 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 previously term from the set $\{\mathbf{0}, \mathbf{s0}, \mathbf{ss0}, \cdots\}$.

We typically write natural numbers in decimal notation.

$$\begin{aligned} 0 &= \mathbf{0} \\ 1 &= \mathbf{s\,0} \\ 2 &= \mathbf{s\,s\,0} \\ 3 &= \mathbf{s\,s\,s\,0} \\ &\vdots \end{aligned}$$

You should think of 3 as a (better) notation for the natural number $\mathbf{s\,s\,s\,0}$.

10.2 Definition by Recursion

We have defined functions by recursion earlier in these notes (*e.g.* the *val* function given in Chapter 2 Def. 2.8). The idea of defining functions by recursion on the structure of one of its arguments presented here is the same. To make a definition by recursion "on the structure" of the natural numbers, we must specify the behavior of the function on inputs by considering the possible cases: the input (or one of them) is $\mathbf{0}$ or it is of the form $\mathbf{s}k$ for some previously constructed natural number k.

As an example, consider the definition of addition over the natural numbers by recursion on the structure of the first argument.

Definition 10.2 (Addition)

$$\begin{aligned} add(\mathbf{0}, k) &= k \\ add(\mathbf{s}n, k) &= \mathbf{s}(add(n, k)) \end{aligned}$$

We will use ordinary infix notation for addition with the symbol +; thus $add(m, n)$ will be written $m+n$. Using this standard notation we take the chance that the reader will assume + has all the properties expected for addition. It turns out that it does, but until we prove a property is true for *this* definition we can not use the property. In the infix notation, the definition would appear as follows:

$$\begin{aligned} \mathbf{0} + k &= k \\ \mathbf{s}n + k &= \mathbf{s}(n + k) \end{aligned}$$

Example 10.1. To add $2 + 3$ using the definition, we compute as follows:

$$\mathbf{ss0} + \mathbf{sss0} = \mathbf{s}(\mathbf{s0} + \mathbf{sss0}) = \mathbf{ss}(\mathbf{0} + \mathbf{sss0}) = \mathbf{ss}(\mathbf{sss0}) = \mathbf{sssss0}$$

Multiplication is defined as iterated addition by recursion on the structure of the first argument.

Definition 10.3 (Multiplication)

$$\begin{aligned} mult(\mathbf{0}, k) &= \mathbf{0} \\ mult(\mathbf{s}n, k) &= mult(n, k) + k \end{aligned}$$

We will use ordinary infix notation for multiplication with the symbol $\cdot$; thus $mult(m, n)$ will be written $m \cdot n$. We will also sometimes just write mn omitting the symbol $\cdot$. In the infix notation, the definition appears as follows:

$$\begin{aligned} \mathbf{0} \cdot k &= k \\ \mathbf{s}n \cdot k &= (n \cdot k) + k \end{aligned}$$

Example 10.2. So, to multiply $2 \cdot 3$,

$$\begin{aligned} \mathbf{ss0} \cdot \mathbf{sss0} &= (\mathbf{s0} \cdot \mathbf{sss0}) + \mathbf{sss0} \\ &= ((\mathbf{0} \cdot \mathbf{sss0}) + \mathbf{sss0}) + \mathbf{sss0} \\ &= (\mathbf{0} + \mathbf{sss0}) + \mathbf{sss0} \\ &= \mathbf{sss0} + \mathbf{sss0} \\ &= \mathbf{ssssss0} \end{aligned}$$

We define exponentiation by recursion on the structure of the exponent.

Definition 10.4 (exponentiation)

$$\begin{aligned} n^{\mathbf{0}} &= \mathbf{s\,0} \\ n^{(\mathbf{s}\,k)} &= n^k \cdot n \end{aligned}$$

The definitions of addition, multiplication and exponentiation are all instances of a syntactic pattern of definition that is called *definition by recursion.* It turns out that any definition that follows this pattern is guaranteed to be a function. This might be seen as an early example of a design pattern [**?**].

Theorem 10.1 (Definition by Recursion) Given a set A and a function $g \in (\mathbb{N} \times A) \to A$, and an element $a \in A$, definitions having the following form:

$$\begin{aligned} f(\mathbf{0}) &= a \\ f(\mathbf{s}k) &= g(k, f(k)) \end{aligned}$$

result in well-defined functions, $f \in \mathbb{N} \to A$.

The proof of the theorem justifying definition by recursion is beyond the scope of these notes (See [30] or [31, pp.45]); however, the theorem justifies definitions of the kind just given for addition, multiplication and exponentiation.

The theorem says that if a definition follows a particular syntactic form, you are justified in claiming you have defined a function; definitions that follow the pattern are guaranteed to be functional and total. Pretty good stuff, no need to prove that each new definition is a function in $\mathbb{N} \to A$, just follow the pattern specified by the theorem and you are guaranteed your definition is a function. And note that the property of being total (*i.e.* that every input gets

mapped to some output) guarantees that definitions which match the pattern are guaranteed to halt on all inputs. In [31] Mac Lane notes that one can work the other way around, one can take the definition by recursion as an axiom and derive the Peano axioms as theorems.

The arithmetic functions we defined above all take two arguments. We state a corollary of the Theorem 11.1 for binary functions defined by recursion on the structure of their first arguments.

Corollary 10.1 (Definition by recursion (for binary functions)) Given sets A and B and a function $g \in (\mathbb{N} \times A) \to A$, and a function $h \in B \to A$, and an element $b \in B$, definitions of the following form:

$$f(\mathbf{0}, b) = h(b)$$
$$f(\mathbf{s}n, b) = g(n, f(n, b))$$

result in well-defined functions, $f \in (\mathbb{N} \times B) \to A$.

Using the corollary, we prove that the definitions given above for addition, multiplication and exponentiation of natural numbers are indeed functions.

Theorem 10.2 (Addition is a function) Addition as given by Definition 10.2 is a function of type $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$.
Proof: Recall the definition

$$\mathbf{0} + k = k$$
$$\mathbf{s}n + k = \mathbf{s}(n + k)$$

To prove that addition is a function of the specified type we show how it fits the pattern given by Corollary 10.1. To do so we must specify the sets A and B and the functions h and g and the element $b \in B$. In this case, let $A = \mathbb{N}$ and $B = \mathbb{N}$. Let $b = k$. Since $B = \mathbb{N}$ and $k \in \mathbb{N}$ it is an acceptable choice for b. The function $h : \mathbb{N} \to \mathbb{N}$ is just the identity function, $h(k) = k$. The function $g \in (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ is the function $g(j, k) = \mathbf{s}k$. Thus, the operation of addition is, by Corollary 10.1 a function in $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$.
□

Theorem 10.3 (Multiplication is a function) Multiplication as given by Definition 10.3 is a function of type $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$.
Proof: Recall the definition:

$$\mathbf{0} \cdot k = \mathbf{0}$$
$$\mathbf{s}n \cdot k = (n \cdot k) + k$$

Multiplication fits the pattern of Corollary 10.1 as follows: let $A = \mathbb{N}$ and $B = \mathbb{N}$ and $k = \mathbf{0}$. The function $h : \mathbb{N} \to \mathbb{N}$ is just the constant function $h(k) = \mathbf{0}$. The function $g \in \mathbb{N} \to \mathbb{N}$ is the function that adds k to the input of g, so $g(m, n) = n + k$. We have just proved that addition is a function and so $g \in (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$. Thus, by Corollary 10.1 multiplication is a function in $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$.

□

Theorem 10.4 (Exponentiation is a function) Exponentiation as given by Definition 10.4 is a function of type $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$.

Exercise 10.1. Prove Theorem 10.4

Exercise 10.2. The Fibonacci function is defined as follows:

$$\begin{aligned} F(0) &= 0 \\ F(\mathbf{s}0) &= 1 \\ F(\mathbf{ss}k) &= F(\mathbf{s}k) + F(k) \end{aligned}$$

Can this be defined using definition by recursion? If so how? If not, why not?

You may have convinced yourself that these definitions look like they "do the right thing" but we will be able to *prove* that they behave in the ways we expect them to using mathematical induction.

10.3 Mathematical Induction

The rule for $\forall R$ provides one method of proving a statement of the form $\forall n : \mathbb{N}.\ \phi$ *e.g.* choose an arbitrary natural number (call it k) and assume $k \in \mathbb{N}$ and show $\phi[n := k]$. But it is not always enough to just choose an arbitrary element, the argument may depend on the structure of the type being quantified over, in this case the natural numbers. Mathematical induction is a principle of proof that takes the structure of the natural numbers into account in the proof.

Peano's axiom (*v.*) (see Definition 10.1) is known as the principle of mathematical induction.

Definition 10.5 (Principle of Mathematical Induction) For a property P of natural numbers we have the following axiom.

$$(P[\mathbf{0}] \wedge \forall k : \mathbb{N}.\ P[k] \Rightarrow P[\mathbf{s}k]) \Rightarrow \forall n : \mathbb{N}.\ P[n]$$

10.3.1 An informal justification for the principle

Suppose you wished to justify the principle of mathematical induction.

$$(P[\mathbf{0}] \wedge \forall k : \mathbb{N}.\ P[k] \Rightarrow P[\mathbf{s}k]) \Rightarrow \forall n : \mathbb{N}.\ P[n]$$

It says, for a property P of natural numbers, to show that $P[n]$ holds for every natural number n, it is enough to show two things:

$$\textit{i.)}\ P[\mathbf{0}] \qquad \text{and} \qquad \textit{ii.)}\ \forall k : \mathbb{N}. P[k] \Rightarrow P[\mathbf{s}k]$$

So, suppose you have accepted proofs of (*i.*) and (*ii.*) but somehow still believe that there might be some $n \in \mathbb{N}$ such that $\neg P[n]$ holds. Since $n \in \mathbb{N}$ it is constructed by n applications of the successor function to 0. You can construct an argument that $P[n]$ *must* hold in $2n+1$ steps. The argument is constructed using (*i.*) and (*ii.*) as follows[2]:

1.	$P[\mathbf{0}]$	you accepted this as (*i.*)
2.	$P[\mathbf{0}] \Rightarrow P[s\mathbf{0}]$	instantiate (*ii.*) with $\mathbf{0}$
3.	$P[\mathbf{s0}]$	modus ponens using 1 and 2
4.	$P[\mathbf{s0}] \Rightarrow P[\mathbf{ss0}]$	instantiate (*ii.*) with $s\mathbf{0}$
5.	$P[\mathbf{ss0}]$	modus ponens using 3 and 4
$\vdots$	$\vdots$	
2n.	$P[\mathbf{s}^{(n-1)}] \Rightarrow P[s^n\mathbf{0}]$	instantiate (*ii.*) with $\mathbf{s}^{(n-1)}$
2n+1.	$P[\mathbf{s}^n\mathbf{0}]$	modus ponens using 2n and 2n+1

Thus, no matter which n is chosen, we can prove $P[n]$ holds in $2n+1$ steps using the base case (*i.*) and the induction step (*ii.*).

10.3.2 A sequent style proof rule

We can derive a sequent style proof rule for mathematical induction by instantiating Peano"s induction axiom and doing as much proof as is possible in schematic form.

$$\Pi_1 : \frac{}{\Gamma, \forall n : \mathbb{N}.\ P[n] \vdash \Delta_1, \forall n : \mathbb{N}.\ P[n], \Delta_2}\ (\text{Ax})$$

$$\dfrac{\dfrac{\Gamma \vdash \Delta_1, P[\mathbf{0}], \Delta_2 \qquad \dfrac{\dfrac{\dfrac{\Gamma, k : \mathbb{N}, P[k] \vdash \Delta_1, P[\mathbf{s}k], \Delta_2}{\Gamma, k : \mathbb{N} \vdash \Delta_1, P[k] \Rightarrow P[\mathbf{s}k], \Delta_2}\Rightarrow\text{R}}{\Gamma \vdash \Delta_1, \forall k : \mathbb{N}.\ P[k] \Rightarrow P[\mathbf{s}k]), \Delta_2}\forall\text{R}}{\Gamma \vdash \Delta_1, P[\mathbf{0}] \wedge \forall k : \mathbb{N}.\ P[k] \Rightarrow P[\mathbf{s}k], \Delta_2}\wedge\text{R} \qquad \Pi_1}{\Gamma, P[\mathbf{0}] \wedge \forall k : \mathbb{N}.\ (P[k] \Rightarrow P[\mathbf{s}k]) \Rightarrow \forall n : \mathbb{N}.\ P[n] \vdash \Delta_1, \forall n : \mathbb{N}.P[n], \Delta_2}(\Rightarrow\text{R})}{\Gamma \vdash \Delta_1, \forall n : \mathbb{N}.P[n], \Delta_2}(\text{Ax})$$

The informal justification is as follows:if you are trying to prove a sequent of the form $\Gamma \vdash \forall m : \mathbb{N}.\, P[m]$, you can add an instance of the principle of mathematical induction to the left side; this is because it is an axiom of Peano arithmetic. After one application of $\forall L$ rule, on the left branch you will be required to show two things: The left branch will be of the form,

$$\Gamma \vdash P[\mathbf{0}] \wedge \forall k : \mathbb{N}.\ P[k] \Rightarrow P[\mathbf{s}k]$$

and the right branch will be an instance of an axiom of the form:

$$\forall m : \mathbb{N}.\, P[m], \Gamma \vdash \forall m : \mathbb{N}.\, P[m]$$

[2]Recall that *modus ponens* is the rule that says that P and $P \Rightarrow Q$ together yield Q.

We can further refine the left branch by applying the $\wedge$R rule which gives two subgoals: One to show $P[0]$ and the other to show $\forall k : \mathbb{N}.\ P[k] \Rightarrow P[\mathbf{s}k]$. This sequent can further be refined by applying $\forall$R and then $\Rightarrow$R.

This yields the following rule[3].

Proof Rule 10.1 (Mathematical Induction)

$$\frac{\Gamma \vdash P[\mathbf{0}] \qquad \Gamma, k \in \mathbb{N}, P[k] \vdash P[\mathbf{s}\,k]}{\Gamma \vdash \forall m : \mathbb{N}.\ P[m]}\ (\mathbb{N}\text{Ind}) \qquad \text{where } k \text{ is fresh.}$$

10.3.3 Some First Inductive Proofs

We have suggested that for k a natural number, $\mathbf{s}k$ (the successor of k) is the same as $k+1$ where 1 is just the decimal representation of $\mathbf{s0}$. We state this as a theorem and present it as the first proof using mathematical induction.

Lemma 10.1 (Successor is add one.)

$$\forall k : \mathbb{N}.\ \mathbf{s}k = k + 1$$

Proof: By mathematical induction on k. The property of k we are to prove is defined as follows:

$$P[k] \stackrel{\text{def}}{=} \mathbf{s}k = k + 1$$

In general, for a formula of the form $\forall k : \mathbb{N}.\ \phi$, the property P can be written as $P[k] \stackrel{\text{def}}{=} \phi$.

As in all proofs by mathematical induction there are two things to show, the base case and the induction step. The forms of these two subgoals are given by the proof rule $\mathbb{N}Ind$ shown above in Def. 10.3.

Base Case: We must show $P[\mathbf{0}]$. We get $P[\mathbf{0}]$ by replacing all free occurrences of k with $\mathbf{0}$ in the body of the definition of P. Thus

$$P[\mathbf{0}] \stackrel{def.P[k]}{=} (\mathbf{s}k = \mathbf{0} + 1)[k := \mathbf{0}] \stackrel{subst}{=} \mathbf{s0} = \mathbf{0} + 1$$

So, at this stage, we must show the following equality holds: $\mathbf{s0} = \mathbf{0} + 1$. To continue the proof, we use the definition of addition and 1 to simplify the right side as follows: $\mathbf{0} + 1 = 1 = \mathbf{s0}$. Thus the left and right sides of the equality are the same and so the base case holds.

Induction Step: To show the induction step holds we assume that for some arbitrary k that $k \in \mathbb{N}$ and furthermore assume that $P[k]$ holds. $P[k]$ is called the induction hypothesis.

$$\mathbf{s}k = k + 1 \qquad (Ind.Hyp.)$$

We must show $P[\mathbf{s}k]$. To get $P[\mathbf{s}k]$ carefully replace all free occurrences of k in the body of P by $\mathbf{s}k$. The result of the substitution is the following equality:

$$\mathbf{ss}k = \mathbf{s}k + 1 \qquad (A)$$

[3]The contexts Δ_1 and Δ_2 on the right side have been omitted to aid readability

To show this, we proceed by computing with the right side using the definition of addition.

$$\mathbf{s}k + 1 = \mathbf{s}(k + 1) \qquad (B)$$

By the induction hypothesis, $\mathbf{s}k = k + 1$ so we replace $k + 1$ by $\mathbf{s}k$ in the right side of (B), this gives

$$\mathbf{s}k + 1 = \mathbf{s}(k + 1) = \mathbf{ss}k$$

But now we have show (A) and so the induction step is completed.
Thus, we have shown that the base case and the induction step hold and this completes the proof.
□

Since we proved in Thm. 10.2 that addition is a function in $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ and by the previous lemma that $sk = k + 1$ we know the successor operation defines a function in $\mathbb{N} \to \mathbb{N}$.

Corollary 10.2 (sucessor is a function)

$$\mathbf{s} \in \mathbb{N} \to \mathbb{N}$$

This proof justifies restating the principle of mathematical induction in the following (perhaps) more familiar form.

Definition 10.6 (Principle of Mathematical Induction (modified)) For a property P of natural numbers we have the following axiom.

$$(P[\mathbf{0}] \wedge \forall k : \mathbb{N}.\ P[k] \Rightarrow P[k + 1]) \Rightarrow \forall n : \mathbb{N}.\ P[n]$$

The following lemma is useful in a number of proofs.

Lemma 10.2 (addition by zero)

$$\forall n, k : \mathbb{N}.\ k = k + n \Rightarrow n = 0$$

Proof: : Choose an arbitrary $n \in \mathbb{N}$ and do induction on k.

$$P[k] \stackrel{\text{def}}{=} k = k + n \Rightarrow n = 0$$

Base Case: Show $P[0]$, *i.e.* that $0 = 0 + n \Rightarrow 0 = n$. Assume $0 = 0 + n$ and show $0 = n$. By the definition of addition $0 + n = n$, so the base case holds.

Induction Step: Assume $k \in \mathbb{N}$, assume $P[k]$ holds and show $P[\mathbf{s}k]$.

$$P[k] : \quad k = k + n \Rightarrow n = 0$$

We must show

$$\mathbf{s}k = \mathbf{s}k + n \Rightarrow n = 0$$

Assume $\mathbf{s}k = \mathbf{s}k + n$ and show $n = 0$. By definition of addition, from the right side of the equality we have: $\mathbf{s}k + n = s(k + n)$ so we know that $\mathbf{s}k = \mathbf{s}(k + m)$. Applying Peano axiom (*iv.*) to this fact we see that $k = k + m$. This formula is the antecedent of the induction hypothesis so we know that $n = 0$ which is what we were to show.

□

Often, we would like to do case analysis on natural numbers, *i.e.* given an assumption that $n \in \mathbb{N}$, we'd like break the proof into two cases, either $n = \mathbf{0}$ or n is a successor ($\exists j : \mathbb{N}.\ n = \mathbf{s}j$). In some way the induction principle says as much and in fact to prove this theorem we need to use induction however, we do not need to utilize the induction hypothesis in the proof! In general, when an induction proof does not use the induction hypothesis, it can be done by case analysis; here we establish this weaker principle using induction.

Lemma 10.3 (Case analysis)

$$\forall n : \mathbb{N}.\ n = 0 \vee \exists j : \mathbb{N}.\ n = \mathbf{s}j$$

Proof: By mathematical induction on n. The property $P[n]$ is defined as follows:

$$P[n] \stackrel{\text{def}}{=} n = 0 \vee \exists j : \mathbb{N}.\ n = \mathbf{s}j$$

Base Case: Show $P[0]$ *i.e.* that $0 = 0 \vee \exists j : \mathbb{N}.\ n = \mathbf{s}j$. By reflexivity of equality $0 = 0$ so the base case holds.

Induction Step: For arbitrary $k \in \mathbb{N}$ assume $P[k]$ and show $P[\mathbf{s}k]$. We do not show the induction hypothesis $P[k]$ because we do not need it. Instead we show $P[\mathbf{s}k]$:

$$\mathbf{s}k = 0 \vee \exists j : \mathbb{N}.\ \mathbf{s}k = \mathbf{s}j$$

By symmetry of equality and by Peano axiom (*iii.*) we know $\mathbf{s}k \neq 0$ so we show $\exists j : \mathbb{N}.\ \mathbf{s}k = \mathbf{s}j$. Use k as the witness and show $\mathbf{s}k = \mathbf{s}k$. To show this we apply Peano axiom (*iv.*) show $k = k$ which is true by the reflexivity of equality.
□

Using this lemma theorem we can derive the following proof rule.

Proof Rule 10.2 (Case Analysis on $\mathbb{N}$)

$$\frac{\Gamma_1, k \in \mathbb{N}, k = 0, \Gamma_2 \vdash \Delta \qquad \Gamma_1, k \in \mathbb{N}, j \in \mathbb{N}, k = \mathbf{s}j, \Gamma_2 \vdash \Delta}{\Gamma_1, k \in \mathbb{N}, \Gamma_2 \vdash \Delta}\ (\mathbb{N}\text{Cases})\ j \text{ fresh.}$$

In cases where induction is not required, case analysis can be used.

Exercise 10.3. Derive Rule 10.2 using Lemma 10.3.

10.4 Properties of the Arithmetic Operators

In Section 10.2 we defined addition, multiplication and exponentiation by recursion on the structure of one of the arguments. We also proved that they are functions. Properties of functions defined by recursion on their structure are invariably established by proofs using mathematical induction. In this section we preset a number of proofs to establish that the arithmetic operators do indeed behave as we expect them to.

The laws for addition and multiplication are given as follows where m, n and k are arbitrary natural numbers.

0 right identity for $+$	$m + 0 = m$
$+$ commutative	$m + n = n + m$
$+$ associative	$m + (n + k) = (m + n) + k$
0 annihilator for $\cdot$	$m \cdot 0 = 0$
1 right identity for $\cdot$	$m \cdot 1 = m$
$\cdot$ commutative	$m{\cdot}n = n{\cdot}m$
$\cdot$ associative	$m{\cdot}(n{\cdot}c) = (m{\cdot}n){\cdot}c$
distributive law	$m \cdot (n + k) = (m \cdot n) + (m \cdot k)$

The fact that **0** is a left identity for addition falls out of the definition for free. That **0** is a right identity requires mathematical induction.

Theorem 10.5 (0 right identity for $+$)

$$\forall n : \mathbb{N}.\, n + 0 = n$$

Proof: By mathematical induction on n. The property P of n is given as:

$$P[n] \stackrel{\text{def}}{=} n + 0 = n$$

Base Case: We must show $P[0]$, *i.e.* that $0 + 0 = 0$ but this follows immediately from the definition of $+$ so the base case holds.

Induction Step: Assume $n \in \mathbb{N}$ and that $P[n]$ holds and show $P[\mathbf{s}n]$. $P[n]$ is the induction hypothesis.

$$P[n] : \quad n + 0 = n$$

Show that $\mathbf{s}n + 0 = \mathbf{s}n$ But by definition of $+$ we know that $\mathbf{s}n + 0 = \mathbf{s}(n + 0)$. By the induction hypothesis $n + 0 = n$ so $\mathbf{s}(n + 0) = \mathbf{s}n$ and the induction step holds.
□

Theorem 10.6 ($+$ is commutative)

$$\forall m, n : \mathbb{N}.\, m + n = n + m$$

Theorem 10.7 ($+$ is associative)

$$\forall m, n, k : \mathbb{N}.\, m + (n + k) = (m + n) + k$$

Theorem 10.8 (1 right identity for $\cdot$)

$$\forall n : \mathbb{N}\, n \cdot 0 = 0$$

Theorem 10.9 ($\cdot$ is commutative)

$$\forall m, n : \mathbb{N}.\, m \cdot n = n \cdot m$$

Theorem 10.10 ($\cdot$ is associative)

$$\forall m, n, k : \mathbb{N}.\, m \cdot (n \cdot k) = (m \cdot n) \cdot k$$

10.4.1 Order Properties

The natural numbers are ordered. Consider the following definition of less than.

Definition 10.7 (Less Than)

$$m < n \stackrel{\text{def}}{=} \exists j : \mathbb{N}.\ m + (j + 1) = n$$

We'd like to establish that the less than relations ($<$) as defined here behaves as expected *i.e.* that it is a strict partial order. Recall the definition from Chap. 7, Def **??** that the relation must be irreflexive (Def. 6.6.13) and transitive (Definition 6.**??**).

Theorem 10.11 ($<$ is irreflexive)

$$\forall n : \mathbb{N}.\ \neg(n < n)$$

Proof: Choose an arbitrary $n \in \mathbb{N}$ and show $\neg(n < n)$. We assume $n < n$ and derive a contradiction. If $n < n$ then, by definition of less than the following holds:

$$\exists j : \mathbb{N}.\ n + (j + 1) = n$$

Let $j \in \mathbb{N}$ be such that $n + (j + 1) = n$. By Lemma 10.2 (addition by zero) we know $j + 1 = 0$. Since $j + 1$ is $\mathbf{s}j$ this contradicts Peano axiom *(iii.)* instantiated with $k = j$.
□

Theorem 10.12 ($<$ is transitive)

$$\forall k, n, m : \mathbb{N}.\ k < n \wedge n < m \Rightarrow k < m$$

Exercise 10.4. Prove Thm. 10.12.

Lemma 10.4 (Zero min)

$$\forall n : \mathbb{N}.\neg(n < \mathbf{0})$$

Proof: Choose arbitrary n. To show $\neg(n < \mathbf{0})$ assume $n < \mathbf{0}$ and derive a contradiction. By definition of less-than, $n < \mathbf{0}$ means

$$\exists j : \mathbb{N}.\ n + (j + 1) = 0$$

Suppose there is such a j. By commutativity of addition and the definition of addition itself, we have the following sequence of equalities.

$$n + (j + 1) = (j + 1) + n = \mathbf{s}j + n = \mathbf{s}(j + n)$$

But then we have $\mathbf{s}(j + n) = \mathbf{0}$ which contradicts Peano axiom *iii.*

□

Theorem 10.13 (Addition Monotone)

$$\forall k, m, n : \mathbb{N}.\ m < n \Rightarrow m + k < n + k$$

Proof:

Choose an arbitrary $k \in \mathbb{N}$ and do induction on m.

$$P[m] \stackrel{\text{def}}{=} \forall n : \mathbb{N}.\ m < n \Rightarrow m + k < n + k$$

Base Case: Show $P[0]$, *i.e.* that

$$\forall n : \mathbb{N}.\ 0 < n \Rightarrow 0 + k < n + k$$

Choose arbitrary $n \in \mathbb{N}$ and assume $0 < n$. Note that by the definition of addition $0 + k = k$, so we must show that $k < n + k$. By definition of less than, we must show:

$$\exists j : \mathbb{N}.\ j \neq 0 \wedge n + k = k + j$$

Use the witness n (for j) and show two things, *i.)* $n \neq 0$ and *ii.)* $n + k = k + n$. Since we assumed $0 < n$, by definition of less than we know that there is some $j \in \mathbb{N}$ such that $j \neq 0$ and $n = 0 + j$. By the definition of addition we know that $n = j$ and hence that $n \neq 0$. By associativity of addition (Thm. 10.7) the second condition holds as well.

Induction Step: Assume $P[m]$ for some $m \in \mathbb{N}$ and show $P[m + 1]$.

$$P[m] : \forall n : \mathbb{N}.\ m < n \Rightarrow m + k < n + k$$

Show

$$\forall n : \mathbb{N}.\ m + 1 < n \Rightarrow (m + 1) + k < n + k$$

Choose arbitrary $n \in \mathbb{N}$. Assume $m + 1 < n$ and show $(m + 1) + k < n + k$. By the induction hypothesis (using n for n) we get,

$$m < n \Rightarrow m + k < n + k$$

Since we assumed $m+1 < n$ we know $m < n$ (if j is a witness for $m+1 < n$ then $j + 1$ is a witness for $m < n$.) Now, since $m < n$ we know $(m + 1) + k < n + k$. To know $m + k < n + k$. To show $(m + 1) + k < n + k$ we must show the following.

$$\exists i : \mathbb{N}.\ i \neq 0 \wedge n + k = (m + 1) + k + i$$

Theorem 10.14 (Exp monotone)

$$\forall n : \mathbb{N}.\ 1 < n \Rightarrow \forall k : \mathbb{N}.\ n^k < n^{k+1}$$

Proof: Choose arbitrary $n \in \mathbb{N}$ and assume $1 < n$. We must show

$$\forall k : \mathbb{N}.\ n^k < n^{k+1}$$

We proceed by induction on k.

$$P[k] \stackrel{\text{def}}{=} n^k < n^{k+1}$$

Base Case: Show $P[0]$, *i.e.* that $n^0 < n^1$. By definition $n^0 = 1$ and

$$n^1 = n^0 \cdot n = 1 \cdot n = n$$

Since we assumed $1 < n$ the base case holds.

Induction Step: Assume $P[k]$ for some $k \in \mathbb{N}$ and show $P[k+1]$.

$$P[k] : n^k < n^{k+1}$$

We must show $n^{k+1} < n^{(k+1)+1}$. Staring on the left side of the inequality we compute as follows:

$$n^{k+1} = n^k + n$$

By the induction hypothesis,$n^k < n^{k+1}$ so, using Theorem **??** we get the following.

$$n^{k+1} = n^k + n < n^{k+1} + n = n^{(k+1)+1}$$

This shows the induction step holds and completes the proof.
□

10.4.2 Iterated Sums and Products

Definition 10.8 (Sum)

$$\sum_{i=k}^{j} f(i) = 0 \qquad \text{if } j < k$$

$$\sum_{i=k}^{j+1} f(i) = f(j+1) + \sum_{i=k}^{j} f(i) \qquad \text{if } (j+1) \geq k$$

Some properties of Sums and Products

Theorem 10.15 (Gauss' identity) For every natural number n, the following identity holds

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof: Our proof is by mathematical induction on n. The predicate we will prove is

$$P[n] \stackrel{\text{def}}{=} \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

We must show the base case and the induction step both hold.
Base Case: We must show $P[0]$, *i.e.*

$$\sum_{i=1}^{0} i = \frac{0(0+1)}{2}$$

But notice, by the definition of summation,

$$\sum_{i=1}^{0} i = 0$$

and also,

$$\frac{0(0+1)}{2} = \frac{0}{2} = 0$$

so the base case holds.
Induction Step: Choose an arbitrary natural number, call it k. We assume $P[k]$ (the induction hypothesis)

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2} \qquad \text{induction hypothesis}$$

and we must show and must show $P[k+1]$ holds.

$$\sum_{i=1}^{(k+1)} i = \frac{(k+1)((k+1)+1)}{2}$$

Starting with the left side, by the definition of summation operator we get the following.

$$\sum_{i=1}^{(k+1)} i = (k+1) + \sum_{i=1}^{(k)} i$$

By the induction hypothesis, we have.

$$(k+1) + \sum_{i=1}^{(k)} i = (k+1) + \frac{k(k+1)}{2}$$

Algebraic reasoning gives us the following sequence of equalities completing the proof.

$$(k+1) + \frac{k(k+1)}{2} = \frac{2k+2}{2} + \frac{k^2+k}{2} = \frac{k^2+3k+2}{2} = \frac{(k+1)(k+2)}{2}$$

□

10.4.3 Applications

In Chapter 9 the pigeonhole principle was presented (without proof) as Theorem **??**. We prove this theorem here using mathematical induction.

Recall, by Definition 9.9.7 that

$$\{0..n\} \stackrel{\text{def}}{=} \{k : \mathbb{N} \mid 0 \leq k < n\}$$

So note that $\{0..0\} = \{\}$ and if $n > 0$ then $\{0..m\} = \{0, 1, \cdots m-1\}$.

Theorem 10.16 (Pigeonhole Principle 1)

$$\forall m, n : \mathbb{N}.\ m > n \Rightarrow \forall f : \{0..m\} \rightarrow \{0..n\}.\neg Inj(f, \{0..m\}, \{0..n\})$$

Proof: By mathematical induction on m. The property we will prove is

$$P(m) \stackrel{\text{def}}{=} \forall n : \mathbb{N}.\ m > n \Rightarrow \forall f : \{0..m\} \rightarrow \{0..n\}.\neg Inj(f, \{0..m\}, \{0..n\})$$

(**Base Case**:) We must show $P(0)$ holds, *i.e.*

$$\forall n : \mathbb{N}.\ 0 > n \Rightarrow \forall f : \{0..0\} \rightarrow \{0..n\}.\neg Inj(f, \{0..0\}, \{0..n\})$$

Choose an arbitrary $n \in \mathbb{N}$ and assume $0 > n$. But this assumption is not possible, there is no natural number less than 0, and so the base case holds by contradiction.
(**Induction Step**:) For arbitrary $m \in \mathbb{N}$ we assume $P(m)$ (the induction hypothesis) and show $P(m+1)$.

$$\text{ind.hyp}: \ \forall n : \mathbb{N}.\ m > n \Rightarrow \forall f : \{0..m\} \rightarrow \{0..n\}.\ \neg Inj(f, \{0..m\}, \{0..n\})$$

We must show.

$$\forall n : \mathbb{N}.\ m+1 > n \Rightarrow \forall f : \{0..m+1\} \rightarrow \{0..n\}.\neg Inj(f, \{0..m+1\}, \{0..n\})$$

Choose an arbitrary $n \in \mathbb{N}$ and assume $m+1 > n$. Then choose an arbitrary function $f \in \{0..m+1\} \rightarrow \{0..n\}$ and show that it is not an injection. To complete this case we assume $Inj(f, \{0..m+1\}, \{0..n\})$ and derive a contradiction *e.g.* we assume:

$$(A) \quad \forall i, j : \{0..m+1\}.\ f(i) = f(j) \Rightarrow i = j$$

Now, consider the possibilities for $n \in \mathbb{N}$, either $n = 0$ or $n > 0$.
Case $n = 0$. In this case, our assumption that $f \in \{0..m+1\} \rightarrow \{0..0\}$ can be used to give a contradiction. The domain $\{0..m+1\}$ has at least 0 in it, even if $m = 0$. This means $f(0) \in \{0..0\}$. But $\{0..()0) = \emptyset$ and so $f(0) \in \emptyset$. This contradicts the corollary of the emptyset axiom Corollary 5.5.1 from Chapter 5.

Case $n > 0$. In this case, our assumption that $m+1 > n$ means that $m > n-1$ (subtracting one from n is justified because $n > 0$. Use $n-1$ for n in the induction hypothesis to get the the following:

$$m > n-1 \Rightarrow \forall f : \{0..m\} \rightarrow \{0..n-1\}.\ \neg Inj(f, \{0..m\}, \{0..n-1\})$$

Since we know $m > n-1$ we assume:

$$(B)\ \ \forall f : \{0..m\} \rightarrow \{0..n-1\}.\ \neg Inj(f, \{0..m\}, \{0..n-1\})$$

Now, consider the injective function $f \in \{0..m+1\} \rightarrow \{0..n\}$ from the hypothesis labeled (A) above.

There are two cases, $f(m) = n-1$ or $f(m) < n-1$.

Case $f(m) = n-1$. In this case, since f is an injection in $\{0..m+1\} \rightarrow \{0..n\}$, removing m from the domain also removes $n-1$ from the co-domain and so $f \downarrow \{0..m\}$ a function of type $\{0..m\} \rightarrow \{0..n-1\}$. Use this restricted f as a witness to the assumption labeled B and we assume

$$\neg Inj(f, \{0..m\}, \{0..n-1\})$$

If we can show that $Inj(f, \{0..m\}, \{0..n-1\})$ we have completed this case. But we already know that f is an injection on the larger domain $\{0..m+1\}$ so it is an injection on the smaller one.

Case $f(m) < n-1$. In this case, since f is an injection with codomain $\{0..n\}$, at most one element of the domain $\{0..m+1\}$ gets mapped by f to $n-1$ if it exists, call it k. Using f we will construct a new function (call it g) by having g behave just like f except on input k (the one such that $f(k) = n-1$) we set $g(k) = f(m)$. Since we assumed $f(m) < n-1$ we know $g(k) \in \{0..n-1\}$ and because f is an injection we know that no other element of the domain was mapped by f to $f(m)$. So, g is defined as follows:

$$g(i) = \text{ if } f(i) = n-1 \text{ then } f(m) \text{ else } f(i)$$

Use g for f in (B) and we have the assumption that

$$\neg Inj(g, \{0..m\}, \{0..n-1\})$$

To prove this case we show $Inj(g, \{0..m\}, \{0..n-1\})$. But we constructed g using the injection f to be an injection as well.

□

10.5 Complete Induction

In the justification for mathematical induction given above in Sect 10.3.1 it can be seen that, given an number n, in the process of building a justification that $P(n)$ holds, justifications for each $P(k)$, where $k < n$ are constructed along the way.

This suggests a *stronger* induction hypothesis may be possible, to not just assume that the property holds for the preceding natural number but that it holds for all proceeding natural numbers. In fact, the following induction principle can be proved using ordinary mathematical induction.

Theorem 10.17 (Principle of Complete Induction)

$$(\forall n : \mathbb{N}.\ (\forall k : \mathbb{N}.\ k < n \Rightarrow P[k]) \Rightarrow P[n]) \\ \Rightarrow \forall n : \mathbb{N}.\ P[n]$$

A sequent style proof rule[4] for complete induction is given as follows:

Proof Rule 10.3 (Complete Induction)

$$\frac{\Gamma, n \in \mathbb{N}, \forall k : \mathbb{N}, k < n \Rightarrow P[k] \vdash P[n]}{\Gamma \vdash \forall m : \mathbb{N}.\ P[m]}\ (\text{Comp}\mathbb{N}\text{Ind}) \qquad \text{where } n \text{ is fresh.}$$

10.5.1 Proof of the Principle of Complete Induction*

Indeed, we can prove theorem 10.17. To do so we introduce a lemma which illustrates an interesting advanced proof strategy - strengthening the induction hypothesis. A attempt to directly prove 10.17 by induction on n fails. The base case is provable but the induction hypothesis is not strong enough to allow a proof of the induction step. The solution is to prove a theorem which gives a stronger induction hypothesis and to use that to prove 10.17.

Definition 10.9 (Complete Predicate) A predicate of natural numbers is *complete* if the following holds:

$$\mathit{complete}(P) \stackrel{\text{def}}{=} \forall n : \mathbb{N}.(\forall m : \mathbb{N}.\, m < n \Rightarrow P[m]) \Rightarrow P[n]$$

Lemma 10.5 (Complete Induction)

$$\mathit{complete}(P) \Rightarrow \forall k : \mathbb{N}.\forall j : \mathbb{N}.j < k \Rightarrow P[j]$$

Proof: Assume P is complete, *i.e.*:

$$C: \quad \forall n : \mathbb{N}.(\forall m : \mathbb{N}.\, m < n \Rightarrow P[m]) \Rightarrow P[n]$$

We prove

$$\forall k : \mathbb{N}.\forall j : \mathbb{N}.j < k \Rightarrow P[j]$$

by induction on k.

base case: Show $\forall j : \mathbb{N}.j < 0 \Rightarrow P[j]$. This holds for arbitrary $j \in \mathbb{N}$ because $j < 0$ is in contradiction with Lemma **??** which says $\neg(j < 0)$ for all j.

[4] We have omitted the contexts Δ_1 and Δ_2 on the right sides of the sequents for readability.

induction step: For arbitrary $k \in \mathbb{N}$ assume the induction hypothesis:

$$IH: \quad \forall j : \mathbb{N}. j < k \Rightarrow P[j]$$

We must show

$$\forall j : \mathbb{N}. j < k + 1 \Rightarrow P[j]$$

Let $j \in \mathbb{N}$ be arbitrary, assume $j < k + 1$ and show $P[j]$. By Lemma **??** there are two cases, $j < k$ or $j = k$. In the case $j < k$ the induction hypothesis yields $P[j]$. In the case $j = k$, use k for n in assumption C that P is complete to get the following:

$$(\forall m : \mathbb{N}.\, m < k \Rightarrow P[m]) \Rightarrow P[k]$$

The antecedent of this last formula is an instance of the induction hypothesis IH (to see this rename the bound variable j in IH to m.) This yields $P[k]$ and since $j = k$ we get $P[j]$, as we were to show.
□

Theorem 10.17, the principle of complete induction, follows from Lemma 10.5.
Proof: To prove

$$(\forall n : \mathbb{N}.\ (\forall k : \mathbb{N}.\ k < n \Rightarrow P[k]) \Rightarrow P[n]) \qquad \Rightarrow \forall n : \mathbb{N}.\ P[n]$$

assume P is complete and show $P[n]$. Since P is complete, by Lemma 10.5 we know

$$\forall k : \mathbb{N}. \forall j : \mathbb{N}. j < k \Rightarrow P[j]$$

. Use $n + 1$ for k and n for j which yields $n < n + 1 \Rightarrow P[n]$. The antecedent is true by Lemma **??** and so $P[n]$ holds.
□

10.5.2 Applications

Complete induction is especially useful in proving properties of functions defined by recursion but which do not follow the structure of the natural numbers. The Fibonacci numbers provide an excellent example of such a function.

Definition 10.10. Fibonacci numbers

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(k+2) &= F(k+1) + F(k) \end{aligned}$$

Theorem 10.18 (Fibonacci grows slower that Exp)

$$\forall n : \mathbb{N}.\ F(n) < 2^n$$

Proof: By complete induction on n. The property is $P[n] \stackrel{\text{def}}{=} F(n) < 2^n$. We assume that $n \in \mathbb{N}$ and our induction hypothesis becomes

$$\text{Induction Hypothesis } \forall k : \mathbb{N}.\ k < n \Rightarrow F(k) < 2^k$$

We must show that $F(n) < 2^n$. We assert the following leaving the proof to the reader.

$$\forall m : \mathbb{N}.\ m = 0 \vee m = 1 \vee \exists k : \mathbb{N}.\ m = k + 2$$

Using n for m in the assertion we have three cases to consider.
case$[n = 0]$**:** Assume $n = 0$ and show $F(0) < 2^0$. By definition, $F(0) = 0$ and $2^0 = 1$ so this case holds.
case$[n = 1]$**:** Assume $n = 1$ and show $F(1) < 2^1$. By definition $F(1) = 1$ and also $2^1 = 2$ so this case holds.
case$[\exists k : \mathbb{N}.\ n = k + 2]$**:** Assume $\exists k : \mathbb{N}.\ n = k + 2$. Let $k \in \mathbb{N}$ be such that $n = k+2$. We must show $F(k+2) < 2^{k+2}$. By definition $F(k+2) = F(k+1)+F(k)$. Since we have assumed $n = k + 2$, we know that $k + 1 < n$ and $k < n$. Using $k + 1$ and k in induction hypothesis we get the following facts $F(k + 1) < 2^{k+1}$ and $F(k) < 2^k$. We use two instances of Thm. 10.13 to get the following:

$$F(k + 1) + F(k) < 2^{k+1} + F(k) < 2^{k+1} + 2^k$$

Note that by Thm. 10.14 we know $2^k < 2^{k+1}$ so, by definition of exponentiation, $2^k < 2 \cdot 2^k$. This justifies the following:

$$2^{k+1} + 2^k = 2 \cdot 2^k + 2^k < 2 \cdot 2^k + 2 \cdot 2^k = 2 \cdot 2^k = 2^{k+1}$$

This string of inequalities and equalities shows that $F(k+2) < 2^{k+2}$ and so this case is complete.

By these three cases we have shown that for all $n \in \mathbb{N}$ the theorem holds.
□

Definition 10.11 (Divisibility) For $m, n \in \mathbb{N}$ we say $m|n$ (read: "m divides n") if there is a $k \in \mathbb{N}$ such that $n = m \cdot k$.

$$m|n \stackrel{\text{def}}{=} \exists k : \mathbb{N}. n = k \cdot m$$

Definition 10.12 (Prime Numbers)

$$Prime(n) \stackrel{\text{def}}{=} n > 1 \wedge \forall k : \{2 \cdots n\}.\ \neg(k|n)$$

Corollary 10.3 (2 is the least prime)

$$\forall n : \mathbb{N}.\, Prime(n) \Rightarrow 2 \leq n$$

Proof: Choose an arbitrary $n \in \mathbb{N}$ and assume $Prime(n)$. Then $n > 1$ and $\forall k : \{2 \cdots n\}.\ \neg(k|n)$. Since $n > 1$ we know $n = 2$ or $n > 2$. If $n = 2$ then the theorem holds because $2 \leq 2$. If $n > 2$, then $2 < n$ and so $2 \leq n$.

□

Thus the set of prime numbers is $\{2, 3, 5, 7, 11, 13, 17, \cdots\}$.

Lemma 10.6 (Not Prime Composite)

$$\forall n.\mathbb{N}.n > 1 \Rightarrow \neg Prime(n) \Leftrightarrow \exists i, j : \{2 \cdots n\}.\ n = i \cdot j$$

In Chapter 11 we completely develop the list data-type and formally describe a function which computes the product of a list of numbers. We present it here, though the intuition behind the function is enough to understand the Theorem 10.19.

Informally,

$$\Pi\,[k_1, k_2, \cdots, k_n] = k_1 \cdot k_2 \cdots k_n$$

If the list is empty ($[\,]$) we stipulate $\Pi[\,] = 1$.

Definition 10.13 (Product of a List) We formally define the product of a list of numbers as follows:

$$\begin{array}{l} \Pi\,[\,] = 1 \\ \Pi\,(k :: rest) = k \cdot \Pi\, rest \end{array}$$

A factorization is a kind of decomposition of a number.

Definition 10.14 (Factorization) We say a list f is a *factorization* of n if the product $\Pi f = n$. A list f is a *prime factorization* of n if f is a factorization of n and every element of f is a prime number.

Example 10.3. Note that $4 = 2 \cdot 2$ so a factorization of 4 is given by the list $[2, 2]$. The factorization of a prime is a singleton list containing the prime. For example, the factorization of 7 is the list $[7]$. A non-prime factorization of of 20 is $[2, 10]$. The prime factorization of 20 is $[2, 2, 5]$. When using lists to represent factorizations the order of the elements in the factorization is not considered. Thus, $[2, 2, 5]$ is the same factorization (up to the order of elements) as $[2, 5, 2]$ and as $[5, 2, 2]$. If we say a factorization is unique we mean modulo the order of the elements.

Exercise 10.5. Note that sets of natural numbers are *not* a suitable representation of factorizations. Why not?

The fundamental theorem of arithmetic says that each natural number n can be uniquely factorized by primes. It says something rather deep about the structure of the natural numbers; in a mathematically precise way it says that the primes are the building-blocks of the natural numbers.

Theorem 10.19 (Fundamental Theorem of Arithmetic)

$$\forall n : \mathbb{N}.\, n > 0 \Rightarrow \exists f : \mathbb{N}List.\, (\forall k : \mathbb{N}.\, k \in f \Rightarrow Prime(k)) \wedge \Pi f = n$$

Proof: We prove the theorem by complete induction on n. The property of n to be proved is:

$$P[n] = n > 0 \Rightarrow \exists f : \mathbb{N}List.\, (\forall k : \mathbb{N}.\, k \in f \Rightarrow Prime(k)) \wedge \Pi f = n$$

Let $n \in \mathbb{N}$ be arbitrary and the induction hypothesis is of the following form:

$$\forall k : \mathbb{N}.k < n \Rightarrow P[k]$$

We must show $P[n]$, *i.e.* that

$$n > 0 \Rightarrow \exists f : \mathbb{N}List.\, (\forall k : \mathbb{N}.\, k \in f \Rightarrow Prime(k)) \wedge \Pi f = n$$

Assume $n > 0$ and show that there exists a factorization f of n. We consider cases, either $n = 1$ or $n > 1$. If $n = 1$ then let $f = [\,]$. We must show the following.

$$(\forall k : \mathbb{N}.\, k \in [\,] \Rightarrow Prime(k)) \wedge \Pi[\,] = n$$

The left conjunct holds vacuously since, assuming $k \in [\,]$ for arbitrary k is a contradiction *i.e.* there are no elements in $[\,]$. By the definition of the product of a list $\Pi[\,] = 1$. In the case $n > 1$ we consider whether n is prime itself or not, *i.e.* by cases, either $Prime(n)$ or $\neg Prime(N)$. If $Prime(n)$ then the list $f = [n]$ is a prime factorization. If $\neg Prime(n)$ then, by Lemma **??** we know there exists $j, k \in \{2..n\}$ such that $n = j \cdot k$.
□

Chapter 11

Lists

11.1 Lists

John McCarthy

John McCarthy (1927-),an American Computer Scientist and one of the fathers of Artificial Intelligence. John McCarty invented the LISP programming language after reading Alonzo Church's monograph on the lambda calculus. LISP stands for *List Processing* and lists are well supported as a the fundamental datatype in the language.

Lists may well be the most ubiquitous datatype in computer science. Functional programming languages like LISP, Scheme, ML and Haskell support lists in significant ways that make them a go-to data-structure. The can be used to model many collection classes (multisets or bags come to mind) as well as relations (as list of pairs) and finite functions.

We define lists here that may contain elements from some set T. These are the so-called *monomorphic* lists; they can only contain elements of type T. There are two constructors to create a list. Nil (written as "[]") is a constant symbol denoting 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 "::" are both written by sequences of two symbols, we consider them to be atomic symbols for the purposes of the syntax.

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

Definition 11.1 (T List)

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

where

T: is a set,
$[\,]$: is a constant symbol denoting the *empty list*, which is called "nil",
a: 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*.

Example 11.1. 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 $List_A$ *lists*. If $A \neq$ then the set of lists in class $List_A$ is infinite, but also, like the representation of natural numbers, the representation of each individual list is finite[1] Finiteness follows from the fact that lists are constructed by consing some value from the set A onto a previously constructed $List_A$. 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 abstract syntax tree: shown in Fig 11.1.

[1] The infinite analog of lists are called streams. They have an interesting theory of their own where induction is replaced by a principle of co-induction. The Haskell programming language supports an elegant style of programming with streams by implementing an evaluation mechanism that is *lazy*; computations are only performed when the result is needed.

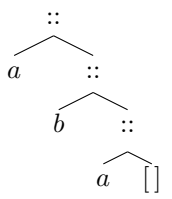


Figure 11.1: Syntax tree for the list $[a, b, a]$ constructed as $a::(b::(a::[\,]))$

11.2 Definition by recursion

In the same way that in Chapter 10 we defined functions by recursion on the structure of an argument of type $\mathbb{N}$, we can define functions of type $List_A \to B$ by recursion on the structure of list arguments.

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

Definition 11.2 (List append)

$$\begin{aligned} append([\,], M) &\stackrel{\text{def}}{=} M \\ append((a::L), M) &\stackrel{\text{def}}{=} a::(append(L, M)) \end{aligned}$$

The first equation of the definition says: if the first argument is the empty 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. Note that since there are only two ways to construct a list, this definition covers all possible ways the first argument could be constructed.

Example 11.2. We give some example computations with the definition of *append*.

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

Using the more compact notation for lists, we have shown $append([a, b], [c]) = [a, b, c]$. Using the pretty notation for lists we can rewrite the derivation as

follows:

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

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.

Append is such a common operation on lists that the ML and Haskell programming languages provide convenient infix notations for the list append operator. In Haskell, the symbol is "++", in the ML family of languages it is "@". We will use "$+\!\!+$" here. Using the infix notation the definition appears as follows:

Definition 11.3 (List append (infix))

$$\begin{aligned} [\,] +\!\!+ M &\stackrel{\text{def}}{=} M \\ (a::L) +\!\!+ M &\stackrel{\text{def}}{=} a::(L +\!\!+ M) \end{aligned}$$

Example 11.3. Here is an example of a computation using the infix operator for append and the more compact notation for lists.

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

We would like to know that append really is a function of type $(List_A \times List_A) \to List_A$. In Chapter 10 we presented a theorem justifying recursive definitions of a particular form (Thm. 11.1). That theorem and Corollary 11.1) guaranteed that definitions that followed a syntactic pattern were guaranteed to be functions. Similar results hold for lists and indeed there are similar theorems justifying definition by recursion for any inductively defined type.

Theorem 11.1 (Definition by recursion for list functions) Given sets A and B and an element $b \in B$ and a function $g \in (A \times B) \to B$, definitions having the following form:

$$\begin{aligned} &f([\,]) = b \\ &f(x :: xs) = g(x, f(xs)) \end{aligned}$$

result in well-defined functions, $f \in List_A \to B$.

The corollary for functions of two arguments is given as:

Corollary 11.1 (Definition by recursion (for binary functions)) Given sets A, B and C and a function $g \in (A \times C) \to C$, and a function $h \in A \to C$, and an element $b \in B$, definitions of the following form:

$$f([\,], b) = h(b)$$
$$f((x::xs), b) = g(x, f(xs, b))$$

result in well-defined functions, $f \in (List_A \times B) \to C$.

Theorem 11.2 ($append \in (List_A \times List_A) \to List_A$) Recall the definition of append

$$append([\,], M) \stackrel{\text{def}}{=} M$$
$$append((a::L), M) \stackrel{\text{def}}{=} a::(append(L, M))$$

Proof: We apply Corollary 11.1. Let A be an arbitrary set and let $B = List_A$ and $C = List_A$. Let h be the identity function on $List_A \to List_A$ and $g(x, m) = x::m$. Then $g \in (A \times List_A) \to List_A$. This fits the pattern and shows that append is a function of type $(List_A \times List_A) \to List_A$.
□

In general, there is no need to apply the theorem or corollary, if a definition is given for an operator where the definition is presented by cases on a list argument and where the [ ] case is not recursive. In that case, the operator can be shown to be a function. The idea is that at each recursive call, the length of the list argument is getting shorter and eventually will become [ ], at that point the base case is invoked and there is no more recursion so it terminates.

Here are a few more functions defined by recursion on lists.

Definition 11.4 (List length)

$$length([\,]) = 0$$
$$length(x::xs) = 1 + length(xs)$$

We will write $|L|$ instead of $length(L)$ in the following.

Definition 11.5 (List member)

$$mem(y, [\,]) = false$$
$$mem(y, x::xs) = y = x \vee mem(y, xs)$$

We will write $y \in M$ for $mem(y, M)$.

Definition 11.6 (List reverse)

$$rev([\,]) = [\,]$$
$$rev(x::xs) = rev(xs) +\!\!+ [x]$$

Exercise 11.1. Show that length, member and reverse are all functions by applying Thm. 11.1 or Corollary 11.1.

11.3 List Induction

The structural induction principle for lists is given as follows:

Definition 11.7 (List Induction) For a set A and a property P of $List_A$ we have the following axiom.

$$(P[[\,]] \wedge \forall x : A.\ \forall xs : List_A.\ P[xs] \Rightarrow P[x::xs]) \Rightarrow \forall ys : List_A.\ P[ys]$$

The corresponding proof rule is given as follows:

Proof Rule 11.1 (List Induction)

$$\frac{\Gamma \vdash P[[\,]] \qquad \Gamma, x \in A, , xs \in List_A, P[xs] \vdash P[x::xs]}{\Gamma \vdash \forall ys : List_A.\ P[ys]}\ (List_A\text{Ind}) \quad x, xs \text{ fresh.}$$

11.3.1 Some proofs by list induction

Def. 11.3 of the append function shows directly that $[\,]$ is a left identity for $+\!\!+$, is it a right identity as well? The following theorem establishes this fact.

Theorem 11.3 (Nil right identity for $+\!\!+$)

$$\forall ys : List_A.\ ys +\!\!+ [\,] = ys$$

Proof: By list induction on ys. The property P of ys is given as:

$$P[ys] \stackrel{\text{def}}{=} ys +\!\!+ [\,] = ys$$

Base Case: Show $P[[\,]]$, *i.e.* that $[\,] +\!\!+ [\,] = [\,]$. This follows immediately from the definition of append.

Induction Step: Assume $P[xs]$ (the induction hypothesis) and show $P[x::xs]$ for arbitrary $x \in A$ and arbitrary $xs \in List_A$. The induction hypothesis is:

$$xs +\!\!+ [\,] = xs$$

We must show $(x::xs) +\!\!+ [\,] = (x::xs)$. Starting with the left side of the equality we get the following:

$$(x::xs) +\!\!+ [\,] \stackrel{\langle\!\langle def.\,of\ +\!\!+ \rangle\!\rangle}{=} x::(xs +\!\!+ [\,]) \stackrel{\langle\!\langle ind.hyp. \rangle\!\rangle}{=} x::xs$$

So the induction step holds and the proof is complete.
□

Together, Def. 11.3 and Thm. 11.3 establish that $[\,]$ is a left and right identity with respect to append. Thus, with respect to $+\!\!+$, $[\,]$ behaves like zero does for ordinary addition.

The next theorem shows that append is associative.

Theorem 11.4 ($+\!\!+$ is associative)

$$\forall ys, zs, xs : List_A.\ xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

Proof: Choose arbitrary $ys, zs \in List_A$. We continue by list induction on xs. The property P of xs is given as:

$$P[xs] \stackrel{\text{def}}{=} xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

Base Case: Show $P[[\,]]$, *i.e.* that

$$[\,] +\!\!+ (ys +\!\!+ zs) = ([\,] +\!\!+ ys) +\!\!+ zs$$

On the left side:

$$[\,] +\!\!+ (ys +\!\!+ zs) \stackrel{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} (ys +\!\!+ zs)$$

On the right side:

$$([\,] +\!\!+ ys) +\!\!+ zs \stackrel{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} (ys +\!\!+ zs)$$

So the base case holds.

Induction Step: For arbitrary $x \in A$ and $xs \in List_A$ we assume $P[xs]$ (the induction hypothesis) and show $P[x::xs]$.

$$xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs \qquad \text{Ind.Hyp.}$$

We must show

$$(x::xs) +\!\!+ (ys +\!\!+ zs) = ((x::xs) +\!\!+ ys) +\!\!+ zs$$

Starting on the left side:

$$(x::xs) +\!\!+ (ys +\!\!+ zs) \stackrel{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} x::(xs +\!\!+ (ys +\!\!+ zs)) \stackrel{\langle\!\langle Ind.Hyp. \rangle\!\rangle}{=} x::((xs +\!\!+ ys) +\!\!+ zs)$$

On the right side:

$$((x::xs) +\!\!+ ys) +\!\!+ zs \stackrel{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} (x::(xs +\!\!+ ys)) +\!\!+ zs \stackrel{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} x::((xs +\!\!+ ys) +\!\!+ zs)$$

So the left and right sides are equal and this completes the proof.

Here's an interesting theorem about reverse. We've seen this pattern before in Chapter 6 in Theorem 6.2 where we proved that the inverse of a composition is the composition of inverses.

Theorem 11.5 (Reverse of append)

$$\forall ys, xs : List_A.\ rev(xs +\!\!+ ys) = rev(ys) +\!\!+ rev(xs)$$

Proof: Choose an arbitrary $ys \in List_A$ and proceed by list induction on xs. Choose arbitrary $xs \in List_A$. The property P of xs is given as:

$$P[xs] \stackrel{\text{def}}{=} rev(xs +\!\!+ ys) = rev(ys) +\!\!+ rev(xs)$$

Base Case: Show $P[[\,]]$ *i.e.* that

$$rev([\,] +\!\!+ ys) = rev(ys) +\!\!+ rev([\,])$$

We start with the left side and reason as follows:

$$rev([\,] +\!\!+ ys) \stackrel{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} rev(ys)$$

On the right side,

$$rev(ys) +\!\!+ rev([\,]) \stackrel{\langle\!\langle def.of rev \rangle\!\rangle}{=} rev(ys) +\!\!+ [\,] \stackrel{\langle\!\langle Thm\ 11.3 \rangle\!\rangle}{=} rev(ys)$$

So the left and right sides are equal and the base case holds.

Induction Step: For arbitrary $xs \in List_A$ assume $P[xs]$ and show $P[x::xs]$ for some arbitrary $x \in A$.

$$rev(xs +\!\!+ ys) = rev(ys) +\!\!+ rev(xs) \qquad \text{Ind.Hyp.}$$

We must show:

$$rev((x::xs) +\!\!+ ys) = rev(ys) +\!\!+ rev((x::xs))$$

We start with the left side.

$$\begin{aligned} & rev((x::xs) +\!\!+ ys) \\ & \stackrel{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} rev(x::(xs +\!\!+ ys)) \\ & \stackrel{\langle\!\langle def.of rev \rangle\!\rangle}{=} rev(xs +\!\!+ ys) +\!\!+ [x] \\ & \stackrel{\langle\!\langle Ind.Hyp. \rangle\!\rangle}{=} (rev(ys) +\!\!+ rev(xs)) +\!\!+ [x] \end{aligned}$$

On the right side:

$$\begin{aligned} & rev(ys) +\!\!+ rev((x::xs)) \\ & \stackrel{\langle\!\langle def.of rev \rangle\!\rangle}{=} rev(ys) +\!\!+ (rev(xs) +\!\!+ [x]) \\ & \stackrel{\langle\!\langle Thm\ 11.4 \rangle\!\rangle}{=} (rev(ys) +\!\!+ rev(xs)) +\!\!+ [x] \end{aligned}$$

□

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Stuart Allen. Discrete math lessons . `http://www.cs.cornell.edu/Info/People/sfa/Nuprl/eduprl/Xcounting%underscoreintro.html`.

[3] Jonathan Barnes. *Logic and the Imperial Stoa*, volume LXXV of *Philosophia Antiqua*. Brill, Leiden · New York · Koln, 1997.

[4] Garrett Birkhoff and Saunders Mac Lane. *A Survey of Modern Algebra*. Macmillan, New York, 2nd edition, 1965.

[5] Noam Chomsky. *Syntactic Structures*. Number 4 in Janua Linguarum, Minor. Mouton, The Hague, 1957.

[6] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951.

[7] Richard Dedekind. *Was sind and was sollen die Zahlen?* 1888. English translation in [8].

[8] Richard Dedekind. *Essays on the Theory of Numbers*. Dover, 1963.

[9] Kees Doets and Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*, volume 4 of *Texts in Computing*. Kings College Press, London, 2004.

[10] Paul Edwards, editor. *The Encyclopedia of Philosophy*, Eight volumes published in four, unabridged, New York · London, 1972. Collier Macmillan & The Free Press.

[11] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[12] E. Engeler. *Formal Languages: Automata and Structures.* Markham, Chicago, 1968.

[13] *Discourses of Epictetus*. D. Appleton and Co., New Youk, 1904. Translated by George Long.

[14] Anita Burdman Feferman and Solomon Feferman. *Alfred Tarski: Life and Logic*. Cambridge University Press, 2004.

[15] Abraham A. Frankel. Georg Cantor. In Edwards [10], pages 20–22.

[16] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[17] Galileo Galilei. *Two New Sciences*. University of Wisconsin Press, 1974. Translated by Stillman Drake.

[18] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[19] Nelson Goodman. *The Structure of Appearance, Third ed.*, volume 107 of *Synthese Library*. D. Reidel, Dordrecht, 1977.

[20] Nelson Goodman and W. V. Quine. Steps toward a constructive nominalism. *Jounral of Symbolic Logic*, 12:105 – 122, 1947.

[21] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, New York, 1993.

[22] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. MIT Press, 1992.

[23] 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.

[24] 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.

[25] Paul R. Halmos. *Boolean Algebra*. Nan Nostrand Rienholt, New York, 1968.

[26] Paul R. Halmos. *Naive Sert Theory*. Springer Verlag, New York · Heidelberg · Berlin, 1974.

[27] Herodotus. *The History*. University of Chicago, 1987. Translated by David Green.

[28] D. Hilbert and W. Ackermann. *Mathematical Logic*. Chelsea Publishing, New York, 1950.

[29] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, Massachusetts, 1969.

[30] Stephen C. Kleene. *Introduction to Metamathematics*. van Nostrand, Princeton, 1952.

[31] Saunders Mac Lane. *Mathematics: Form and Function*. Springer Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.

[32] F. William Lawvere and Stephen Schanuel. *Conceptual Mathematics: A First Introiduction to Categories*. Cambridge University Press, 1997.

[33] Xavier Leroy. *The Objective Caml system release 1.07*. INRIA, France, May 1997.

[34] Nicholas Lobachevski. *Geometrical Researches on The Theory of Parallels*. Open Court Publishing Co., La Salle, Illinois, 1914. Originally published in German, Berlin in 1840, translated by George Bruce Halsted.

[35] Zohar Manna. *Theory of Computation*. Addison-Wesley, 1978.

[36] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming: Deductive Reasoning*. Addison-Wesley, 1985. Published in 2 Volumes.

[37] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1991.

[38] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[39] L. C. Paulson. *Standard ML for the Working Programmer*. Cambridge University Press, 1991.

[40] Giuseppe Peano. *Arithmetices principia, nova methodo exposita*, 1899. English translation in [51], pp. 83–97.

[41] Benjamin Pierce. *Basic Category Theory for COmputer Scientists*. MIT Press, 1991.

[42] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Dept., Denmark, 1981.

[43] David J. Pym and Eike Ritter. *Reductive Logic and Proof-search: Proof Theory, Semantics and Contol*, volume 45 of *Oxford Logic Guides*. Clarendon Press, Oxford, 2004.

[44] W.V.O. Quine. *Methods of Logic*. Holt, Rinehart and Winston, 1950.

[45] Arto Salomaa. *Formal Languages*. ACM Monograph Series. Academic Press, 1973.

[46] S. A. Schmidt. *Denotational Semantics*. W. C. Brown, Dubuque, Iowa, 1986.

[47] 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.

[48] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[49] A. Tarski. *Logic, Semantics, Metamathematics, Papers from 1923 to 1938.* Clarendon Press, Oxford, 1956.

[50] R. D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, London, 1991.

[51] Jan van Heijenoort, editor. *From Frege to Gödel: A sourcebook in mathematical logic, 1879 – 1931.* Harvard University Press, 1967.

[52] Wikipedia. Galileo's paradox — wikipedia, the free encyclopedia, 2005.

[53] G. Winskel. *Formal Semantics of Programming Languages.* MIT Press, Cambridge, 1993.

[54] Glen Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing. MIT Press, 1993.

[55] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus.* Routledge and Kegan Paul Ltd., London, 1955.

Index