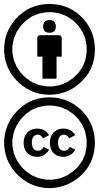
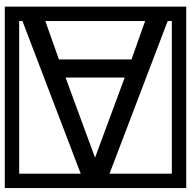
THE OPEN LOGIC TEXT

Complete Build

Open Logic Project

Revision: 9d4975e
2015-06-26





About the Open Logic Project

The *Open Logic Text* is an open-source, collaborative textbook of formal metalogic and formal methods, starting at an intermediate level (i.e., after an introductory formal logic course). Though aimed at a non-mathematical audience (in particular, students of philosophy and computer science), it is rigorous.

The *Open Logic Text* is a collaborative project and is under active development. Coverage of some topics currently included may not yet be complete, and many sections still require substantial revision. We plan to expand the text to cover more topics in the future. We also plan to add features to the text, such as a glossary, a list of further reading, historical notes, pictures, better explanations, sections explaining the relevance of results to philosophy, computer science, and mathematics, and more problems and examples. If you find an error, or have a suggestion, please let the project team know.

The project operates in the spirit of open source. Not only is the text freely available, we provide the LaTeX source under the Creative Commons Attribution license, which gives anyone the right to download, use, modify, rearrange, convert, and re-distribute our work, as long as they give appropriate credit.

Please see the Open Logic Project website at openlogicproject.org for additional information.

Contents

Part I

First-order Logic

Chapter 1

Syntax and Semantics

1.1 First-Order Languages

Expressions of first-order logic are built up from a basic vocabulary containing *variables*, *constant symbols*, *predicate symbols* and sometimes *function symbols*. From them, together with logical connectives, quantifiers, and punctuation symbols such as parentheses and commas, *terms* and *formulas* are formed.

Informally, predicate symbols are names for properties and relations, constant symbols are names for individual objects, and function symbols are names for mappings. These, except for the identity predicate $=$, are the *non-logical symbols* and together make up a language. Any first-order language $\mathcal{L}$ is determined by its non-logical symbols. In the most general case, $\mathcal{L}$ contains infinitely many symbols of each kind.

In the general case, we make use of the following symbols in first-order logic:

1. Logical symbols
    a) Logical connectives: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (conditional), $\leftrightarrow$ (biconditional), $\forall$ (universal quantifier), $\exists$ (existential quantifier).
    b) The propositional constant for falsity $\bot$.
    c) The propositional constant for truth $\top$.
    d) The two-place identity predicate $=$.
    e) A denumerable set of variables: v_0, v_1, v_2, ...
2. Non-logical symbols, making up the *standard language* of first-order logic
    a) A denumerable set of n-place predicate symbols for each $n > 0$: A^n_0, A^n_1, A^n_2, ...
    b) A denumerable set of constant symbols: c_0, c_1, c_2, ...

c) A denumerable set of n-place function symbols for each $n > 0$: f^n_0, f^n_1, f^n_2, ...

3. Punctuation marks: (,), and the comma.

Most of our definitions and results will be formulated for the full standard language of first-order logic. However, depending on the application, we may also restrict the language to only a few predicate symbols, constant symbols, and function symbols.

Example 1.1. The language $\mathcal{L}_A$ of arithmetic contains a single two-place predicate symbol $<$, a single constant symbol 0, one one-place function symbol $'$, and two two-place function symbols $+$ and $\times$.

Example 1.2. The language of set theory $\mathcal{L}_Z$ contains only the single two-place predicate symbol $\in$.

Example 1.3. The language of orders $\mathcal{L}_\leq$ contains only the two-place predicate symbol $\leq$.

Again, these are conventions: officially, these are just aliases, e.g., $<$, $\in$, and $\leq$ are aliases for A^2_0, 0 for c_0, $'$ for f^1_0, $+$ for f^2_0, $\times$ for f^2_1.

You may be familiar with different terminology and symbols than the ones we use above. Logic texts (and teachers) commonly use either $\sim$, $\neg$, and ! for "negation", $\wedge$, $\cdot$, and & for "conjunction". Commonly used symbols for the "conditional" or "implication" are $\rightarrow$, $\Rightarrow$, and $\supset$. Symbols for "biconditional," "bi-implication," or "(material) equivalence" are $\leftrightarrow$, $\Leftrightarrow$, and $\equiv$. The $\bot$ symbol is variously called "falsity," "falsum,", "absurdity,", or "bottom." The $\top$ symbol is variously called "truth," "verum,", or "top."

It is conventional to use lower case letters (e.g., a, b, c) from the beginning of the Latin alphabet for constant symbols (sometimes called names), and lower case letters from the end (e.g., x, y, z) for variables. Quantifiers combine with variables, e.g., x; notational variations include $\forall x$, $(\forall x)$, (x), Πx, $\bigwedge_x$ for the universal quantifier and $\exists x$, $(\exists x)$, (Ex), Σx, $\bigvee_x$ for the existential quantifier.

We might treat all the propositional operators and both quantifiers as primitive symbols of the language. We might instead choose a smaller stock of primitive symbols and treat the other logical operators as defined. "Truth functionally complete" sets of Boolean operators include $\{\neg, \vee\}$, $\{\neg, \wedge\}$, and $\{\neg, \rightarrow\}$—these can be combined with either quantifier for an expressively complete first-order language.

You may be familiar with two other logical operators: the Sheffer stroke $|$ (named after Henry Sheffer), and Peirce's arrow $\downarrow$, also known as Quine's dagger. When given their usual readings of "nand" and "nor" (respectively), these operators are truth functionally complete by themselves.

1.2 Terms and Formulas

Once a first-order language $\mathcal{L}$ is given, we can define expressions built up from the basic vocabulary of $\mathcal{L}$. These include in particular *terms* and *formulas*.

Definition 1.4 (Terms)**.** The set of *terms* $\mathrm{Trm}(\mathcal{L})$ of $\mathcal{L}$ is defined inductively by:

1. Every variable is a term.
2. Every constant symbol of $\mathcal{L}$ is a term.
3. If f is an n-place function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.
4. Nothing else is a term.

A term containing no variables is a *closed term*.

The constant symbols appear in our specification of the language and the terms as a separate category of symbols, but they could instead have been included as zero-place function symbols. We could then do without the second clause in the definition of terms. We just have to understand $f(t_1, \ldots, t_n)$ as just f by itself if $n = 0$.

Definition 1.5 (Formula)**.** The set of *formulas* $\mathrm{Frm}(\mathcal{L})$ of the language $\mathcal{L}$ is defined inductively as follows:

1. $\bot$ is an atomic formula.
2. $\top$ is an atomic formula.
3. If R is an n-place predicate symbol of $\mathcal{L}$ and $t_1, \ldots, t_n$ are terms of $\mathcal{L}$, then $R(t_1, \ldots, t_n)$ is an atomic formula.
4. If t_1 and t_2 are terms of $\mathcal{L}$, then $=(t_1, t_2)$ is an atomic formula.
5. If φ is a formula, then $\neg\varphi$ is formula.
6. If φ and ψ are formulas, then $(\varphi \wedge \psi)$ is a formula.
7. If φ and ψ are formulas, then $(\varphi \vee \psi)$ is a formula.
8. If φ and ψ are formulas, then $(\varphi \to \psi)$ is a formula.
9. If φ and ψ are formulas, then $(\varphi \leftrightarrow \psi)$ is a formula.
10. If φ is a formula and x is a variable, then $\forall x\, \varphi$ is a formula.
11. If φ is a formula and x is a variable, then $\exists x\, \varphi$ is a formula.

12. Nothing else is a formula.

The definitions of the set of terms and that of formulas are *inductive definitions*. Essentially, we construct the set of formulas in infinitely many stages. In the initial stage, we pronounce all atomic formulas to be formulas; this corresponds to the first few cases of the definition, i.e., the cases for $\top$, $\bot$, $R(t_1, \ldots, t_n)$ and $=(t_1, t_2)$. "Atomic formula" thus means any formula of this form.

The other cases of the definition give rules for constructing new formulas out of formulas already constructed. At the second stage, we can use them to construct formulas out of atomic formulas. At the third stage, we construct new formulas from the atomic formulas and those obtained in the second stage, and so on. A formula is anything that is eventually constructed at such a stage, and nothing else.

By convention, we write $=$ between its arguments and leave out the parentheses: $t_1 = t_2$ is an abbreviation for $=(t_1, t_2)$. Moreover, $\neg=(t_1, t_2)$ is abbreviated as $t_1 \neq t_2$. When writing a formula $(\psi * \chi)$ constructed from ψ, χ using a two-place connective $*$, we will often leave out the outermost pair of parentheses and write simply $\psi * \chi$.

Some logic texts require that the variable x must occur in φ in order for $\exists x\, \varphi$ and $\forall x\, \varphi$ to count as formulas. Nothing bad happens if you don't require this, and it makes things easier.

If we work in a language for a specific application, we will often write two-place predicate symbols and function symbols between the respective terms, e.g., $t_1 < t_2$ and $(t_1 + t_2)$ in the language of arithmetic and $t_1 \in t_2$ in the language of set theory. The successor function in the language of arithmetic is even written conventionally *after* its argument: t'. Officially, however, these are just conventional abbreviations for $A^2_0(t_1, t_2)$, $f^2_0(t_1, t_2)$, $A^2_0(t_1, t_2)$ and $f^1_0(t)$, respectively.

Definition 1.6. The symbol $\equiv$ expresses syntactic identity between strings of symbols, i.e., $\varphi \equiv \psi$ iff φ and ψ are strings of symbols of the same length and which contain the same symbol in each place.

The $\equiv$ symbol may be flanked by strings obtained by concatenation, e.g., $\varphi \equiv (\psi \vee \chi)$ means: the string of symbols φ is the same string as the one obtained by concatenating an opening parenthesis, the string ψ, the $\vee$ symbol, the string χ, and a closing parenthesis, in this order. If this is the case, then we know that the first symbol of φ is an opening parenthesis, φ contains ψ as a substring (starting at the second symbol), that substring is followed by $\vee$, etc.

1.3 Unique Readability

The way we defined formulas guarantees that every formula has a *unique reading*, i.e., there is essentially only one way of constructing it according to our

formation rules for formulas and only one way of "interpreting" it. If this were not so, we would have ambiguous formulas, i.e., formulas that have more than one reading or intepretation—and that is clearly something we want to avoid. But more importantly, without this property, most of the definitions and proofs we are going to give will not go through.

Perhaps the best way to make this clear is to see what would happen if we had given bad rules for forming formulas that would not guarantee unique readability. For instance, we could have forgotten the parentheses in the formation rules for connectives, e.g., we might have allowed this:

> If φ and ψ are formulas, then so is $\varphi \to \psi$.

Starting from an atomic formula θ, this would allow us to form $\theta \to \theta$, and from this, together with θ, we would get $\theta \to \theta \to \theta$. But there are two ways to do this: one where we take θ to be φ and $\theta \to \theta$ to be ψ, and the other where φ is $\theta \to \theta$ and ψ is θ. Correspondingly, there are two ways to "read" the formula $\theta \to \theta \to \theta$. It is of the form $\psi \to \chi$ where ψ is θ and χ is $\theta \to \theta$, but *it is also* of the form $\psi \to \chi$ with ψ being $\theta \to \theta$ and χ being θ.

If this happens, our definitions will not always work. For instance, when we define the main operator of a formula, we say: in a formula of the form $\psi \to \chi$, the main operator is the indicated occurrence of $\to$. But if we can match the formula $\theta \to \theta \to \theta$ with $\psi \to \chi$ in the two different ways mentioned above, then in one case we get the first occurrence of $\to$ as the main operator, and in the second case the second occurrence. But we intend the main operator to be a *function* of the formula, i.e., every formula must have exactly one main operator occurrence.

Lemma 1.7. *The number of left and right parentheses in a formula φ are equal.*

Proof. We prove this by induction on the way φ is constructed. This requires two things: (a) We have to prove first that all atomic formulas have the property in question (the induction basis). (b) Then we have to prove that when we construct new formulas out of given formulas, the new formulas have the property provided the old ones do.

Let $l(\varphi)$ be the number of left parentheses, and $r(\varphi)$ the number of right parentheses in φ, and $l(t)$ and $r(t)$ similarly the number of left and right parentheses in a term t. We leave the proof that for any term t, $l(t) = r(t)$ as an exercise.

1. $\varphi \equiv \bot$: φ has 0 left and 0 right parentheses.
2. $\varphi \equiv \top$: φ has 0 left and 0 right parentheses.
3. $\varphi \equiv R(t_1, \ldots, t_n)$: $l(\varphi) = 1 + l(t_1) + \cdots + l(t_n) = 1 + r(t_1) + \cdots + r(t_n) = r(\varphi)$. Here we make use of the fact, left as an exercise, that $l(t) = r(t)$ for any term t.

4. $\varphi \equiv t_1 = t_2$: $l(\varphi) = l(t_1) + l(t_2) = r(t_1) + r(t_2) = r(\varphi)$.

5. $\varphi \equiv \neg\psi$: By induction hypothesis, $l(\psi) = r(\psi)$. Thus $l(\varphi) = l(\psi) = r(\psi) = r(\varphi)$.

6. $\varphi \equiv (\psi * \chi)$: By induction hypothesis, $l(\psi) = r(\psi)$ and $l(\chi) = r(\chi)$. Thus $l(\varphi) = 1 + l(\psi) + l(\chi) = 1 + r(\psi) + r(\chi) = r(\varphi)$.

7. $\varphi \equiv \forall x\, \psi$: By induction hypothesis, $l(\psi) = r(\psi)$. Thus, $l(\varphi) = l(\psi) = r(\psi) = r(\varphi)$.

8. $\varphi \equiv \exists x\, \psi$: Similarly.

□

Definition 1.8. A string of symbols ψ is a proper prefix of a string of symbols φ if concatenating ψ and a non-empty string of symbols yields φ.

Lemma 1.9. *If φ is a formula, and ψ is a proper prefix of φ, then ψ is not a formula.*

Proof. Exercise. □

Proposition 1.10. *If φ is an atomic formula, then it satisfes one, and only one of the following conditions.*

1. $\varphi \equiv \bot$.

2. $\varphi \equiv R(t_1, \ldots, t_n)$ *where R is an n-place predicate symbol, $t_1, \ldots, t_n$ are terms, and each of R, $t_1, \ldots, t_n$ is uniquely determined.*

3. $\varphi \equiv t_1 = t_2$ *where t_1 and t_2 are uniquely determined terms.*

Proof. Exercise. □

Proposition 1.11 (Unique Readability)**.** *Every formula satisfies one, and only one of the following conditions.*

1. *φ is atomic.*

2. *φ is of the form $\neg\psi$.*

3. *φ is of the form $(\psi \wedge \chi)$.*

4. *φ is of the form $(\psi \vee \chi)$.*

5. *φ is of the form $(\psi \to \chi)$.*

6. *φ is of the form $(\psi \leftrightarrow \chi)$.*

7. *φ is of the form $\forall x\, \psi$.*

8. *φ is of the form $\exists x\, \psi$.*

Moreover, in each case ψ, or ψ and χ, are uniquely determined. This means that, e.g., there are no different pairs ψ, χ and ψ', χ' so that φ is both of the form $(\psi \to \chi)$ and $(\psi' \to \chi')$.

Proof. The formation rules require that if a formula is not atomic, it must start with an opening parenthesis (, $\neg$, or with a quantifier. On the other hand, every formula that start with one of the following symbols must be atomic: a predicate symbol, a function symbol, a constant symbol, $\bot$, $\top$.

So we really only have to show that if φ is of the form $(\psi * \chi)$ and also of the form $(\psi' *' \chi')$, then $\psi \equiv \psi'$, $\chi \equiv \chi'$, and $* = *'$.

So suppose both $\varphi \equiv (\psi * \chi)$ and $\varphi \equiv (\psi' *' \chi')$. Then either $\psi \equiv \psi'$ or not. If it is, clearly $* = *'$ and $\chi \equiv \chi'$, since they then are substrings of φ that begin in the same place and are of the same length. The other case is $\chi \not\equiv \chi'$. Since χ and χ' are both substrings of φ that begin at the same place, one must be a prefix of the other. But this is impossible by Lemma 1.9. □

1.4 Main operator of a Formula

It is often useful to talk about the last operator used in constructing a formula φ. This operator is called the *main operator* of φ. Intuitively, it is the "outermost" operator of φ. For example, the main operator of $\neg\varphi$ is $\neg$, the main operator of $(\varphi \vee \psi)$ is $\vee$, etc.

Definition 1.12 (Main operator)**.** The *main operator* of a formula φ is defined as follows:

1. φ is atomic: φ has no main operator.
2. $\varphi \equiv \neg\psi$: the main operator of φ is $\neg$.
3. $\varphi \equiv (\psi \wedge \chi)$: the main operator of φ is $\wedge$.
4. $\varphi \equiv (\psi \vee \chi)$: the main operator of φ is $\vee$.
5. $\varphi \equiv (\psi \to \chi)$: the main operator of φ is $\to$.
6. $\varphi \equiv (\psi \leftrightarrow \chi)$: the main operator of φ is $\leftrightarrow$.
7. $\varphi \equiv \forall x\, \psi$: the main operator of φ is $\forall$.
8. $\varphi \equiv \exists x\, \psi$: the main operator of φ is $\exists$.

In each case, we intend the specific indicated *occurrence* of the main operator in the formula. For instance, since the formula $((\theta \to \alpha) \to (\alpha \to \theta))$ is of the form $(\psi \to \chi)$ where ψ is $(\theta \to \alpha)$ and χ is $(\alpha \to \theta)$, the second occurrence of $\to$ is the main operator.

This is a *recursive* definition of a function which maps all non-atomic formulas to their main operator occurrence. Because of the way formulas are defined inductively, every formula φ satisfies one of the cases in Definition 1.12. This guarantees that for each non-atomic formula φ a main operator exists. Because each formula satisfies only one of these conditions, and because the smaller formulas from which φ is constructed are uniquely determined in each case, the main operator occurrence of φ is unique, and so we have defined a function.

We call formulas by the following names depending on which symbol their main operator is:

Main operator	Type of formula	Example
none	atomic (formula)	$\bot$, $\top$, $R(t_1, \ldots, t_n)$, $t_1 = t_2$
$\neg$	negation	$\neg\varphi$
$\wedge$	conjunction	$(\varphi \wedge \psi)$
$\vee$	disjunction	$(\varphi \vee \psi)$
$\rightarrow$	conditional	$(\varphi \rightarrow \psi)$
$\forall$	universal (formula)	$\forall x\, \varphi$
$\exists$	existential (formula)	$\exists x\, \varphi$

1.5 Subformulas

It is often useful to talk about the formulas that "make up" a given formula. We call these its *subformulas*. Any formula counts as a subformula of itself; a subformula of φ other than φ itself is a *proper subformula*.

Definition 1.13 (Immediate Subformula)**.** If φ is a formula, the *immediate subformulas* of φ are defined inductively as follows:

1. Atomic formulas have no immediate subformulas.
2. $\varphi \equiv \neg\psi$: The only immediate subformula of φ is ψ.
3. $\varphi \equiv (\psi * \chi)$: The immediate subformulas of φ are ψ and χ ($*$ is any one of the two-place connectives).
4. $\varphi \equiv \forall x\, \psi$: The only immediate subformula of φ is ψ.
5. $\varphi \equiv \exists x\, \psi$: The only immediate subformula of φ is ψ.

Definition 1.14 (Proper Subformula)**.** If φ is a formula, the *proper subformulas* of φ are recursively as follows:

1. Atomic formulas have no proper subformulas.
2. $\varphi \equiv \neg\psi$: The proper subformulas of φ are ψ together with all proper subformulas of ψ.

3. $\varphi \equiv (\psi * \chi)$: The proper subformulas of φ are ψ, χ, together with all proper subformulas of ψ and those of χ.

4. $\varphi \equiv \forall x\, \psi$: The proper subformulas of φ are ψ together with all proper subformulas of ψ.

5. $\varphi \equiv \exists x\, \psi$: The proper subformulas of φ are ψ together with all proper subformulas of ψ.

Definition 1.15 (Subformula)**.** The subformulas of φ are φ itself together with all its proper subformulas.

Note the subtle difference in how we have defined immediate subformulas and proper subformulas. In the first case, we have directly defined the immediate subformulas of a formula φ for each possible form of φ. It is an explicit definition by cases, and the cases mirror the inductive definition of the set of formulas. In the second case, we have also mirrored the way the set of all formulas is defined, but in each case we have also included the proper subformulas of the smaller formulas ψ, χ in addition to these formulas themselves. This makes the definition *recursive*. In general, a definition of a function on an inductively defined set (in our case, formulas) is recursive if the cases in the definition of the function make use of the function itself. To be well defined, we must make sure, however, that we only ever use the values of the function for arguments that come "before" the one we are defining—in our case, when defining "proper subformula" for $(\psi * \chi)$ we only use the proper subformulas of the "earlier" formulas ψ and χ.

1.6 Free Variables and Sentences

Definition 1.16 (Free occurrences of a variable)**.** The *free* occurrences of a variable in a formula are defined inductively as follows:

1. φ is atomic: all variable occurrences in φ are free.

2. $\varphi \equiv \neg\psi$: the free variable occurrences of φ are exactly those of ψ.

3. $\varphi \equiv (\psi * \chi)$: the free variable occurrences of φ are those in ψ together with those in χ.

4. $\varphi \equiv \forall x\, \psi$: the free variable occurrences in φ are all of those in ψ except for occurrences of x.

5. $\varphi \equiv \exists x\, \psi$: the free variable occurrences in φ are all of those in ψ except for occurrences of x.

Definition 1.17 (Bound Variables)**.** An occurrence of a variable in a formula φ is *bound* if it is not free.

Definition 1.18 (Scope)**.** If $\forall x\, \psi$ is an occurrence of a subformula in a formula φ, then the corresponding occurrence of ψ in φ is called the *scope* of the corresponding occurrence of $\forall x$. Similarly for $\exists x$.

If ψ is the scope of a quantifier occurrence $\forall x$ or $\exists x$ in φ, then all occurrences of x which are free in ψ are said to be *bound by* the mentioned quantifier occurrence.

Example 1.19. Here is a somewhat complicated formula φ:

$$\forall x_0\, \underbrace{(A^1_0(x_0) \to A^2_0(x_0, x_1))}_{\psi} \to \exists x_1\, \underbrace{(A^2_1(x_0, x_1) \vee \forall x_0\, \overbrace{\neg A^1_1(x_0)}^{\theta})}_{\chi}$$

ψ is the scope of the first $\forall x_0$, χ is the scope of $\exists x_1$, and θ is the scope of the second $\forall x_0$. The first $\forall x_0$ binds the occurrences of x_0 in ψ, $\exists x_1$ the occurrence of x_1 in χ, and the second $\forall x_0$ binds the occurrence of x_0 in θ. The first occurrence of x_1 and the fourth occurrence of x_0 are free in φ. The last occurrence of x_0 is free in θ, but bound in χ and φ.

Definition 1.20 (Sentence)**.** A formula φ is a *sentence* iff it contains no free occurrences of variables.

1.7 Substitution

Definition 1.21 (Substitution in a term)**.** We define $s[t/x]$, the result of *substituting* t for every occurrence of x in s, recursively:

1. $s \equiv c$: $s[t/x]$ is just s.
2. $s \equiv y$: $s[t/x]$ is also just s, provided y is a variable other than x.
3. $s \equiv x$: $s[t/x]$ is t.
4. $s \equiv f(t_1, \ldots, t_n)$: $s[t/x]$ is $f(t_1[t/x], \ldots, t_n[t/x])$.

Definition 1.22. A term t is *free for* x in φ if none of the free occurrences of x in φ occur in the scope of a quantifier that binds a variable in t.

Definition 1.23 (Substitution in a formula)**.** If φ is a formula, x is a variable, and t is a term free for x in φ, then $\varphi[t/x]$ is the result of substituting t for all free occurrences of x in φ.

1. $\varphi \equiv P(t_1, \ldots, t_n)$: $\varphi[t/x]$ is $P(t_1[t/x], \ldots, t_n[t/x])$.
2. $\varphi \equiv t_1 = t_2$: $\varphi[t/x]$ is $t_1[t/x] = t_2[t/x]$.
3. $\varphi \equiv \neg\psi$: $\varphi[t/x]$ is $\neg\psi[t/x]$.

4. $\varphi \equiv (\psi \wedge \chi)$: $\varphi[t/x]$ is $(\psi[t/x] \wedge \chi[t/x])$.

5. $\varphi \equiv (\psi \vee \chi)$: $\varphi[t/x]$ is $(\psi[t/x] \vee \chi[t/x])$.

6. $\varphi \equiv (\psi \to \chi)$: $\varphi[t/x]$ is $(\psi[t/x] \to \chi[t/x])$.

7. $\varphi \equiv (\psi \leftrightarrow \chi)$: $\varphi[t/x]$ is $(\psi[t/x] \leftrightarrow \chi[t/x])$.

8. $\varphi \equiv \forall y\, \psi$: $\varphi[t/x]$ is $\forall y\, \psi[t/x]$, provided y is a variable other than x; otherwise $\varphi[t/x]$ is just φ.

9. $\varphi \equiv \exists y\, \psi$: $\varphi[t/x]$ is $\exists y\, \psi[t/x]$, provided y is a variable other than x; otherwise $\varphi[t/x]$ is just φ.

Note that substitution may be vacuous: If x does not occur in φ at all, then $\varphi[t/x]$ is just φ.

The restriction that t must be free for x in φ is necessary to exclude cases like the following. If $\varphi \equiv \exists y\, x < y$ and $t \equiv y$, then $\varphi[t/y]$ would be $\exists y\, y < y$. In this case the free variable y is "captured" by the quantifier $\exists y$ upon substitution, and that is undesirable. For instance, we would like it to be the case that whenever $\forall x\, \psi$ holds, so does $\psi[t/x]$. But consider $\forall x\, \exists y\, x < y$ (here ψ is $\exists y\, x < y$). It is sentence that is true about, e.g., the natural numbers: for every number x there is a number y greater than it. If we allowed y as a possible substitution for x, we would end up with $\psi[y/x] \equiv \exists y\, y < y$, which is false. We prevent this by requiring that none of the free variables in t would end up being bound by a quantifier in φ.

We often use the following convention to avoid cumbersume notation: If φ is a formula with a free variable x, we write $\varphi(x)$ to indicate this. When it is clear which φ and x we have in mind, and t is a term (assumed to be free for x in $\varphi(x)$), then we write $\varphi(t)$ as short for $\varphi(x)[t/x]$.

1.8 Structures for First-order Languages

First-order languages are, by themselves, *uninterpreted:* the constant symbols, function symbols, and predicate symbols have no specific meaning attached to them. Meanings are given by specifying a *structure*. It specifies the *domain*, i.e., the objects which the constant symbols pick out, the function symbols operate on, and the quantifiers range over. In addition, it specifies which constant symbols pick out which objects, how a function symbol maps objects to objects, and which objects the predicate symbols apply to. Structures are the basis for *semantic* notions in logic, e.g., the notion of consequence, validity, satisfiablity. They are variously called "structures," "interpretations," or "models" in the literature.

Definition 1.24 (Structures)**.** A *structure* $\mathfrak{M}$, for a language $\mathcal{L}$ of first-order logic consists of the following elements:

1. *Domain:* a non-empty set, $|\mathfrak{M}|$
2. *Interpretation of constant symbols:* for each constant symbol c of $\mathcal{L}$, an element $c^{\mathfrak{M}} \in |\mathfrak{M}|$
3. *Interpretation of predicate symbols:* for each n-place predicate symbol R of $\mathcal{L}$ (other than $=$), an n-ary relation $R^{\mathfrak{M}} \subseteq |\mathfrak{M}|^n$
4. *Interpretation of function symbols:* for each n-place function symbol f of $\mathcal{L}$, an n-ary function $f^{\mathfrak{M}} \colon |\mathfrak{M}|^n \to |\mathfrak{M}|$

Example 1.25. A structure $\mathfrak{M}$ for the language of arithmetic consists of a set, an element of $|\mathfrak{M}|$, $\mathsf{o}^{\mathfrak{M}}$, as interpretation of the constant symbol o, a one-place function $\mathsf{'}^{\mathfrak{M}} \colon |\mathfrak{M}| \to |\mathfrak{M}|$, two two-place functions $+^{\mathfrak{M}}$ and $\times^{\mathfrak{M}}$, both $|\mathfrak{M}|^2 \to |\mathfrak{M}|$, and a two-place relation $<^{\mathfrak{M}} \subseteq |\mathfrak{M}|^2$.

An obvious example of such a structure is the following:

1. $|\mathfrak{N}| = \mathbb{N}$
2. $\mathsf{o}^{\mathfrak{N}} = 0$
3. $\mathsf{'}^{\mathfrak{N}}(n) = n + 1$ for all $n \in \mathbb{N}$
4. $+^{\mathfrak{N}}(n, m) = n + m$ for all $n, m \in \mathbb{N}$
5. $\times^{\mathfrak{N}}(n, m) = n \cdot m$ for all $n, m \in \mathbb{N}$
6. $<^{\mathfrak{N}} = \{\langle n, m\rangle : n \in \mathbb{N}, m \in \mathbb{N}, n < m\}$

The structure $\mathfrak{N}$ for $\mathcal{L}_A$ so defined is called the *standard model of arithmetic*, because it interprets the non-logical constants of $\mathcal{L}_A$ exactly how you would expect.

However, there are many other possible structures for $\mathcal{L}_A$. For instance, we might take as the domain the set $\mathbb{Z}$ of integers instead of $\mathbb{N}$, and define the interpretations of o, $\mathsf{'}$, $+$, $\times$, $<$ accordingly. But we can also define structures for $\mathcal{L}_A$ which have nothing even remotely to do with numbers.

Example 1.26. A structure $\mathfrak{M}$ for the language $\mathcal{L}_Z$ of set theory requires just a set and a single-two place relation. So technically, e.g., the set of people plus the relation "x is older than y" could be used as a structure for $\mathcal{L}_Z$, as well as $\mathbb{N}$ together with $n \geq m$ for $n, m \in \mathbb{N}$.

A particularly interesting structure for $\mathcal{L}_Z$ in which the elements of the domain are actually sets, and the interpretation of $\in$ actually is the relation "x is an element of y" is the structure $\mathfrak{HF}$ of *hereditarily finite sets*:

1. $|\mathfrak{HF}| = \emptyset \cup \wp(\emptyset) \cup \wp(\wp(\emptyset)) \cup \wp(\wp(\wp(\emptyset))) \cup \ldots$;
2. $\in^{\mathfrak{HF}} = \{\langle x, y\rangle : x, y \in |\mathfrak{HF}|, x \in y\}$.

Recall that a term is *closed* if it contains no variables.

Definition 1.27 (Value of closed terms)**.** If t is a closed term of the langage $\mathcal{L}$ and $\mathfrak{M}$ is a structure for $\mathcal{L}$, the *value* $\mathrm{Val}^{\mathfrak{M}}(t)$ is defined as follows:

1. If t is just the constant symbol c, then $\mathrm{Val}^{\mathfrak{M}}(c) = c^{\mathfrak{M}}$.
2. If t is of the form $f(t_1, \ldots, t_n)$, then $\mathrm{Val}^{\mathfrak{M}}(t)$ is $f^{\mathfrak{M}}(\mathrm{Val}^{\mathfrak{M}}(t_1), \ldots, \mathrm{Val}^{\mathfrak{M}}(t_n))$.

Definition 1.28 (Covered structure)**.** A structure is *covered* if every element of the domain is the value of some closed term.

Example 1.29. Let $\mathcal{L}$ be the language with constant symbols zero, one, two, ..., the binary predicate symbols $=$ and $<$, and the binary function symbols $+$ and $\times$. Then a structure $\mathfrak{M}$ for $\mathcal{L}$ is the one with domain $|\mathfrak{M}| = \{0, 1, 2, \ldots\}$ and name assignment $\mathit{zero}^{\mathfrak{M}} = 0$, $\mathit{one}^{\mathfrak{M}} = 1$, $\mathit{two}^{\mathfrak{M}} = 2$, and so forth. For the binary relation symbol $<$, the set $<^{\mathfrak{M}}$ is the set of all pairs $\langle c_1, c_2 \rangle \in |\mathfrak{M}|^2$ such that the integer c_1 is less than the integer c_2: for example, $\langle 1, 3 \rangle \in <^{\mathfrak{M}}$ but $\langle 2, 2 \rangle \notin <^{\mathfrak{M}}$. For the binary function symbol $+$, define $+^{\mathfrak{M}}$ in the usual way—for example, $+^{\mathfrak{M}}(2, 3)$ maps to 5, and similarly for the binary function symbol $\times$. Hence, the value of four is just 4, and the value of $\times(\mathit{two}, +(\mathit{three}, \mathit{zero}))$ (or in infix notation, $\mathit{two} \times (\mathit{three} + \mathit{zero})$) is

$$\begin{aligned}
\mathrm{Val}^{\mathfrak{M}}(\times(\mathit{two}, +(\mathit{three}, \mathit{zero})) &= \times^{\mathfrak{M}}(\mathrm{Val}^{\mathfrak{M}}(\mathit{two}), \mathrm{Val}^{\mathfrak{M}}(\mathit{two}, +(\mathit{three}, \mathit{zero}))) \\
&= \times^{\mathfrak{M}}(\mathrm{Val}^{\mathfrak{M}}(\mathit{two}), +^{\mathfrak{M}}(\mathrm{Val}^{\mathfrak{M}}(\mathit{three}), \mathrm{Val}^{\mathfrak{M}}(\mathit{zero}))) \\
&= \times^{\mathfrak{M}}(\mathit{two}^{\mathfrak{M}}, +^{\mathfrak{M}}(\mathit{three}^{\mathfrak{M}}, \mathit{zero}^{\mathfrak{M}})) \\
&= \times^{\mathfrak{M}}(2, +^{\mathfrak{M}}(3, 0)) \\
&= \times^{\mathfrak{M}}(2, 3) \\
&= 6
\end{aligned}$$

The stipulations we make as to what counts as a structure impact our logic. For example, the choice to prevent empty domains ensures, given the usual account of satisfaction (or truth) for quantified sentences, that $\exists x\, (\varphi(x) \vee \neg\varphi(x))$ is valid—that is, a logical truth. And the stipulation that all constant symbols must refer to an object in the domain ensures that the existential generalization is a sound pattern of inference: $\varphi(a)$, therefore $\exists x\, \varphi(x)$. If we allowed names to refer outside the domain, or to not refer, then we would be on our way to a *free logic*, in which existential generalization requires an additional premise: $\varphi(a)$ and $\exists x\, x = a$, therefore $\exists x\, \varphi(x)$.

1.9 Satisfaction of a Formula in a Structure

The basic notion that relates expressions such as terms and formulas, on the one hand, and structures on the other, are those of *value* of a term and *satisfac-*

tion of a formula. Informally, the value of a term is an element of a structure—if the term is just a constant, its value is the object assigned to the constant by the structure, and if it is built up using function symbols, the value is computed from the values of constants and the functions assigned to the functions in the term. A formula is *satisfied* in a structure if the interpretation given to the predicates makes the formula true in the domain of the structure. This notion of satisfaction is specified inductively: the specification of the structure directly states when atomic formulas are satisfied, and we define when a complex formula is satisfied depending on the main connective or quantifier and whether or not the immediate subformulas are satisfied. The case of the quantifiers here is a bit tricky, as the immediate subformula of a quantified formula has a free variable, and structures don't specify the values of variables. In order to deal with this difficulty, we also introduce *variable assignments* and define satisfaction not with respect to a structure alone, but with respect to a structure plus a variable assignment.

Definition 1.30 (Variable Assignment)**.** A *variable assignment* s for a structure $\mathfrak{M}$ is a function which maps each variable to an element of $|\mathfrak{M}|$, i.e., $s\colon \mathrm{Var} \to |\mathfrak{M}|$.

A structure assigns a value to each constant symbol, and a variable assignment to each variable. But we want to use terms built up from them to also name elements of the domain. For this we define the value of terms inductively. For constant symbols and variables the value is just as the structure or the variable assignment specifies it; for more complex terms it is computed recursively using the functions the structure assigns to the function symbols.

Definition 1.31 (Value of Terms)**.** If t is a term of the language $\mathcal{L}$, $\mathfrak{M}$ is a structure for $\mathcal{L}$, and s is a variable assignment for $\mathfrak{M}$, the *value* $\mathrm{Val}^{\mathfrak{M}}_s(t)$ is defined as follows:

1. $t \equiv c$: $\mathrm{Val}^{\mathfrak{M}}_s(t) = c^{\mathfrak{M}}$.
2. $t \equiv x$: $\mathrm{Val}^{\mathfrak{M}}_s(t) = s(x)$.
3. $t \equiv f(t_1, \ldots, t_n)$:
$$\mathrm{Val}^{\mathfrak{M}}_s(t) = f^{\mathfrak{M}}(\mathrm{Val}^{\mathfrak{M}}_s(t_1), \ldots, \mathrm{Val}^{\mathfrak{M}}_s(t_n)).$$

Definition 1.32 (x-Variant)**.** If s is a variable assignment for a structure $\mathfrak{M}$, then any variable assignment s' for $\mathfrak{M}$ which differs from s at most in what it assigns to x is called an *x-variant* of s. If s' is an x-variant of s we write $s \sim_x s'$.

Note that an x-variant of an assignment s does not *have* to assign something different to x. In fact, every assignment counts as an x-variant of itself.

Definition 1.33 (Satisfaction)**.** Satisfaction of a formula φ in a structure $\mathfrak{M}$ relative to a variable assignment s, in symbols: $\mathfrak{M}, s \models \varphi$, is defined recursively as follows. (We write $\mathfrak{M}, s \not\models \varphi$ to mean "not $\mathfrak{M}, s \models \varphi$.")

1. $\varphi \equiv \bot$: not $\mathfrak{M}, s \models \varphi$.
2. $\varphi \equiv \top$: $\mathfrak{M}, s \models \varphi$.
3. $\varphi \equiv R(t_1, \ldots, t_n)$: $\mathfrak{M}, s \models \varphi$ iff $\langle \mathrm{Val}^{\mathfrak{M}}_s(t_1), \ldots, \mathrm{Val}^{\mathfrak{M}}_s(t_n) \rangle \in R^{\mathfrak{M}}$.
4. $\varphi \equiv t_1 = t_2$: $\mathfrak{M}, s \models \varphi$ iff $\mathrm{Val}^{\mathfrak{M}}_s(t_1) = \mathrm{Val}^{\mathfrak{M}}_s(t_2)$.
5. $\varphi \equiv \neg\psi$: $\mathfrak{M}, s \models \varphi$ iff $\mathfrak{M}, s \not\models \psi$.
6. $\varphi \equiv (\psi \wedge \chi)$: $\mathfrak{M}, s \models \varphi$ iff $\mathfrak{M}, s \models \psi$ and $\mathfrak{M}, s \models \chi$.
7. $\varphi \equiv (\psi \vee \chi)$: $\mathfrak{M}, s \models \varphi$ iff $\mathfrak{M}, s \models \varphi$ or $\mathfrak{M}, s \models \psi$ (or both).
8. $\varphi \equiv (\psi \to \chi)$: $\mathfrak{M}, s \models \varphi$ iff $\mathfrak{M}, s \not\models \psi$ or $\mathfrak{M}, s \models \chi$ (or both).
9. $\varphi \equiv (\psi \leftrightarrow \chi)$: $\mathfrak{M}, s \models \varphi$ iff either both $\mathfrak{M}, s \models \psi$ and $\mathfrak{M}, s \models \chi$, or neither $\mathfrak{M}, s \models \psi$ nor $\mathfrak{M}, s \models \chi$.
10. $\varphi \equiv \forall x\, \psi$: $\mathfrak{M}, s \models \varphi$ iff for every x-variant s' of s, $\mathfrak{M}, s' \models \psi$.
11. $\varphi \equiv \exists x\, \psi$: $\mathfrak{M}, s \models \varphi$ iff there is an x-variant s' of s so that $\mathfrak{M}, s' \models \psi$.

The variable assignments are important in the last two clauses. We cannot define satisfaction of $\forall x\, \psi(x)$ by "for all $a \in |\mathfrak{M}|$, $\mathfrak{M} \models \psi(a)$. We cannot define satisfaction of $\exists x\, \psi(x)$ by "for at least one $a \in |\mathfrak{M}|$, $\mathfrak{M} \models \psi(a)$. The reason is that a is not symbol of the language, and so $\psi(a)$ is not a formula (that is, $\psi[a/x]$ is undefined). We can also not assume that we have constant symbols or terms available that name every element of $\mathfrak{M}$, since there is nothing in the definition of structures that requires it. Even in the standard language the set of constant symbols is denumerable, so if $|\mathfrak{M}|$ is not enumerable there aren't even enough constant symbols to name every object.

A variable assignment s provides a value for *every* variable in the language. This is of course not necessary: whether or not a formula φ is satisfied in a structure with respect to s only depends on the assignments s makes to the free variables that actually occur in φ. This is the content of the next theorem. We require variable assignments to assign values to all variables simply because it makes things a lot easier.

Proposition 1.34. *If $x_1, \ldots, x_n$ are the only free variables in φ and $s(x_i) = s'(x_i)$ for $i = 1, \ldots, n$, then $\mathfrak{M}, s \models \varphi$ iff $\mathfrak{M}, s' \models \varphi$.*

Proof. We use induction on the complexity of φ. For the base case, where φ is atomic, φ can be: $\top$, $\bot$, $R(t_1, \ldots, t_k)$ for a k-place predicate R and terms t_1, $\ldots$, t_k, or $t_1 = t_2$ for terms t_1 and t_2.

1. $\varphi \equiv \top$: both $\mathfrak{M}, s \models \varphi$ and $\mathfrak{M}, s' \models \varphi$.

2. $\varphi \equiv \bot$: both $\mathfrak{M}, s \not\models \varphi$ and $\mathfrak{M}, s' \not\models \varphi$.

3. $\varphi \equiv R(t_1, \ldots, t_k)$: let $\mathfrak{M}, s \models \varphi$. Then $\langle \mathrm{Val}^{\mathfrak{M}}_s(t_1), \ldots, \mathrm{Val}^{\mathfrak{M}}_s(t_k)\rangle \in R^{\mathfrak{M}}$. For $i = 1, \ldots, k$, if t_i is a constant, then $\mathrm{Val}^{\mathfrak{M}}_s(t_i) = \mathrm{Val}^{\mathfrak{M}}(t_i) = \mathrm{Val}^{\mathfrak{M}}_{s'}(t_i)$. If t_i is a free variable, then since the mappings s and s' agree on all free variables, $\mathrm{Val}^{\mathfrak{M}}_s(t_i) = s(t_i) = s'(t_i) = \mathrm{Val}^{\mathfrak{M}}_{s'}(t_i)$. Similarly, if t_i is of the form $f(t'_1, \ldots, t'_j)$, we will also get $\mathrm{Val}^{\mathfrak{M}}_s(t_i) = \mathrm{Val}^{\mathfrak{M}}_{s'}(t_i)$. Hence, $\mathrm{Val}^{\mathfrak{M}}_s(t_i) = \mathrm{Val}^{\mathfrak{M}}_{s'}(t_i)$ for any term t_i for $i = 1, \ldots, k$, so we also have $\langle \mathrm{Val}^{\mathfrak{M}}_{s'}(t_i), \ldots, \mathrm{Val}^{\mathfrak{M}}_{s'}(t_k)\rangle \in R^{\mathfrak{M}}$.

4. $\varphi \equiv t_1 = t_2$: if $\mathfrak{M}, s \models \varphi$, $\mathrm{Val}^{\mathfrak{M}}_{s'}(t_1) = \mathrm{Val}^{\mathfrak{M}}_s(t_1) = \mathrm{Val}^{\mathfrak{M}}_s(t_2) = \mathrm{Val}^{\mathfrak{M}}_{s'}(t_2)$, so $\mathfrak{M}, s' \models t_1 = t_2$.

Now assume $\mathfrak{M}, s \models \psi$ iff $\mathfrak{M}, s' \models \psi$ for all formulas ψ less complex than φ. The induction step proceeds by cases determined by the main operator of φ. In each case, we only demonstrate the forward direction of the biconditional; the proof of the reverse direction is symmetrical.

1. $\varphi \equiv \neg\psi$: if $\mathfrak{M}, s \models \varphi$, then $\mathfrak{M}, s \not\models \psi$, so by the induction hypothesis, $\mathfrak{M}, s' \not\models \psi$, hence $\mathfrak{M}, s' \models \varphi$.

2. $\varphi \equiv \psi \wedge \chi$: if $\mathfrak{M}, s \models \varphi$, then $\mathfrak{M}, s \models \psi$ and $\mathfrak{M}, s \models \chi$, so by induction hypothesis, $\mathfrak{M}, s' \models \psi$ and $\mathfrak{M}, s' \models \chi$. Hence, $\mathfrak{M}, s' \models \varphi$.

3. $\varphi \equiv \psi \vee \chi$: if $\mathfrak{M}, s \models \varphi$, then $\mathfrak{M}, s \models \psi$ or $\mathfrak{M}, s \models \chi$. By induction hypothesis, $\mathfrak{M}, s' \models \psi$ or $\mathfrak{M}, s' \models \chi$, so $\mathfrak{M}, s' \models \varphi$.

4. $\varphi \equiv \psi \to \chi$: if $\mathfrak{M}, s \models \varphi$, then $\mathfrak{M}, s \not\models \psi$ or $\mathfrak{M}, s \models \chi$. By the induction hypothesis, $\mathfrak{M}, s' \not\models \psi$ or $\mathfrak{M}, s' \models \chi$, so $\mathfrak{M}, s' \models \varphi$.

5. $\varphi \equiv \psi \leftrightarrow \chi$: if $\mathfrak{M}, s \models \varphi$, then either $\mathfrak{M}, s \models \psi$ and $\mathfrak{M}, s \models \chi$, or $\mathfrak{M}, s \not\models \psi$ and $\mathfrak{M}, s \not\models \chi$. By the induction hypothesis, either $\mathfrak{M}, s' \models \psi$ and $\mathfrak{M}, s' \models \chi$ or $\mathfrak{M}, s' \not\models \psi$ and $\mathfrak{M}, s' \not\models \chi$. In either case, $\mathfrak{M}, s' \models \varphi$.

6. $\varphi \equiv \exists x\, \psi$: if $\mathfrak{M}, s \models \varphi$, there is an x-variant $\bar{s}$ of s so that $\mathfrak{M}, \bar{s} \models \psi$. Let $\bar{s}'$ denote the x-variant of s' that assigns the same thing to x as does $\bar{s}$: then by the induction hypothesis, $\mathfrak{M}, \bar{s}' \models \psi$. Hence, there is an x-variant of s' that satisfies ψ, so $\mathfrak{M}, s' \models \varphi$.

7. $\varphi \equiv \forall x\, \psi$: if $\mathfrak{M}, s \models \varphi$, then for every x-variant $\bar{s}$ of s, $\mathfrak{M}, \bar{s} \models \psi$. Hence, if $\bar{s}'$ is the x-variant of s' that assigns the same thing to x as does $\bar{s}$, then we have $\mathfrak{M}, \bar{s}' \models \psi$. Hence, every x-variant of s' satisfies ψ, so $\mathfrak{M}, s' \models \varphi$

By induction, we get that $\mathfrak{M}, s \models \varphi$ iff $\mathfrak{M}, s' \models \varphi$ whenever $x_1, \ldots, x_n$ are the only free variables in φ and $s(x_i) = s'(x_i)$ for $i = 1, \ldots, n$. □

Definition 1.35. If φ is a sentence, we say that a structure $\mathfrak{M}$ *satisfies* φ, $\mathfrak{M} \models \varphi$, iff $\mathfrak{M}, s \models \varphi$ for all variable assignments s.

Proposition 1.36. *Suppose $\varphi(x)$ only contains x free, and $\mathfrak{M}$ is a structure. Then:*

1. *$\mathfrak{M} \models \exists x\, \varphi(x)$ iff $\mathfrak{M}, s \models \varphi(x)$ for at least one variable assignment s.*
2. *$\mathfrak{M} \models \forall x\, \varphi(x)$ iff $\mathfrak{M}, s \models \varphi(x)$ for all variable assignments s.*

Proof. Exercise. □

1.10 Extensionality

Extensionality, sometimes called relevance, can be expressed informally as follows: the only thing that bears upon the satisfaction of formula φ in a structure $\mathfrak{M}$ relative to a variable assignment s, are the assignments made by $\mathfrak{M}$ and s to the elements of the language that actually appear in φ.

One immediate consequence of extensionality is that where two structures $\mathfrak{M}$ and $\mathfrak{M}'$ agree on all the elements of the language appearing in a sentence φ and have the same domain, $\mathfrak{M}$ and $\mathfrak{M}'$ must also agree on φ itself.

Proposition 1.37 (Extensionality)**.** *Let φ be a sentence, and $\mathfrak{M}$ and $\mathfrak{M}'$ be structures. If $c^{\mathfrak{M}} = c^{\mathfrak{M}'}$, $R^{\mathfrak{M}} = R^{\mathfrak{M}'}$, and $f^{\mathfrak{M}} = f^{\mathfrak{M}'}$ for every constant symbol c, relation symbol R, and function symbol f occurring in φ, then $\mathfrak{M} \models \varphi$ iff $\mathfrak{M}' \models \varphi$.*

Moreover, the value of a term, and whether or not a structure satisfies a formula, only depends on the values of its subterms.

Proposition 1.38. *Let $\mathfrak{M}$ be a structure, t and t' terms, and s a variable assignment. Let $s' \sim_x s$ be the x-variant of s given by $s'(x) = \mathrm{Val}^{\mathfrak{M}}_s(t')$. Then $\mathrm{Val}^{\mathfrak{M}}_s(t[t'/x]) = \mathrm{Val}^{\mathfrak{M}}_{s'}(t)$.*

Proof. By induction on t.

1. If t is a constant, say, $t \equiv c$, then $t[t'/x] = c$, and $\mathrm{Val}^{\mathfrak{M}}_s(c) = c^{\mathfrak{M}} = \mathrm{Val}^{\mathfrak{M}}_{s'}(c)$.
2. If t is a variable other than x, say, $t \equiv y$, then $t[t'/x] = y$, and $\mathrm{Val}^{\mathfrak{M}}_s(y) = \mathrm{Val}^{\mathfrak{M}}_{s'}(y)$ since $s' \sim_x s$.
3. If $t \equiv x$, then $t[t'/x] = t'$. But $\mathrm{Val}^{\mathfrak{M}}_{s'}(x) = \mathrm{Val}^{\mathfrak{M}}_s(t')$ by definition of s'.
4. If $t \equiv f(t_1, \ldots, t_n)$ then we have:

$$\begin{aligned}
\mathrm{Val}^{\mathfrak{M}}_s(t[t'/x]) &= \mathrm{Val}^{\mathfrak{M}}_s(f(t_1[t'/x], \ldots, t_n[t'/x])) \text{ by definition of } t[t'/x] \\
&= f^{\mathfrak{M}}(\mathrm{Val}^{\mathfrak{M}}_s(t_1[t'/x]), \ldots, \mathrm{Val}^{\mathfrak{M}}_s(t_n[t'/x])) \text{ by definition of } \mathrm{Val}^{\mathfrak{M}}_s(f(\ldots)) \\
&= f^{\mathfrak{M}}(\mathrm{Val}^{\mathfrak{M}}_{s'}(t_1), \ldots, \mathrm{Val}^{\mathfrak{M}}_{s'}(t_n)) \text{ by induction hypothesis} \\
&= \mathrm{Val}^{\mathfrak{M}}_{s'}(t) \text{ by definition of } \mathrm{Val}^{\mathfrak{M}}_{s'}(f(\ldots))
\end{aligned}$$

□

Proposition 1.39. *Let $\mathfrak{M}$ be a structure, φ a formula, t a term, and s a variable assignment. Let $s' \sim_x s$ be the x-variant of s given by $s'(x) = \mathrm{Val}^{\mathfrak{M}}_s(t)$. Then $\mathfrak{M}, s \models \varphi[t/x]$ iff $\mathfrak{M}, s' \models \varphi$.*

Proof. Exercise. □

1.11 Semantic Notions

Give the definition of structures for first-order languages, we can define some basic semantic properties of and relationships between sentences. The simplest of these is the notion of *validity* of a sentence. A sentence is valid if it is satisfied in every structure. Valid sentences are those that are satisfied regardless of how the non-logical symbols in it are interpreted. Valid sentences are therefore also called *logical truths*—they are true, i.e., satisfied, in any structure and hence their truth depends only on the logical symbols occurring in them and their syntactic structure, but not on the non-logical symbols or their interpretation.

Definition 1.40 (Validity)**.** A sentence φ is *valid*, $\models \varphi$, iff $\mathfrak{M} \models \varphi$ for every structure $\mathfrak{M}$.

Definition 1.41 (Entailment)**.** A set of sentences Γ *entails* a sentence φ, $\Gamma \models \varphi$, iff for every structure $\mathfrak{M}$ with $\mathfrak{M} \models \Gamma$, $\mathfrak{M} \models \varphi$.

Definition 1.42 (Satisfiability)**.** A set of sentences Γ is *satisfiable* if $\mathfrak{M} \models \Gamma$ for some structure $\mathfrak{M}$. If Γ is not satisfiable it is called *unsatisfiable*.

Proposition 1.43. *A sentence φ is valid iff $\Gamma \models \varphi$ for every set of sentences Γ.*

Proof. For the forward direction, let φ be valid, and let Γ be a set of sentences. Let $\mathfrak{M}$ be a structure so that $\mathfrak{M} \models \Gamma$. Since φ is valid, $\mathfrak{M} \models \varphi$, hence $\Gamma \models \varphi$.

For the contrapositive of the reverse direction, let φ be invalid, so there is a structure $\mathfrak{M}$ with $\mathfrak{M} \not\models \varphi$. When $\Gamma = \{\top\}$, since $\top$ is valid, $\mathfrak{M} \models \Gamma$. Hence, there is a structure $\mathfrak{M}$ so that $\mathfrak{M} \models \Gamma$ but $\mathfrak{M} \not\models \varphi$, hence Γ does not entail φ. □

Proposition 1.44. *$\Gamma \models \varphi$ iff $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable.*

Proof. For the forward direction, suppose $\Gamma \models \varphi$ and suppose to the contrary that there is a structure $\mathfrak{M}$ so that $\mathfrak{M} \models \Gamma \cup \{\neg\varphi\}$. Since $\mathfrak{M} \models \Gamma$ and $\Gamma \models \varphi$, $\mathfrak{M} \models \varphi$. Also, since $\mathfrak{M} \models \Gamma \cup \{\neg\varphi\}$, $\mathfrak{M} \models \neg\varphi$, so we have both $\mathfrak{M} \models \varphi$ and $\mathfrak{M} \not\models \varphi$, a contradiction. Hence, there can be no such structure $\mathfrak{M}$, so $\Gamma \cup \{\varphi\}$ is unsatisfiable.

For the reverse direction, suppose $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable. So for every structure $\mathfrak{M}$, either $\mathfrak{M} \not\models \Gamma$ or $\mathfrak{M} \models \varphi$. Hence, for every structure $\mathfrak{M}$ with $\mathfrak{M} \models \Gamma$, $\mathfrak{M} \models \varphi$, so $\Gamma \models \varphi$. □

Proposition 1.45. *If $\Gamma \subseteq \Gamma'$ and $\Gamma \models \varphi$, then $\Gamma' \models \varphi$.*

Proof. Suppose that $\Gamma \subseteq \Gamma'$ and $\Gamma \models \varphi$. Let $\mathfrak{M}$ be such that $\mathfrak{M} \models \Gamma'$; then $\mathfrak{M} \models \Gamma$, and since $\Gamma \models \varphi$, we get that $\mathfrak{M} \models \varphi$. Hence, whenever $\mathfrak{M} \models \Gamma'$, $\mathfrak{M} \models \varphi$, so $\Gamma' \models \varphi$. □

Theorem 1.46 (Semantic Deduction Theorem)**.** *$\Gamma \cup \{\varphi\} \models \psi$ iff $\Gamma \models \varphi \to \psi$.*

Proof. For the forward direction, let $\Gamma \cup \{\varphi\} \models \psi$ and let $\mathfrak{M}$ be a structure so that $\mathfrak{M} \models \Gamma$. If $\mathfrak{M} \models \varphi$, then $\mathfrak{M} \models \Gamma \cup \{\varphi\}$, so since $\Gamma \cup \{\varphi\}$ entails ψ, we get $\mathfrak{M} \models \psi$. Therefore, $\mathfrak{M} \models \varphi \to \psi$, so $\Gamma \models \varphi \to \psi$.

For the reverse direction, let $\Gamma \models \varphi \to \psi$ and $\mathfrak{M}$ be a structure so that $\mathfrak{M} \models \Gamma \cup \{\varphi\}$. Then $\mathfrak{M} \models \Gamma$, so $\mathfrak{M} \models \varphi \to \psi$, and since $\mathfrak{M} \models \varphi$, $\mathfrak{M} \models \psi$. Hence, whenever $\mathfrak{M} \models \Gamma \cup \{\varphi\}$, $\mathfrak{M} \models \psi$, so $\Gamma \cup \{\varphi\} \models \psi$. □

Chapter 2

Theories and Their Models

2.1 Introduction

The development of the axiomatic method is a significant achievement in the history of science, and is of special importnace in the history of mathematics. An axiomatic development of a field involves the clarification of many questions: What is the field about? What are the most fundamental concepts? How are they related? Can all the concepts of the field be defined in terms of these fundamental concepts? What laws do, and must, these concepts obey?

The axiomatic method and logic were made for each other. Formal logic provides the tools for formulating axiomatic theories, for proving theorems from the axioms of the theory in a precisely specified way, for studying the properties of all systems satisfying the axioms in a systematic way.

Definition 2.1. A set of sentences Γ is *closed* iff, whenever $\Gamma \models \varphi$ then $\varphi \in \Gamma$. The *closure* of a set of sentences Γ is $\{\varphi : \Gamma \models \varphi\}$.

We say that Γ is *axiomatized by* a set of sentences Δ if Γ is the closure of Δ

We can think of an axiomatic theory as the set of sentences that is axiomatized by its set of axioms Δ. In other words, when we have a first-order language which contains non-logical symbols for the primitives of the axiomatically developed science we wish to study, together with a set of sentences that express the fundamental laws of the science, we can think of the theory as represented by all the sentences in this language that are entailed by the axioms. This ranges from simple examples with only a single primitive and simple axioms, such as the theory of partial orders, to complex theories such as Newtonian mechanics.

The important logical facts that make this formal approach to the axiomatic method so important are the following. Suppose Γ is an axiom system for a theory, i.e., a set of sentences.

1. We can state precisely when an axiom system captures an intended class of structures. That is, if we are interested in a certain class of structures, we will successfully capture that class by an axiom system Γ iff the structures are exactly those $\mathfrak{M}$ such that $\mathfrak{M} \models \Gamma$.

2. We may fail in this respect because there are $\mathfrak{M}$ such that $\mathfrak{M} \models \Gamma$, but $\mathfrak{M}$ is not one of the structures we intend. This may lead us to add axioms which are not true in $\mathfrak{M}$.

3. If we are successful at least in the respect that Γ is true in all the intended structures, then a sentence φ is true in all intended structures whenever $\Gamma \models \varphi$. Thus we can use logical tools (such as proof methods) to show that sentences are true in all intended structures simply by showing that they are entailed by the axioms.

4. Sometimes we don't have intended structures in mind, but instead start from the axioms themselves: we begin with some primitives that we want to satisfy certain laws which we codify in an axiom system. One thing that we would like to verify right away is that the axioms do not contradict each other: if they do, there can be no concepts that obey these laws, and we have tried to set up an incoherent theory. We can verify that this doesn't happen by finding a model of Γ. And if there are models of our theory, we can use logical methods to investigate them, and we can also use logical methods to construct models.

5. The independence of the axioms is likewise an important question. It may happen that one of the axioms is actually a consequence of the others, and so is redundant. We can prove that an axiom φ in Γ is redundant by proving $\Gamma \setminus \{\varphi\} \models \varphi$. We can also prove that an axiom is not redundant by showing that $(\Gamma \setminus \{\varphi\}) \cup \{\neg\varphi\}$ is satisfiable. For instance, this is how it was shown that the parallel postulate is independent of the other axioms of geometry.

6. Another important question is that of definability of concepts in a theory: The choice of the language determines what the models of a theory consists of. But not every aspect of a theory must be represented separately in its models. For instance, every ordering $\leq$ determines a corresponding strict ordering $<$—given one, we can define the other. So it is not necessary that a model of a theory involving such an order must *also* contain the corresponding strict ordering. When is it the case, in general, that one relation can be defined in terms of others? When is it impossible to define a relation in terms of other (and hence must add it to the primitives of the language)?

2.2 Expressing Properties of Structures

It is often useful and important to express conditions on functions and relations, or more generally, that the functions and relations in a structure satisfy these conditions. For instance, we would like to have ways of distinguishing those structures for a language which "capture" what we want the predicate symbols to "mean" from those that do not. Of course we're completely free to specify which structures we "intend," e.g., we can specify that the interpretation of the predicate symbol $\leq$ must be an ordering, or that we are only interested in interpretations of $\mathcal{L}$ in which the domain consists of sets and $\in$ is interpreted by the "is an element of" relation. But can we do this with sentences of the language? In other words, which conditions on a structure $\mathfrak{M}$ can we express by a sentence (or perhaps a set of sentences) in the language of $\mathfrak{M}$? There are some conditions that we will not be able to express. For instance, there is no sentence of $\mathcal{L}_A$ which is only true in a structure $\mathfrak{M}$ if $|\mathfrak{M}| = \mathbb{N}$. We cannot express "the domain contains only natural numbers." But there are "structural properties" of structures that we perhaps can express. Which properties of structures can we express by sentences? Or, to put it another way, which collections of structures can we describe as those making a sentence (or set of sentences) true?

Definition 2.2. Let Γ be a set of sentences in a language $\mathcal{L}$. We say that a structure $\mathfrak{M}$ *is a model of* Γ if $\mathfrak{M} \models \varphi$ for all $\varphi \in \Gamma$.

Example 2.3. The sentence $\forall x\, x \leq x$ is true in $\mathfrak{M}$ iff $\leq^{\mathfrak{M}}$ is a reflexive relation. The sentence $\forall x \forall y\, ((x \leq y \wedge y \leq x) \rightarrow x = y)$ is true in $\mathfrak{M}$ iff $\leq^{\mathfrak{M}}$ is antisymmetric. The sentence $\forall x \forall y \forall z\, ((x \leq y \wedge y \leq z) \rightarrow x \leq z)$ is true in $\mathfrak{M}$ iff $\leq^{\mathfrak{M}}$ is transitive. Thus, the models of

$$\begin{aligned}\{\quad & \forall x\, x \leq x,\\ & \forall x \forall y\, ((x \leq y \wedge y \leq x) \rightarrow x = y),\\ & \forall x \forall y \forall z\, ((x \leq y \wedge y \leq z) \rightarrow x \leq z) \quad \}\end{aligned}$$

are exactly those structures in which $\leq^{\mathfrak{M}}$ is reflexive, symmetric, and transitive, i.e., a partial order. Hence, we can take them as axioms for the *first-order theory of partial orders*.

2.3 Examples of First-Order Theories

Example 2.4. The theory of strict linear orders in the language $\mathcal{L}_<$ is axiomatized by the set

$$\begin{aligned}& \forall x \neg x < x,\\ & \forall x \forall y\, ((x < y \vee y < x) \vee x = y),\\ & \forall x \forall y \forall z\, ((x < y \wedge y < z) \rightarrow x < z)\end{aligned}$$

It completely captures the intended structures: every strict linear order is a model of this axiom system, and vice versa, if R is a linear order on a set X, then the structure $\mathfrak{M}$ with $|\mathfrak{M}| = X$ and $<^{\mathfrak{M}} = R$ is a model of this theory.

Example 2.5. The theory of groups in the language 1 (constant symbol), $\cdot$ (two-place function symbol) is axiomatized by

$$\begin{aligned}
&\forall x\,(x \cdot 1) = x\\
&\forall x\,\forall y\,\forall z\,(x \cdot (y \cdot z)) = ((x \cdot y) \cdot z)\\
&\forall x\,\exists y\,(x \cdot y) = 1
\end{aligned}$$

Example 2.6. The theory of Peano arithmetic is axiomatized by the following sentences in the language of arithmetic $\mathcal{L}_A$.

$$\begin{aligned}
&\neg\exists x\, x' = 0\\
&\forall x\,\forall y\,(x' = y' \to x = y)\\
&\forall x\,\forall y\,(x < y \leftrightarrow \exists z\,(x + z' = y))\\
&\forall x\,(x + 0) = x\\
&\forall x\,\forall y\,(x + y') = (x + y)'\\
&\forall x\,(x \times 0) = 0\\
&\forall x\,\forall y\,(x \times y') = ((x \times y) + x)
\end{aligned}$$

plus all sentences of the form

$$(\varphi(0) \wedge \forall x\,(\varphi(x) \to \varphi(x'))) \to \forall x\,\varphi(x)$$

Since there are infinitely many sentences of the latter form, this axiom system is infinite. The latter form is called the *induction schema*. (Actually, the induction schema is a bit more complicated than we let on here.)

The third axiom is an *explicit definition* of $<$.

Example 2.7. The theory of pure sets plays an important role in the foundations (and in the philosophy) of mathematics. A set is pure if all its elements are also pure sets. The empty set counts therefore as pure, but a set that has something as an element that is not a set would not be pure. So the pure sets are those that are formed just from the empty set and no "urelements," i.e., objects that are not themselves sets.

The following might be considered as an axiom system for a theory of pure sets:

$$\begin{aligned}
&\exists x\,\neg\exists y\, y \in x\\
&\forall x\,\forall y\,(\forall z(z \in x \leftrightarrow z \in y) \to x = y)\\
&\forall x\,\forall y\,\exists z\,\forall u\,(u \in z \leftrightarrow (u = x \vee u = y))\\
&\forall x\,\exists y\,\forall z\,(z \in y \leftrightarrow \exists u\,(z \in u \vee u \in x))
\end{aligned}$$

plus all sentences of the form

$$\exists x \, \forall y \, (y \in x \leftrightarrow \varphi(y))$$

The first axiom says that there is a set with no elements (i.e., $\emptyset$ exists); the second says that sets are extensional; the third that for any sets X and Y, the set $\{X, Y\}$ exists; the fourth that for any sets X and Y, the set $X \cup Y$ exists.

The sentences mentioned last are collectively called the *naive comprehension scheme*. It essentially says that for every $\varphi(x)$, the set $\{x : \varphi(x)\}$ exists—so at first glance a true, useful, and perhaps even necessary axiom. It is called "naive" because, as it turns out, it makes this theory unsatisfiable: if you take $\varphi(y)$ to be $\neg y \in y$, you get the sentence

$$\exists x \, \forall y \, (y \in x \leftrightarrow \neg y \in y)$$

and this sentence is not satisfied in any structure.

Example 2.8. In the area of *mereology*, the relation of *parthood* is a fundamental relation. Just like theories of sets, there are theories of parthood that axiomatize various conceptions (sometimes conflicting) of this relation.

The language of mereology contains a single two-place predicate symbol P, and $P(x, y)$ "means" that x is a part of y. When we have this interpretation in mind, a structure for this language is called a *parthood structure*. Of course, not every structure for a single two-place predicate will really deserve this name. To have a chance of capturing "parthood," $P^{\mathfrak{M}}$ must satisfy some conditions, which we can lay down as axioms for a theory of parthood. For instance, parthood is a partial order on objects: every object is a part (albeit an *improper* part) of itself; no two different objects can be parts of each other; a part of a part of an object is itself part of that object. Note that in this sense "is a part of" resembles "is a subset of," but does not resemble "is an element of" which is neither reflexive nor transitive.

$$\begin{aligned}
&\forall x \, P(x, x), \\
&\forall x \, \forall y \, ((P(x, y) \wedge P(y, x)) \rightarrow x = y), \\
&\forall x \, \forall y \, \forall z \, ((P(x, y) \wedge P(y, z)) \rightarrow P(x, z)),
\end{aligned}$$

Moreover, any two objects have a fusion (an object that has only these two objects and all their parts as parts).

$$\forall x \, \forall y \, \exists z \, \forall u \, (P(u, z) \leftrightarrow (P(u, x) \wedge P(u, y)))$$

These are only some of the basic principles of parthood considered by metaphysicians. Further principles, however, quickly become hard to formulate or write down without first introducting some defined relations. For instance, most metaphysicians interested in mereology also view the following as a valid principle: whenever an object x has a proper part y, it also has a part z that has no parts in common with y, and so that the fusion of y and z is x.

2.4 Expressing Relations in a Structure

One main use formulas can be put to is to express properties and relations in a structure $\mathfrak{M}$ in terms of the primitives of the language $\mathcal{L}$ of $\mathfrak{M}$. By this we mean the following: the domain of $\mathfrak{M}$ is a set of objects. The constant symbols, function symbols, and predicate symbols are interpreted in $\mathfrak{M}$ by some objects in$|\mathfrak{M}|$, functions on $|\mathfrak{M}|$, and relations on $|\mathfrak{M}|$. For instance, if A^2_0 is in $\mathcal{L}$, then $\mathfrak{M}$ assigns to it a relation $R = A^{2\mathfrak{M}}_0$. Then the formula $A^2_0(x_1, x_2)$ *expresses* that very relation, in the following sense: if a variable assignment s maps x_1 to $a \in |\mathfrak{M}|$ and x_2 to $b \in |\mathfrak{M}|$, then

$$Rab \quad \text{iff} \quad \mathfrak{M}, s \models A^2_0(x_1, x_2).$$

Note that we have to involve variable assignments here: we can't just say "Rab iff $\mathfrak{M} \models A^2_0(a, b)$" because a and b are not symbols of our language: they are elements of $|\mathfrak{M}|$.

Since we don't just have atomic formulas, but can combine them using the logical connectives and the quantifiers, more complex formulas can define other relations which aren't directly built into $\mathfrak{M}$. We're interested in how to do that, and specifically, which relations we can define in a structure.

Definition 2.9. Let $\varphi(x_1, \ldots, x_n)$ be a formula of $\mathcal{L}$ in which only $x_1, \ldots, x_n$ occur free, and let $\mathfrak{M}$ be a structure for $\mathcal{L}$. $\varphi(x_1, \ldots, x_n)$ *expresses the relation* $R \subseteq |\mathfrak{M}|^n$ iff

$$Ra_1 \ldots a_n \quad \text{iff} \quad \mathfrak{M}, s \models \varphi(x_1, \ldots, x_n)$$

for any variable assignment s with $s(x_i) = a_i$ $(i = 1, \ldots, n)$.

Example 2.10. In the standard model of arithmetic $\mathfrak{N}$, the formula $x_1 < x_2 \vee x_1 = x_2$ expresses the $\leq$ relation on $\mathbb{N}$. The formula $x_2 = x_1'$ expresses the successor relation, i.e., the relation $R \subseteq \mathbb{N}^2$ where Rnm holds if m is the successor of n. The formula $x_1 = x_2'$ expresses the predecessor relation. The formulas $\exists x_3\,(x_3 \neq 0 \wedge x_2 = (x_1 + x_3))$ and $\exists x_3\,(x_1 + x_3') = x_2$ both express the $<$ relation. This means that the predicate symbol $<$ is actually superfluous in the language of arithmetic; it can be defined.

This idea is not just interesting in specific structures, but generally whenever we use a language to describe an intended model or models, i.e., when we consider theories. These theories often only contain a few predicate symbols as basic symbols, but in the domain they are used to describe often many other relations play an important role. If these other relations can be systematically expressed by the relations that interpret the basic predicate symbols of the language, we say we can *define* them in the language.

2.5 The Theory of Sets

Almost all of mathematics can be developed in the theory of sets. Developing mathematics in this theory involves a number of things. First, it requires a set of axioms for the relation $\in$. A number of different axiom systems have been developed, sometimes with conflicting properties of $\in$. The axiom system known as **ZFC**, Zermelo-Fraenkel set theory with the axiom of choice stands out: it is by far the most widely used and studied, because it turns out that its axioms suffice to prove almost all the things mathematicians expect to be able to prove. But before that can be established, it first is necessary to make clear how we can even *express* all the things mathematicians would like to express. For starters, the language contains no constant symbols or function symbols, so it seems at first glance unclear that we can talk about particular sets (such as $\emptyset$ or $\mathbb{N}$), can talk about operations on sets (such as $X \cup Y$ and $\wp(X)$), let alone other constructions which involve things other than sets, such as relations and functions.

To begin with, "is an element of" is not the only relation we are interested in: "is a subset of" seems almost as important. But we can *define* "is a subset of" in terms of "is an element of." To do this, we have to find a formula $\varphi(x, y)$ in the language of set theory which is satisfied by a pair of sets $\langle X, Y \rangle$ iff $X \subseteq Y$. But X is a subset of Y just in case all elements of X are also elements of Y. So we can define $\subseteq$ by the formula

$$\forall z\,(z \in x \to z \in y)$$

Now, whenever we want to use the relation $\subseteq$ in a formula, we could instead use that formula (with x and y suitably replaced, and the bound variable z renamed if necessary). For instance, extensionality of sets means that if any sets x and y are contained in each other, then x and y must be the same set. This can be expressed by $\forall x\,\forall y\,((x \subseteq y \wedge y \subseteq x) \to x = y)$, or, if we replace $\subseteq$ by the above definition, by

$$\forall x\,\forall y\,((\forall z\,(z \in x \to z \in y) \wedge \forall z\,(z \in y \to z \in x)) \to x = y).$$

This is in fact one of the axioms of **ZFC**, the "axiom of extensionality."

There is no constant symbol for $\emptyset$, but we can express "x is empty" by $\neg\exists y\, y \in x$. Then "$\emptyset$ exists" becomes the sentence $\exists x\,\neg\exists y\, y \in x$. This is another axiom of **ZFC**. (Note that the axiom of extensionality implies that there is only one empty set.) Whenever we want to talk about $\emptyset$ in the language of set theory, we would write this as "there is a set that's empty and ..." As an example, to express the fact that $\emptyset$ is a subset of every set, we could write

$$\exists x\,(\neg\exists y\, y \in x \wedge \forall z\, x \subseteq z)$$

where, of course, $x \subseteq z$ would in turn have to be replaced by its definition.

To talk about operations on sets, such has $X \cup Y$ and $\wp(X)$, we have to use a similar trick. There are no function symbols in the language of set theory, but we can express the functional relations $X \cup Y = Z$ and $\wp(X) = Y$ by

$$\forall u\,((u \in x \vee u \in y) \leftrightarrow u \in z)$$
$$\forall u\,(u \subseteq x \leftrightarrow u \in y)$$

since the elements of $X \cup Y$ are exactly the sets that are either elements of X or elements of Y, and the elements of $\wp(X)$ are exactly the subsets of X. However, this doesn't allow us to use $x \cup y$ or $\wp(x)$ as if they were terms: we can only use the entire formulas that define the relations $X \cup Y = Z$ and $\wp(X) = Y$. In fact, we do not know that these relations are ever satisfied, i.e., we do not know that unions and power sets always exist. For instance, the sentence $\forall x\,\exists y\,\wp(x) = y$ is another axiom of **ZFC** (the power set axiom).

Now what about talk of ordered pairs or functions? Here we have to explain how we can think of ordered pairs and functions as special kinds of sets. One way to define the ordered pair $\langle x, y\rangle$ is as the set $\{\{x\}, \{x, y\}\}$. But like before, we cannot introduce a function symbol that names this set; we can only define the relation $\langle x, y\rangle = z$, i.e., $\{\{x\}, \{x, y\}\} = z$:

$$\forall u\,(u \in z \leftrightarrow (\forall v\,(v \in u \leftrightarrow v = x) \vee \forall v\,(v \in u \leftrightarrow (v = x \vee v = y))))$$

This says that the elements u of z are exactly those sets which either have x as its only element or have x and y as its only elements (in other words, those sets that are either identical to $\{x\}$ or identical to $\{x, y\}$). Once we have this, we can say further things, e.g., that $X \times Y = Z$:

$$\forall z\,(z \in Z \leftrightarrow \exists x\,\exists y\,(x \in X \wedge y \in Y \wedge \langle x, y\rangle = z))$$

A function $f\colon X \to Y$ can be thought of as the relation $f(x) = y$, i.e., as the set of pairs $\{\langle x, y\rangle : f(x) = y\}$. We can then say that a set f is a function from X to Y if (a) it is a relation $\subseteq X \times Y$, (b) it is total, i.e., for all $x \in X$ there is some $y \in Y$ such that $\langle x, y\rangle \in f$ and (c) it is functional, i.e., whenever $\langle x, y\rangle, \langle x, y'\rangle \in f$, $y = y'$ (because values of functions must be unique). So "f is a function from X to Y" can be written as:

$$\begin{aligned}&\forall u\,(u \in f \to \exists x\,\exists y\,(x \in X \wedge y \in Y \wedge \langle x, y\rangle = u)) \wedge\\ &\forall x\,(x \in X \to (\exists y\,(y \in Y \wedge \mathrm{maps}(f, x, y)) \wedge\\ &\qquad\qquad (\forall y\,\forall y'\,((\mathrm{maps}(f, x, y) \wedge \mathrm{maps}(f, x, y')) \to y = y')))\end{aligned}$$

where $\mathrm{maps}(f, x, y)$ abbreviates $\exists v\,(v \in f \wedge \langle x, y\rangle = v)$ (this formula expresses "$f(x) = y$").

It is now also not hard to express that $f\colon X \to Y$ is injective, for instance:

$$\begin{aligned}&f\colon X \to Y \wedge\\ &\forall x\,\forall x'\,((x \in X \wedge x' \in X \wedge \exists y\,(\mathrm{maps}(f, x, y) \wedge \mathrm{maps}(f, x', y))) \to x = x')\end{aligned}$$

A function $f\colon X \to Y$ is injective iff, whenever f maps $x, x' \in X$ to a single y, $x = x'$. If we abbreviate this formula as $\mathrm{inj}(f, X, Y)$, we're already in a position to state in the language of set theory something as non-trivial as Cantor's theorem: there is no injective function from $\wp(X)$ to X:

$$\forall X\, \forall Y\, (\wp(X) = Y \to \neg \exists f\, \mathrm{inj}(f, Y, X))$$

2.6 Expressing the Size of Structures

There are some properties of structures we can express even without using the non-logical symbols of a language. For instance, there are sentences which are true in a structure iff the domain of the structure has at least, at most, or exactly a certain number n of elements.

Proposition 2.11. *The sentence*

$$\begin{aligned}\varphi_{\geq n} \equiv \exists x_1\, \exists x_2\, \ldots \exists x_n \quad (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \wedge \cdots \wedge x_1 \neq x_n \wedge \\ x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge \cdots \wedge x_2 \neq x_n \wedge \\ \vdots \\ x_{n-1} \neq x_n)\end{aligned}$$

is true in a structure $\mathfrak{M}$ iff $|\mathfrak{M}|$ contains at least n elements. Consequently, $\mathfrak{M} \models \neg\varphi_{\geq n+1}$ iff $|\mathfrak{M}|$ contains at most n elements.

Proposition 2.12. *The sentence*

$$\begin{aligned}\varphi_{=n} \equiv \exists x_1\, \exists x_2\, \ldots \exists x_n \quad (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \wedge \cdots \wedge x_1 \neq x_n \wedge \\ x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge \cdots \wedge x_2 \neq x_n \wedge \\ \vdots \\ x_{n-1} \neq x_n \wedge \\ \forall y\, (y = x_1 \vee \ldots y = x_n) \ldots))\end{aligned}$$

is true in a structure $\mathfrak{M}$ iff $|\mathfrak{M}|$ contains exactly n elements.

Proposition 2.13. *A structure is infinite iff it is a model of*

$$\{\varphi_{\geq 1}, \varphi_{\geq 2}, \varphi_{\geq 3}, \ldots\}$$

There is no single purely logical sentence which is true in $\mathfrak{M}$ iff $|\mathfrak{M}|$ is infinite. However, one can give sentences with non-logical predicate symbols which only have infinite models (although not every infinite structure is a model of them). The property of being a finite structure, and the property of being a non-enumerable structure cannot even be expressed with an infinite set of sentences. These facts follow from the compactness and Löwenheim-Skolem theorems.

Chapter 3

The Sequent Calculus

3.1 Rules and Derivations

Let $\mathcal{L}$ be a first-order language with the usual constants, variables, logical symbols, and auxiliary symbols (parentheses and the comma).

Definition 3.1 (sequent)**.** A *sequent* is an expression of the form

$$\Gamma \Rightarrow \Delta$$

where Γ and Δ are finite (possibly empty) sets of sentences of the language $\mathcal{L}$. The formulas in Γ are the *antecedent formulas*, while the formulae in Δ are the *succedent formulas*.

The intuitive idea behind a sequent is: if all of the antecedent formulas hold, then at least one of the succedent formulas holds. That is, if $\Gamma = \{\Gamma_1, \ldots, \Gamma_m\}$ and $\Delta = \{\Delta_1, \ldots, \Delta_n\}$, then $\Gamma \Rightarrow \Delta$ holds iff

$$(\Gamma_1 \wedge \cdots \wedge \Gamma_m) \rightarrow (\Delta_1 \vee \cdots \vee \Delta_n)$$

holds.

When $m = 0$, $\quad \Rightarrow \Delta$ holds iff $\Delta_1 \vee \cdots \vee \Delta_n$ holds. When $n = 0$, $\Gamma \Rightarrow$ holds iff $\Gamma_1 \wedge \cdots \wedge \Gamma_m$ does not. An empty succedent is sometimes filled with the $\bot$ symbol. The empty sequent $\quad \Rightarrow \quad$ canonically represents a contradiction.

We write Γ, φ (or φ, Γ) for $\Gamma \cup \{\varphi\}$, and Γ, Δ for $\Gamma \cup \Delta$.

Definition 3.2 (Inference)**.** An *inference* is an expression of the form

$$\frac{S_1}{S} \quad \text{or} \quad \frac{S_1 \qquad S_2}{S}$$

where S, S_1, and S_2 are sequents. S_1 and S_2 are called the *upper sequents* and S the *lower sequent* of the inference.

Inferences represent the idea that whenever the upper sequent(s) is (are) asserted, from it, we may logically infer the lower sequent.

For the following, let $\Gamma, \Delta, \Pi, \Lambda$ represent finite sets of sentences.

The rules for **LK** are divided into two main types: *structural* rules and *logical* rules. The logical rules are further divided into *propositional* rules (quantifier-free) and *quantifier* rules.

Structural rules: Weakening:

$$\frac{\Gamma \Rightarrow \Delta}{\varphi, \Gamma \Rightarrow \Delta} \quad \text{and} \quad \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \varphi}$$

where φ is called the *weakening formula*.

Cut:

$$\frac{\Gamma \Rightarrow \Delta, \varphi \qquad \varphi, \Pi \Rightarrow \Lambda}{\Gamma, \Pi \Rightarrow \Delta, \Lambda}$$

Logical rules: The rules are named by the main operator of the *principal formula* of the inference (the formula containing φ and/or ψ in the lower sequent). The designations "left" and "right" indicate whether the logical symbol has been introduced in an antecedent formula or a succedent formula (to the left or to the right of the sequent symbol).

Propositional Rules:

$$\frac{\Gamma \Rightarrow \Delta, \varphi}{\neg\varphi, \Gamma \Rightarrow \Delta} \neg \text{ left} \quad \frac{\varphi, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \neg\varphi} \neg \text{ right}$$

$$\frac{\varphi, \Gamma \Rightarrow \Delta}{\varphi \wedge \psi, \Gamma \Rightarrow \Delta} \wedge \text{ left} \quad \frac{\psi, \Gamma \Rightarrow \Delta}{\varphi \wedge \psi, \Gamma \Rightarrow \Delta} \wedge \text{ left} \quad \frac{\Gamma \Rightarrow \Delta, \varphi \qquad \Gamma \Rightarrow \Delta, \psi}{\Gamma \Rightarrow \Delta, \varphi \wedge \psi} \wedge \text{ right}$$

$$\frac{\varphi, \Gamma \Rightarrow \Delta \qquad \psi, \Gamma \Rightarrow \Delta}{\varphi \vee \psi, \Gamma \Rightarrow \Delta} \vee \text{ left} \quad \frac{\Gamma \Rightarrow \Delta, \varphi}{\Gamma \Rightarrow \Delta, \varphi \vee \psi} \vee \text{ right} \quad \frac{\Gamma \Rightarrow \Delta, \psi}{\Gamma \Rightarrow \Delta, \varphi \vee \psi} \vee \text{ right}$$

$$\frac{\Gamma \Rightarrow \Delta, \varphi \qquad \psi, \Pi \Rightarrow \Lambda}{\varphi \to \psi, \Gamma, \Pi \Rightarrow \Delta, \Lambda} \to \text{ left} \quad \frac{\varphi, \Gamma \Rightarrow \Delta, \psi}{\Gamma \Rightarrow \Delta, \varphi \to \psi} \to \text{ right}$$

Quantifier Rules:

$$\frac{\varphi(t), \Gamma \Rightarrow \Delta}{\forall x\, \varphi(x), \Gamma \Rightarrow \Delta} \forall \text{ left} \quad \frac{\Gamma \Rightarrow \Delta, \varphi(a)}{\Gamma \Rightarrow \Delta, \forall x\, \varphi(x)} \forall \text{ right}$$

where t is a ground term (i.e., one without variables), and a is a constant which does not occur anywhere in the lower sequent of the $\forall$ right rule. We call a the *eigenvariable* of the $\forall$ right inference.

$$\frac{\varphi(a), \Gamma \Rightarrow \Delta}{\exists x\, \varphi(x), \Gamma \Rightarrow \Delta} \exists \text{ left} \quad \frac{\Gamma \Rightarrow \Delta, \varphi(t)}{\Gamma \Rightarrow \Delta, \exists x\, \varphi(x)} \exists \text{ right}$$

where t is a ground term, and a is a constant which does not occur in the lower sequent of the $\exists$ left rule. We call a the *eigenvariable* of the $\exists$ left inference.

The condition that an eigenvariable not occur in the upper sequent of the $\forall$ right or $\exists$ left inference is called the *eigenvariable condition*.

We use the term "eigenvariable" even though a in the above rules is a constant. This has historical reasons.

In $\exists$ right and $\forall$ left there are no restrictions, and the term t can be anything, so we do not have to worry about any conditions. However, because the t may appear elsewhere in the sequent, the values of t for which the sequent is satisfied are constrained. On the other hand, in the $\exists$ left and $\forall$ right rules, the eigenvariable condition requires that a does not occur anywhere else in the sequent. Thus, if the upper sequent is valid, the truth values of the formulas other than $\varphi(a)$ are independent of a.

Definition 3.3 (Initial Sequent)**.** An *initial sequent* is a sequent of the form $\varphi \Rightarrow \varphi$ for any sentence φ in the language.

Definition 3.4 (LK derivation)**.** An **LK**-*derivation* of a sequent S is a tree of sequents satisfying the following conditions:

1. The topmost sequents of the tree are initial sequents.
2. Every sequent in the tree (except S) is an upper sequent of an inference whose lower sequent stands directly below that sequent in the tree.

We then say that S is the *end-sequent* of the derivation and that S is *derivable in* **LK** (or **LK**-derivable).

Definition 3.5 (LK theorem)**.** A sentence φ is a *theorem* of **LK** if the sequent $\Rightarrow \varphi$ is **LK**-derivable.

3.2 Examples of Derivations

Example 3.6. Give an **LK**-derivation for the sequent $\varphi \wedge \psi \Rightarrow \varphi$.

We begin by writing the desired end-sequent at the bottom of the derivation.

$$\frac{}{\varphi \wedge \psi \Rightarrow \varphi}$$

Next, we need to figure out what kind of inference could have a lower sequent of this form. This could be a structural rule, but it is a good idea to start by looking for a logical rule. The only logical connective occurring in a formula in the lower sequent is $\wedge$, so we're looking for an $\wedge$ rule, and since the $\wedge$ symbol occurs in the antecedent formulas, we're looking at the $\wedge$ left rule.

$$\frac{}{\varphi \wedge \psi \Rightarrow \varphi}\ \wedge \text{left}$$

There are two options for what could have been the upper sequent of the $\wedge$ left inference: we could have an upper sequent of $\varphi \Rightarrow \varphi$, or of $\psi \Rightarrow \varphi$. Clearly, $\varphi \Rightarrow \varphi$ is an initial sequent (which is a good thing), while $\psi \Rightarrow \varphi$ is not derivable in general. We fill in the upper sequent:

$$\dfrac{\varphi \Rightarrow \varphi}{\varphi \wedge \psi \Rightarrow \varphi} \wedge \text{left}$$

We now have a correct **LK**-derivation of the sequent $\varphi \wedge \psi \Rightarrow \varphi$.

Example 3.7. Give an **LK**-derivation for the sequent $\neg\varphi \vee \psi \Rightarrow \varphi \to \psi$.

Begin by writing the desired end-sequent at the bottom of the derivation.

$$\dfrac{}{\neg\varphi \vee \psi \Rightarrow \varphi \to \psi}$$

To find a logical rule that could give us this end-sequent, we look at the logical connectives in the end-sequent: $\neg$, $\vee$, and $\to$. We only care at the moment about $\vee$ and $\to$ because they are main operators of sentences in the end-sequent, while $\neg$ is inside the scope of another connective, so we will take care of it later. Our options for logical rules for the final inference are therefore the $\vee$ left rule and the $\to$ right rule. We could pick either rule, really, but let's pick the $\to$ right rule (if for no reason other than it allows us to put off splitting into two branches). According to the form of $\to$ right inferences which can yield the lower sequent, this must look like:

$$\dfrac{\dfrac{}{\varphi, \neg\varphi \vee \psi \Rightarrow \psi}}{\neg\varphi \vee \psi \Rightarrow \varphi \to \psi} \to \text{right}$$

Now we can apply the $\vee$ left rule. According to the schema, this must split into two upper sequents as follows:

$$\dfrac{\dfrac{\dfrac{}{\varphi, \neg\varphi \Rightarrow \psi} \quad \dfrac{}{\varphi, \psi \Rightarrow \psi}}{\varphi, \neg\varphi \vee \psi \Rightarrow \psi} \vee \text{left}}{\neg\varphi \vee \psi \Rightarrow \varphi \to \psi} \to \text{right}$$

Remember that we are trying to wind our way up to initial sequents; we seem to be pretty close! The right branch is just one weakening away from an initial sequent and then it is done:

$$\dfrac{\dfrac{\dfrac{}{\varphi, \neg\varphi \Rightarrow \psi} \quad \dfrac{\psi \Rightarrow \psi}{\varphi, \psi \Rightarrow \psi}}{\varphi, \neg\varphi \vee \psi \Rightarrow \psi} \vee \text{left}}{\neg\varphi \vee \psi \Rightarrow \varphi \to \psi} \to \text{right}$$

The double line indicates a weakening inference has been left out.

Now looking at the left branch, the only logical connective in any sentence is the $\neg$ symbol in the antecedent sentences, so we're looking at an instance of the $\neg$ left rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \Rightarrow \psi, \varphi}}{\varphi, \neg\varphi \Rightarrow \psi}\neg\text{ left} \quad \dfrac{\dfrac{\psi \Rightarrow \psi}{\varphi, \psi \Rightarrow \psi}}{}}{\varphi, \neg\varphi \vee \psi \Rightarrow \psi}\vee\text{ left}}{\neg\varphi \vee \psi \Rightarrow \varphi \to \psi}\to\text{ right}$$

Similarly to how we finished off the right branch, we are just one weakening away from finishing off this left branch as well.

$$\dfrac{\dfrac{\dfrac{\dfrac{\varphi \Rightarrow \varphi}{\varphi \Rightarrow \psi, \varphi}}{\varphi, \neg\varphi \Rightarrow \psi}\neg\text{ left} \quad \dfrac{\dfrac{\psi \Rightarrow \psi}{\varphi, \psi \Rightarrow \psi}}{}}{\varphi, \neg\varphi \vee \psi \Rightarrow \psi}\vee\text{ left}}{\neg\varphi \vee \psi \Rightarrow \varphi \to \psi}\to\text{ right}$$

Example 3.8. Give an **LK**-derivation of the sequent $\neg\varphi \vee \neg\psi \Rightarrow \neg(\varphi \wedge \psi)$

Using the techniques from above, we start by writing the desired end-sequent at the bottom.

$$\dfrac{}{\neg\varphi \vee \neg\psi \Rightarrow \neg(\varphi \wedge \psi)}$$

The available main connectives of sentences in the end-sequent are the $\vee$ symbol and the $\neg$ symbol. It would work to apply either the $\vee$ left or the $\neg$ right rule here, but we start with the $\neg$ right rule because it avoids splitting up into two branches for a moment:

$$\dfrac{\dfrac{}{\varphi \wedge \psi, \neg\varphi \vee \neg\psi \Rightarrow}}{\neg\varphi \vee \neg\psi \Rightarrow \neg(\varphi \wedge \psi)}\neg\text{ right}$$

Now we have a choice of whether to look at the $\wedge$ left or the $\vee$ left rule. Let's see what happens when we apply the $\wedge$ left rule: we have a choice to start with either the sequent $\varphi, \neg\varphi \vee \psi \Rightarrow$ or the sequent $\psi, \neg\varphi \vee \psi \Rightarrow$. Since the proof is symmetric with regards to φ and ψ, let's go with the former:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi, \neg\varphi \vee \neg\psi \Rightarrow}}{\varphi \wedge \psi, \neg\varphi \vee \neg\psi \Rightarrow}\wedge\text{ left}}{\neg\varphi \vee \neg\psi \Rightarrow \neg(\varphi \wedge \psi)}\neg\text{ right}$$

Continuing to fill in the derivation, we see that we run into a problem:

$$\dfrac{\dfrac{\dfrac{\dfrac{\varphi \Rightarrow \varphi}{\varphi, \neg\varphi \Rightarrow}\neg\text{ left} \quad \dfrac{\dfrac{}{\varphi \Rightarrow \psi}?}{\varphi, \neg\psi \Rightarrow}\neg\text{ left}}{\varphi, \neg\varphi \vee \neg\psi \Rightarrow}\vee\text{ left}}{\varphi \wedge \psi, \neg\varphi \vee \neg\psi \Rightarrow}\wedge\text{ left}}{\neg\varphi \vee \neg\psi \Rightarrow \neg(\varphi \wedge \psi)}\neg\text{ right}$$

The top of the right branch cannot be reduced any further, and it cannot be brought by way of structural inferences to an initial sequent, so this is not the right path to take. So clearly, it was a mistake to apply the $\wedge$ left rule above. Going back to what we had before and carrying out the $\vee$ left rule instead, we get

$$\dfrac{\dfrac{\dfrac{\varphi \wedge \psi, \neg\varphi \Rightarrow}{\varphi \wedge \psi, \neg\varphi \vee \neg\psi \Rightarrow}\vee\text{ left} \quad \dfrac{\varphi \wedge \psi, \neg\psi \Rightarrow}{\varphi \wedge \psi, \neg\varphi \vee \neg\psi \Rightarrow}\vee\text{ left}}{\neg\varphi \vee \neg\psi \Rightarrow \neg(\varphi \wedge \psi)}\neg\text{ right}$$

Completing each branch as we've done before, we get

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\varphi \Rightarrow \varphi}{\varphi \wedge \psi \Rightarrow \varphi}\wedge\text{ left}}{\varphi \wedge \psi, \neg\varphi \Rightarrow}\neg\text{ left} \quad \dfrac{\dfrac{\dfrac{\psi \Rightarrow \psi}{\varphi \wedge \psi \Rightarrow \psi}\wedge\text{ left}}{\varphi \wedge \psi, \neg\psi \Rightarrow}\neg\text{ left}}{\varphi \wedge \psi, \neg\varphi \vee \neg\psi \Rightarrow}\vee\text{ left}}{\neg\varphi \vee \neg\psi \Rightarrow \neg(\varphi \wedge \psi)}\neg\text{ right}$$

(We could have carried out the $\wedge$ rules lower than the $\neg$ rules in these steps and still obtained a correct derivation).

Example 3.9. Give an **LK**-derivation of the sequent $\exists x\, \neg\varphi(x) \Rightarrow \neg\forall x\, \varphi(x)$.

When dealing with quantifiers, we have to make sure not to violate the eigenvariable condition, and sometimes this requires us to play around with the order of carrying out certain inferences. In general, it helps to try and take care of rules subject to the eigenvariable condition first (they will be lower down in the finished proof). Also, it is a good idea to try and look ahead and try to guess what the initial sequent might look like. In our case, it will have to be something like $\varphi(a) \Rightarrow \varphi(a)$. That means that when we are "reversing" the quantifier rules, we will have to pick the same term—what we will call a—for both the $\forall$ and the $\exists$ rule. If we picked different terms for each rule, we would end up with something like $\varphi(a) \Rightarrow \varphi(b)$, which, of course, is not derivable.

Starting as usual, we write

$$\dfrac{}{\exists x\, \neg\varphi(x) \Rightarrow \neg\forall x\, \varphi(x)}$$

We could either carry out the $\exists$ left rule or the $\neg$ right rule. Since the $\exists$ left rule is subject to the eigenvariable condition, it's a good idea to take care of it sooner rather than later, so we'll do that one first.

$$\dfrac{\dfrac{}{\neg\varphi(a) \Rightarrow \neg\forall x\, \varphi(x)}}{\exists x\, \neg\varphi(x) \Rightarrow \neg\forall x\, \varphi(x)}\,\exists \text{ left}$$

Applying the $\neg$ left and right rules to eliminate the $\neg$ signs, we get

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x\, \varphi(x) \Rightarrow \varphi(a)}}{\Rightarrow \neg\forall x\, \varphi(x), \varphi(a)}\,\neg \text{ right}}{\neg\varphi(a) \Rightarrow \neg\forall x\varphi(x)}\,\neg \text{ left}}{\exists x\neg\varphi(x) \Rightarrow \neg\forall x\varphi(x)}\,\exists \text{ left}$$

At this point, our only option is to carry out the $\forall$ left rule. Since this rule is not subject to the eigenvariable restriction, we're in the clear. Remember, we want to try and obtain an initial sequent (of the form $\varphi(a) \Rightarrow \varphi(a)$), so we should choose a as our argument for φ when we apply the rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\varphi(a) \Rightarrow \varphi(a)}{\forall x\, \varphi(x) \Rightarrow \varphi(a)}\,\forall \text{ left}}{\Rightarrow \neg\forall x\, \varphi(x), \varphi(a)}\,\neg \text{ right}}{\neg\varphi(a) \Rightarrow \neg\forall x\, \varphi(x)}\,\neg \text{ left}}{\exists x\, \neg\varphi(x) \Rightarrow \neg\forall x\, \varphi(x)}\,\exists \text{ left}$$

It is important, especially when dealing with quantifiers, to double check at this point that the eigenvariable condition has not been violated. Since the only rule we applied that is subject to the eigenvariable condition was $\exists$ left, and the eigenvariable a does not occur in its lower sequent (the end-sequent), this is a correct derivation.

3.3 Proof-Theoretic Notions

Just as we've defined a number of important semantic notions (validity, entailment, satisfiabilty), we now define corresponding *proof-theoretic notions*. These are not defined by appeal to satisfaction of sentences in structures, but by appeal to the derivability or non-derivability of certain sequents. It was an important discovery, due to Gödel, that these notions coincide. That they do is the content of the *completeness theorem*.

Definition 3.10 (Theorems)**.** A sentence φ is a *theorem* if there is a derivation in **LK** of the sequent $\Rightarrow \varphi$. We write $\vdash_{\mathbf{LK}} \varphi$ if φ is a theorem and $\nvdash_{\mathbf{LK}} \varphi$ if it is not.

Definition 3.11 (Derivability)**.** A sentence φ is *derivable from* a set of sentences Γ, $\Gamma \vdash_{\mathbf{LK}} \varphi$, if there is a finite subset $\Gamma_0 \subseteq \Gamma$ such that **LK** derives $\Gamma_0 \Rightarrow \varphi$. If φ is not derivable from Γ we write $\Gamma \nvdash_{\mathbf{LK}} \varphi$.

Definition 3.12 (Consistency)**.** A set of sentences Γ is *consistent* iff $\Gamma \nvdash_{\mathbf{LK}} \bot$. If Γ is not consistent, i.e., if $\Gamma \vdash_{\mathbf{LK}} \bot$, we say it is *inconsistent*.

Proposition 3.13. *$\Gamma \vdash_{\mathbf{LK}} \varphi$ iff $\Gamma \cup \{\neg\varphi\}$ is inconsistent.*

Proof. Exercise. □

Proposition 3.14. *Γ is inconsistent iff $\Gamma \vdash_{\mathbf{LK}} \varphi$ for every sentence φ.*

Proof. Exercise. □

Proposition 3.15. *If $\Gamma \vdash \varphi$ iff for some finite $\Gamma_0 \subseteq \Gamma$, $\Gamma_0 \vdash \varphi$.*

Proof. Follows immediately from the definion of $\vdash$. □

3.4 Properties of Derivability

Proposition 3.16 (Monotony)**.** *If $\Gamma \subseteq \Delta$ and $\Gamma \vdash \varphi$, then $\Delta \vdash \varphi$.*

Proof. Any finite $\Gamma_0 \subseteq \Gamma$ is also a finite subset of Δ, so a derivation of $\Gamma_0 \Rightarrow \varphi$ also shows $\Delta \vdash \varphi$. □

Proposition 3.17. 1. *If $\Gamma \vdash_{\mathbf{LK}} \varphi$ and $\Gamma \cup \{\varphi\} \vdash_{\mathbf{LK}} \bot$, then Γ is inconsistent.*

2. *If $\Gamma \cup \{\varphi\} \vdash_{\mathbf{LK}} \bot$, then $\Gamma \vdash_{\mathbf{LK}} \neg\varphi$.*

3. *If $\Gamma \cup \{\varphi\} \vdash_{\mathbf{LK}} \bot$ and $\Gamma \cup \{\neg\varphi\} \vdash_{\mathbf{LK}} \bot$, then $\Gamma \vdash_{\mathbf{LK}} \bot$.*

4. *If $\Gamma \cup \{\varphi\} \vdash_{\mathbf{LK}} \bot$ and $\Gamma \cup \{\psi\} \vdash_{\mathbf{LK}} \bot$, then $\Gamma \cup \{\varphi \vee \psi\} \vdash_{\mathbf{LK}} \bot$.*

5. *If $\Gamma \vdash_{\mathbf{LK}} \varphi$ or $\Gamma \vdash_{\mathbf{LK}} \psi$, then $\Gamma \vdash_{\mathbf{LK}} \varphi \vee \psi$.*

6. *If $\Gamma \vdash_{\mathbf{LK}} \varphi \wedge \psi$ then $\Gamma \vdash_{\mathbf{LK}} \varphi$ and $\Gamma \vdash_{\mathbf{LK}} \psi$.*

7. *If $\Gamma \vdash_{\mathbf{LK}} \varphi$ and $\Gamma \vdash_{\mathbf{LK}} \psi$, then $\Gamma \vdash_{\mathbf{LK}} \varphi \wedge \psi$.*

8. *If $\Gamma \vdash_{\mathbf{LK}} \varphi$ and $\Gamma \vdash_{\mathbf{LK}} \varphi \to \psi$, then $\Gamma \vdash_{\mathbf{LK}} \psi$.*

9. *If $\Gamma \vdash_{\mathbf{LK}} \neg\varphi$ or $\Gamma \vdash_{\mathbf{LK}} \psi$, then $\Gamma \vdash_{\mathbf{LK}} \varphi \to \psi$.*

Proof. 1. Let the **LK**-derivation of $\Gamma_0 \Rightarrow \varphi$ be Π_0 and the **LK**-derivation of $\Gamma_1 \cup \{\varphi\} \Rightarrow \bot$ be Π_1. We can then derive

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Pi_0\end{array}}{\Gamma_0 \Rightarrow \varphi}\qquad \dfrac{\begin{array}{c}\vdots\\ \Pi_1\end{array}}{\Gamma_1, \varphi \Rightarrow \bot}}{\Gamma_0, \Gamma_1 \Rightarrow \bot}\,\text{cut}$$

Since $\Gamma_0 \subseteq \Gamma$ and $\Gamma_1 \subseteq \Gamma$, $\Gamma_0 \cup \Gamma_1 \subseteq \Gamma$, hence $\Gamma \vdash_{\mathbf{LK}} \bot$.

2. Suppose that $\Gamma \cup \{\varphi\} \vdash_{\mathbf{LK}} \bot$. Then there is a finite set $\Gamma_0 \subseteq \Gamma$ with $\vdash_{\mathbf{LK}} \Gamma_0 \cup \{\varphi\} \Rightarrow \bot$. Let Π_0 be an **LK**-derivation of $\Gamma_0 \cup \{\varphi\} \Rightarrow \bot$, and consider

$$\dfrac{\begin{array}{c}\vdots\\ \Pi_0\end{array}}{\Gamma_0, \varphi \Rightarrow \bot}\,\neg\,\text{right}$$

3. There are finite sets $\Gamma_0 \subseteq \Gamma$ and $\Gamma_1 \subseteq \Gamma$ and **LK**-derivations Π_0 and Π_1 of $\Gamma_0, \varphi \Rightarrow \bot$ and $\Gamma_1, \neg\varphi \Rightarrow \bot$, respectively. We can then derive

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Pi_0\end{array}}{\dfrac{\Gamma_0, \varphi \Rightarrow \bot}{\Gamma_0 \Rightarrow \neg\varphi}\,\neg\,\text{right}}\qquad \dfrac{\begin{array}{c}\vdots\\ \Pi_1\end{array}}{\Gamma_1, \neg\varphi \Rightarrow \bot}}{\Gamma_0, \Gamma_1 \Rightarrow \bot}\,\text{cut}$$

Since $\Gamma_0 \subseteq \Gamma$ and $\Gamma_1 \subseteq \Gamma$, $\Gamma_0 \cup \Gamma_1 \subseteq \Gamma$. Hence $\Gamma \vdash_{\mathbf{LK}} \bot$.

4. There are finite sets $\Gamma_0, \Gamma_1 \subseteq \Gamma$ and **LK**-derivations Π_0 and Π_1 such that

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Pi_0\end{array}}{\Gamma_0, \varphi \Rightarrow \bot}\qquad \dfrac{\begin{array}{c}\vdots\\ \Pi_1\end{array}}{\Gamma_1, \psi \Rightarrow \bot}}{\Gamma_0, \Gamma_1, \varphi \vee \psi \Rightarrow \bot}\,\vee\,\text{left}$$

Since $\Gamma_0, \Gamma_1 \subseteq \Gamma$ and $\Gamma \cup \{\varphi \vee \psi\} \vdash \bot$.

5. There is an **LK**-derivation Π_0 and a finite set $\Gamma_0 \subseteq \Gamma$ such that we can derive

$$\dfrac{\begin{array}{c}\vdots\\ \Pi_0\\ \vdots\\ \Gamma_0 \Rightarrow \varphi\end{array}}{\Gamma_0 \Rightarrow \varphi \vee \psi} \vee \text{right}$$

   Therefore $\Gamma \vdash \varphi \vee \psi$. The proof for when $\Gamma \vdash_{\mathbf{LK}} \psi$ is similar.

6. If $\Gamma \vdash_{\mathbf{LK}} \varphi \wedge \psi$, there is a finite set $\Gamma_0 \subseteq \Gamma$ and an **LK**-derivation Π_0 of $\Gamma_0 \Rightarrow \varphi \wedge \psi$. Consider

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Pi_0\\ \vdots\\ \Gamma_0, \Rightarrow \varphi \wedge \psi\end{array} \qquad \dfrac{\varphi \Rightarrow \varphi}{\varphi \wedge \psi \Rightarrow \varphi} \wedge \text{left}}{\Gamma_0 \Rightarrow \varphi} \text{cut}$$

   Hence, $\Gamma \vdash_{\mathbf{LK}} \varphi$. A similar derivation starting with $\psi \Rightarrow \psi$ on the right side shows that $\Gamma \vdash_{\mathbf{LK}} \psi$.

7. If $\Gamma \vdash_{\mathbf{LK}} \varphi$ as well as $\Gamma \vdash_{\mathbf{LK}} \psi$, there are finite sets $\Gamma_0, \Gamma_1 \subseteq \Gamma$ and an **LK**-derivations Π_0 of $\Gamma_0 \Rightarrow \varphi$ and Π_1 of $\Gamma_1 \Rightarrow \varphi$. Consider

$$\dfrac{\begin{array}{c}\vdots\\ \Pi_0\\ \vdots\\ \Gamma_0, \Rightarrow \varphi\end{array} \qquad \begin{array}{c}\vdots\\ \Pi_1\\ \vdots\\ \Gamma_1, \Rightarrow \psi\end{array}}{\Gamma_0, \Gamma_1 \Rightarrow \varphi \wedge \psi} \wedge \text{right}$$

   Since $\Gamma_0 \cup \Gamma_1 \subseteq \Gamma$, we have $\Gamma \vdash_{\mathbf{LK}} \varphi \wedge \psi$.

8. Suppose that $\Gamma \vdash_{\mathbf{LK}} \varphi$ and $\Gamma \vdash_{\mathbf{LK}} \varphi \to \psi$. There are finite sets $\Gamma_0, \Gamma_1 \subseteq \Gamma$ such that there are **LK**-derivations Π_0 of $\Gamma_0 \Rightarrow \varphi$ and Π_1 of $\Gamma_1 \Rightarrow \varphi \to \psi$. Consider:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Pi_0\\ \vdots\\ \Gamma_1 \Rightarrow \varphi \to \psi\end{array}}{\Gamma_0, \Gamma_1, \Gamma_2 \Rightarrow \varphi \to \psi} \qquad \dfrac{\begin{array}{c}\vdots\\ \Pi_1\\ \vdots\\ \Gamma_0 \Rightarrow \varphi\end{array} \qquad \dfrac{\psi \Rightarrow \psi}{\Gamma_0, \psi \Rightarrow \psi}}{\Gamma_0, \varphi \to \psi \Rightarrow \psi} \to \text{left}}{\Gamma_0, \Gamma_1 \Rightarrow \psi} \text{cut}$$

Since $\Gamma_0 \cup \Gamma_1 \subseteq \Gamma$, this means that $\Gamma \vdash_{\mathbf{LK}} \psi$.

9. First suppose $\Gamma \vdash_{\mathbf{LK}} \neg\varphi$. Then for some finite $\Gamma_0 \subseteq \Gamma$ there is a **LK**-derivation of $\Gamma_0 \Rightarrow \neg\varphi$. The following derivation shows that $\Gamma \vdash_{\mathbf{LK}} \varphi \to \psi$:

$$\dfrac{\dfrac{\vdots\ \Pi_0}{\Gamma_0 \Rightarrow \neg\varphi} \qquad \dfrac{\dfrac{\dfrac{}{\varphi \Rightarrow \varphi}}{\neg\varphi, \varphi \Rightarrow}\,\neg\mathit{right}}{\varphi, \neg\varphi \Rightarrow \psi}}{\Gamma_0 \Rightarrow \varphi \to \psi}\,\to\text{right}\ \text{cut}$$

Now suppose $\Gamma \vdash_{\mathbf{LK}} \psi$. Then for some finite $\Gamma_0 \subseteq \Gamma$ there is a **LK**-derivation of $\Gamma_0 \Rightarrow \psi$. The following derivation shows that $\Gamma \vdash_{\mathbf{LK}} \varphi \to \psi$:

$$\dfrac{\dfrac{\vdots\ \Pi_0}{\Gamma_0 \Rightarrow \psi} \qquad \dfrac{\dfrac{}{\psi \Rightarrow \psi}}{\varphi, \psi \Rightarrow \psi}}{\Gamma_0 \Rightarrow \varphi \to \psi}\,\to\text{right}\ \text{cut}$$

□

Theorem 3.18. *If c is a constant not occurring in Γ or $\varphi(x)$ and $\Gamma \vdash \varphi(c)$, then $\Gamma \vdash \forall x\, \varphi(c)$.*

Proof. Let Π_0 be an **LK**-derivation of $\Gamma_0 \Rightarrow \varphi(c)$ for some finite $\Gamma_0 \subseteq \Gamma$. By adding a $\forall$ right inference, we obtain a proof of $\Gamma \Rightarrow \forall x\, \varphi(x)$, since c does not occur in Γ or $\varphi(x)$ and thus the eigenvariable condition is satisfied. □

Theorem 3.19. *1. If $\Gamma \vdash \varphi(t)$ then $\Gamma \vdash \exists x\, \varphi(x)$.*

2. If $\Gamma \vdash \forall x\, \varphi(x)$ then $\Gamma \vdash \varphi(t)$.

Proof. 1. Suppose $\Gamma \vdash \varphi(t)$. Then for some finite $\Gamma_0 \subseteq \Gamma$, **LK** derives $\Gamma_0 \Rightarrow \varphi(t)$. Add an $\exists$ right inference to get a derivation of $\Gamma_0 \Rightarrow \exists x\, \varphi(x)$.

2. Suppose $\Gamma \vdash \neg\varphi(t)$. Then there is a finite $\Gamma_0 \subseteq \Gamma$ and an **LK**-derivation Π of $\Gamma_0 \Rightarrow \forall x\, \varphi(x)$. Then

$$\dfrac{\dfrac{\vdots\ \Pi}{\Gamma_0 \Rightarrow \forall x\, \varphi(x)} \qquad \dfrac{\varphi(t) \Rightarrow \varphi(t)}{\forall x\, \varphi(x) \Rightarrow \varphi(t)}\,\forall\text{ left}}{\Gamma_0 \Rightarrow \varphi(t)}\ \text{cut}$$

shows that $\Gamma_0 \vdash \varphi(t)$.

□

3.5 Soundness

A derivation system, such as the sequent calculus, is *sound* if it cannot derive things that do not actually hold. Soundness is thus a kind of guaranteed safety property for derivation systems. Depending on which proof theoretic property is in question, we would like to know for instance, that

1. every derivable sentence is valid;
2. if a sentence is derivable from some others, it is also a consequence of them;
3. if a set of sentences is inconsistent, it is unsatisfiable.

These are important properties of a derivation system. If any of them do not hold, the derivation system is deficient—it would derive too much. Consequently, establishing the soundness of a derivation system is of the utmost importance.

Because all these proof-theoretic properties are defined via derivability in the sequent calculus of certain sequents, proving (1)–(3) above requires proving something about the semantic properties of derivable sequents. We will first define what it means for a sequent to be *valid*, and then show that every derivable sequent is valid. (1)–(3) then follow as corollaries from this result.

Definition 3.20. A structure $\mathfrak{M}$ *satisfies* a sequent $\Gamma \Rightarrow \Delta$ iff either $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma$ or $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta$.

A sequent is *valid* iff every structure $\mathfrak{M}$ satisfies it.

Theorem 3.21 (Soundness)**.** *If* **LK** *derives* $\Gamma \Rightarrow \Delta$*, then* $\Gamma \Rightarrow \Delta$ *is valid.*

Proof. Let Π be a derivation of $\Gamma \Rightarrow \Delta$. We proceed by induction on the number of inferences in Π.

If the number of inferences is 0, then Π consists only of an initial sequent. Every initial sequent $\varphi \Rightarrow \varphi$ is obviously valid, since for every $\mathfrak{M}$, either $\mathfrak{M} \not\models \varphi$ or $\mathfrak{M} \models \varphi$.

If the number of inferences is greater than 0, we distinguish cases according to the type of the lowermost inference. By induction hypothesis, we can assume that the premises of that inference are valid.

First, we consider the possible inferences with only one premise $\Gamma' \Rightarrow \Delta'$.

1. The last inference is a weakening. Then $\Gamma' \subseteq \Gamma$ and $\Delta = \Delta'$ if it's a weakening on the left, or $\Gamma = \Gamma'$ and $\Delta' \subseteq \Delta$ if it's a weaking on the right. In either case, $\Delta' \subseteq \Delta$ and $\Gamma' \subseteq \Gamma$. If $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma'$, then,

since $\Gamma' \subseteq \Gamma$, $\alpha \in \Gamma$ as well, and so $\mathfrak{M} \not\models \alpha$ for the same $\alpha \in \Gamma$. Similarly, if $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta'$, as $\alpha \in \Delta$, $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta$. Since $\Gamma' \Rightarrow \Delta'$ is valid, one of these cases obtains for every $\mathfrak{M}$. Consequently, $\Gamma \Rightarrow \Delta$ is valid.

2. The last inference is $\neg$ left: Then for some $\varphi \in \Delta'$, $\neg\varphi \in \Gamma$. Also, $\Gamma' \subseteq \Gamma$, and $\Delta' \setminus \{\varphi\} \subseteq \Delta$.

   If $\mathfrak{M} \models \varphi$, then $\mathfrak{M} \not\models \neg\varphi$, and since $\neg\varphi \in \Gamma$, $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$. Since $\Gamma' \Rightarrow \Delta'$ is valid, if $\mathfrak{M} \not\models \varphi$, then either $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma'$ or $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta'$ different from φ. Consequently, $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma$ (since $\Gamma' \subseteq \Gamma$) or $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta'$ different from φ (since $\Delta' \setminus \{\varphi\} \subseteq \Delta$).

3. The last inference is $\neg$ right: Exercise.

4. The last inference is $\wedge$ left: There are two variants: $\varphi \wedge \psi$ may be inferred on the left from φ or from ψ on the left side of the premise. In the first case, $\varphi \in \Gamma'$. Consider a structure $\mathfrak{M}$. Since $\Gamma' \Rightarrow \Delta'$ is valid, (a) $\mathfrak{M} \not\models \varphi$, (b) $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma' \setminus \{\varphi\}$, or (c) $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta'$. In case (a), $\mathfrak{M} \not\models \varphi \wedge \psi$. In case (b), there is an $\alpha \in \Gamma \setminus \{\varphi \wedge \psi\}$ such that $\mathfrak{M} \not\models \alpha$, since $\Gamma' \setminus \{\varphi\} \subseteq \Gamma \setminus \{\varphi \wedge \psi\}$. In case (c), there is a $\alpha \in \Delta$ such that $\mathfrak{M} \models \alpha$, as $\Delta = \Delta'$. So in each case, $\mathfrak{M}$ satisfies $\varphi \wedge \psi, \Gamma \Rightarrow \Delta$. Since $\mathfrak{M}$ was arbitrary, $\Gamma \Rightarrow \Delta$ is valid. The case where $\varphi \wedge \psi$ is inferred from ψ is handled the same, changing φ to ψ.

5. The last inference is $\vee$ right: There are two variants: $\varphi \vee \psi$ may be inferred on the right from φ or from ψ on the right side of the premise. In the first case, $\varphi \in \Delta'$. Consider a structure $\mathfrak{M}$. Since $\Gamma' \Rightarrow \Delta'$ is valid, (a) $\mathfrak{M} \models \varphi$, (b) $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma'$, or (c) $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta' \setminus \{\varphi\}$. In case (a), $\mathfrak{M} \models \varphi \vee \psi$. In case (b), there is $\alpha \in \Gamma$ such that $\mathfrak{M} \not\models \alpha$, as $\Gamma = \Gamma'$. In case (c), there is an $\alpha \in \Delta$ such that $\mathfrak{M} \models \alpha$, since $\Delta' \setminus \{\varphi\} \subseteq \Delta$. So in each case, $\mathfrak{M}$ satisfies $\varphi \wedge \psi, \Gamma \Rightarrow \Delta$. Since $\mathfrak{M}$ was arbitrary, $\Gamma \Rightarrow \Delta$ is valid. The case where $\varphi \vee \psi$ is inferred from ψ is handled the same, changing φ to ψ.

6. The last inference is $\rightarrow$ right: Then $\varphi \in \Gamma'$, $\psi \in \Delta'$, $\Gamma' \setminus \{\varphi\} \subseteq \Gamma$ and $\Delta' \setminus \{\psi\} \subseteq \Delta$. Since $\Gamma' \Rightarrow \Delta'$ is valid, for any structure $\mathfrak{M}$, (a) $\mathfrak{M} \not\models \varphi$, (b) $\mathfrak{M} \models \psi$, (c) $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma' \setminus \{\varphi\}$, or $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta' \setminus \{\psi\}$. In cases (a) and (b), $\mathfrak{M} \models \varphi \rightarrow \psi$. In case (c), for some $\alpha \in \Gamma$, $\mathfrak{M} \not\models \alpha$. In case (d), for some $\alpha \in \Delta$, $\mathfrak{M} \models \alpha$. In each case, $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$. Since $\mathfrak{M}$ was arbitrary, $\Gamma \Rightarrow \Delta$ is valid.

7. The last inference is $\forall$ left: Then there is a formula $\varphi(x)$ and a ground term t such that $\varphi(t) \in \Gamma'$, $\forall x\, \varphi(x) \in \Gamma$, and $\Gamma' \setminus \{\varphi(t)\} \subseteq \Gamma$. Consider a structure $\mathfrak{M}$. Since $\Gamma' \Rightarrow \Delta'$ is valid, (a) $\mathfrak{M} \not\models \varphi(t)$, (b) $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma' \setminus \{\varphi(t)\}$, or (c) $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta'$. In case (a), $\mathfrak{M} \not\models \forall x\, \varphi(x)$.

In case (b), there is an $\alpha \in \Gamma \setminus \{\varphi(t)\}$ such that $\mathfrak{M} \not\models \alpha$. In case (c), there is a $\alpha \in \Delta$ such that $\mathfrak{M} \models \alpha$, as $\Delta = \Delta'$. So in each case, $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$. Since $\mathfrak{M}$ was arbitrary, $\Gamma \Rightarrow \Delta$ is valid.

8. The last inference is $\exists$ right: Exercise.

9. The last inference is $\forall$ right: Then there is a formula $\varphi(x)$ and a constant symbol a such that $\varphi(a) \in \Delta'$, $\forall x\, \varphi(x) \in \Delta$, and $\Delta' \setminus \{\varphi(a)\} \subseteq \Delta$. Furthermore, $a \notin \Gamma \cup \Delta$. Consider a structure $\mathfrak{M}$. Since $\Gamma' \Rightarrow \Delta'$ is valid, (a) $\mathfrak{M} \models \varphi(a)$, (b) $\mathfrak{M} \not\models \alpha$ for some $\alpha \in \Gamma'$, or (c) $\mathfrak{M} \models \alpha$ for some $\alpha \in \Delta' \setminus \{\varphi(a)\}$.

   First, suppose (a) is the case but neither (b) nor (c), i.e., $\mathfrak{M} \models \alpha$ for all $\alpha \in \Gamma'$ and $\mathfrak{M} \not\models \alpha$ for all $\alpha \in \Delta' \setminus \{\varphi(a)\}$. In other words, assume $\mathfrak{M} \models \varphi(a)$ and that $\mathfrak{M}$ does not satisfy $\Gamma' \Rightarrow \Delta' \setminus \{\varphi(a)\}$. Since $a \notin \Gamma \cup \Delta$, also $a \notin \Gamma' \cup (\Delta' \setminus \{\varphi(a)\})$. Thus, if $\mathfrak{M}'$ is like $\mathfrak{M}$ except that $a^{\mathfrak{M}} \neq a^{\mathfrak{M}'}$, $\mathfrak{M}'$ also does not satisfy $\Gamma' \Rightarrow \Delta' \setminus \{\varphi(a)\}$ by extensionality. But since $\Gamma' \Rightarrow \Delta'$ is valid, we must have $\mathfrak{M}' \models \varphi(a)$.

   We now show that $\mathfrak{M} \models \forall x\, \varphi(x)$. To do this, we have to show that for every variable assignment s, $\mathfrak{M}, s \models \forall x\, \varphi(x)$. This in turn means that for every x-variant s' of s, we must have $\mathfrak{M}, s' \models \varphi(x)$. So consider any variable assignment s and let s' be an x-variant of s. Since Γ' and Δ' consist entirely of sentences, $\mathfrak{M}, s \models \alpha$ iff $\mathfrak{M}, s' \models \alpha$ iff $\mathfrak{M} \models \alpha$ for all $\alpha \in \Gamma' \cup \Delta'$. Let $\mathfrak{M}'$ be like $\mathfrak{M}$ except that $a^{\mathfrak{M}'} = s'(x)$. Then $\mathfrak{M}, s' \models \varphi(x)$ iff $\mathfrak{M}' \models \varphi(a)$ (as $\varphi(x)$ does not contain a). Since we've already established that $\mathfrak{M}' \models \varphi(a)$ for all $\mathfrak{M}'$ which differ from $\mathfrak{M}$ at most in what they assign to a, this means that $\mathfrak{M}, s' \models \varphi(x)$. Thus weve shown that $\mathfrak{M}, s \models \forall x\, \varphi(x)$. Since s is an arbitrary variable assignment and $\forall x\, \varphi(x)$ is a sentence, then $\mathfrak{M} \models \forall x\, \varphi(x)$.

   If (b) is the case, there is a $\alpha \in \Gamma$ such that $\mathfrak{M} \not\models \alpha$, as $\Gamma = \Gamma'$. If (c) is the case, there is an $\alpha \in \Delta' \setminus \{\varphi(a)\}$ such that $\mathfrak{M} \models \alpha$. So in each case, $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$. Since $\mathfrak{M}$ was arbitrary, $\Gamma \Rightarrow \Delta$ is valid.

10. The last inference is $\exists$ left: Exercise.

Now let's consider the possible inferences with two premises: cut, $\vee$ left, $\wedge$ right, and $\rightarrow$ left.

1. The last inference is a cut: Suppose the premises are $\Gamma' \Rightarrow \Delta'$ and $\Pi' \Rightarrow \Lambda'$ and the cut formula φ is in both Δ' and Π'. Since each is valid, every structure $\mathfrak{M}$ satisfies both premises. We distinguish two cases: (a) $\mathfrak{M} \not\models \varphi$ and (b) $\mathfrak{M} \models \varphi$. In case (a), in order for $\mathfrak{M}$ to satisfy the left premise, it must satisfy $\Gamma' \Rightarrow \Delta' \setminus \{\varphi\}$. But $\Gamma' \subseteq \Gamma$ and $\Delta' \setminus \{\varphi\} \subseteq \Delta$, so $\mathfrak{M}$ also satisfies $\Gamma \Rightarrow \Delta$. In case (b), in order for $\mathfrak{M}$ to satisfy the right premise, it must satisfy $\Pi' \setminus \{\varphi\} \Rightarrow \Lambda'$. But $\Pi' \setminus \{\varphi\} \subseteq \Gamma$ and $\Lambda' \subseteq \Delta$, so $\mathfrak{M}$ also satisfies $\Gamma \Rightarrow \Delta$.

2. The last inference is $\wedge$ right. The premises are $\Gamma \Rightarrow \Delta'$ and $\Gamma \Rightarrow \Delta''$, where $\varphi \in \Delta'$ an $\psi \in \Delta''$. By induction hypothesis, both are valid. Consider a structure $\mathfrak{M}$. We have two cases: (a) $\mathfrak{M} \not\models \varphi \wedge \psi$ or (b) $\mathfrak{M} \models \varphi \wedge \psi$. In case (a), either $\mathfrak{M} \not\models \varphi$ or $\mathfrak{M} \not\models \psi$. In the former case, in order for $\mathfrak{M}$ to satisfy $\Gamma \Rightarrow \Delta'$, it must already satisfy $\Gamma \Rightarrow \Delta' \setminus \{\varphi\}$. In the latter case, it must satisfy $\Gamma \Rightarrow \Delta'' \setminus \{\psi\}$. But since both $\Delta' \setminus \{\varphi\} \subseteq \Delta$ and $\Delta'' \setminus \{\psi\} \subseteq \Delta$, that means $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$. In case (b), $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$ since $\varphi \wedge \psi \in \Delta$.

3. The last inference is $\vee$ left: Exercise.

4. The last inference is $\rightarrow$ left. The premises are $\Gamma \Rightarrow \Delta'$ and $\Gamma' \Rightarrow \Delta$, where $\varphi \in \Delta'$ an $\psi \in \Gamma'$. By induction hypothesis, both are valid. Consider a structure $\mathfrak{M}$. We have two cases: (a) $\mathfrak{M} \not\models \varphi \rightarrow \psi$ or (b) $\mathfrak{M} \models \varphi \rightarrow \psi$. In case (a), either $\mathfrak{M} \not\models \varphi$ or $\mathfrak{M} \models \psi$. In the former case, in order for $\mathfrak{M}$ to satisfy $\Gamma \Rightarrow \Delta'$, it must already satisfy $\Gamma \Rightarrow \Delta' \setminus \{\varphi\}$. In the latter case, it must satisfy $\Gamma' \setminus \{\psi\} \Rightarrow \Delta$. But since both $\Delta' \setminus \{\varphi\} \subseteq \Delta$ and $\Gamma' \setminus \{\psi\} \subseteq \Gamma$, that means $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$. In case (b), $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$ since $\varphi \rightarrow \psi \in \Gamma$.

□

Corollary 3.22. *If $\vdash \varphi$ then φ is valid.*

Corollary 3.23. *If $\Gamma \vdash \varphi$ then $\Gamma \models \varphi$.*

Proof. If $\Gamma \vdash \varphi$ then for some finite subset $\Gamma_0 \subseteq \Gamma$, there is a derivation of $\Gamma_0 \Rightarrow \varphi$. By Theorem 3.21, every structure $\mathfrak{M}$ either makes some $\psi \in \Gamma_0$ false or makes φ true. Hence, if $\mathfrak{M} \models \Gamma_0$ then also $\mathfrak{M} \models \varphi$. □

Corollary 3.24. *If Γ is satisfiable, then it is consistent.*

Proof. We prove the contrapositive. Suppose that Γ is not consistent. Then $\Gamma \vdash \bot$, i.e., there is a finite $\Gamma_0 \subseteq \Gamma$ and a derivation of $\Gamma_0 \Rightarrow \bot$. By Theorem 3.21, $\Gamma_0 \Rightarrow \bot$ is valid. Since $\mathfrak{M} \not\models \bot$ for every structure $\mathfrak{M}$, for $\mathfrak{M}$ to satisfy $\Gamma_0 \Rightarrow \bot$ there must be an $\alpha \in \Gamma_0$ so that $\mathfrak{M} \not\models \alpha$, and since $\Gamma_0 \subseteq \Gamma$, that α is also in Γ. In other words, no $\mathfrak{M}$ satisfies Γ, i.e., Γ is not satisfiable. □

3.6 Derivations with Identity predicate

Derivations with the identity predicate require additional inference rules.

Initial sequents for $=$: If t is a closed term, then $\Rightarrow t = t$ is an initial sequent.

Rules for $=$:

$$\frac{\Gamma, t_1 = t_2 \Rightarrow \Delta, \varphi(t_1)}{\Gamma, t_1 = t_2 \Rightarrow \Delta, \varphi(t_2)} = \quad \text{and} \quad \frac{\Gamma, t_1 = t_2 \Rightarrow \Delta, \varphi(t_2)}{\Gamma, t_1 = t_2 \Rightarrow \Delta, \varphi(t_1)} =$$

where t_1 and t_2 are closed terms.

Example 3.25. If s and t are ground terms, then $\varphi(s), s = t \vdash \varphi(t)$:

$$\dfrac{\dfrac{\varphi(s) \Rightarrow \varphi(s)}{\varphi(s), s = t \Rightarrow \varphi(s)}\,\text{weak}}{\varphi(s), s = t \Rightarrow \varphi(t)}\,=$$

This may be familiar as the principle of substitutability of identicals, or Leibniz' Law.

LK proves that $=$ is symmetric and transitive:

$$\dfrac{\dfrac{t_1 = t_2 \Rightarrow t_1 = t_1}{t_1 = t_2 \Rightarrow t_1 = t_1}\,\text{weak}}{t_1 = t_2 \Rightarrow t_2 = t_1}\,= \qquad \dfrac{\dfrac{t_1 = t_2 \Rightarrow t_1 = t_2}{t_1 = t_2, t_2 = t_3 \Rightarrow t_1 = t_2}\,\text{weak}}{t_1 = t_2, t_2 = t_3 \Rightarrow t_1 = t_3}\,=$$

In the proof on the left, the formula $x = t_1$ is our $\varphi(x)$, and correspondingly, $\varphi(t_2) \equiv x[t_2/x] = t_1$. On the right, we take $\varphi(x)$ to be $t_1 = x$.

Proposition 3.26. LK *with initial sequents and rules for identity is sound.*

Proof. Initial sequents of the form $\Rightarrow t = t$ are valid, since for every structure $\mathfrak{M}$, $\mathfrak{M} \models t = t$. (Note that we assume the term t to be ground, i.e., it contains no variables, so variable assignments are irrelevant).

Suppose the last inference in a derivation is $=$. Then the premise $\Gamma' \Rightarrow \Delta'$ contains $t_1 = t_2$ on the left and $\varphi(t_1)$ on the right, and the conclusion is $\Gamma \Rightarrow \Delta$ where $\Gamma = \Gamma'$ and $\Delta = (\Delta' \setminus \{\varphi(t_1)\}) \cup \{\varphi(t_2)\}$. Consider a structure $\mathfrak{M}$. Since, by induction hypothesis, the premise $\Gamma' \Rightarrow \Delta'$ is valid, either (a) for some $\alpha \in \Gamma'$, $\mathfrak{M} \not\models \alpha$, (b) for some $\alpha \in \Delta' \setminus \{\varphi(s)\}$, $\mathfrak{M} \models \alpha$, or (c) $\mathfrak{M} \models \varphi(t_1)$. In both cases cases (a) and (b), since $\Gamma = \Gamma'$, and $\Delta' \setminus \{\varphi(s)\} \subseteq \Delta$, $\mathfrak{M}$ satisfies $\Gamma \Rightarrow \Delta$. So assume cases (a) and (b) do not apply, but case (c) does. If (a) does not apply, $\mathfrak{M} \models \alpha$ for all $\alpha \in \Gamma'$, in particular, $\mathfrak{M} \models t_1 = t_2$. Therefore, $\mathrm{Val}^{\mathfrak{M}}(t_1) = \mathrm{Val}^{\mathfrak{M}}(t_2)$. Let s be any variable assignment, and s' be the x-variant given by $s'(x) = \mathrm{Val}^{\mathfrak{M}}(t_1) = \mathrm{Val}^{\mathfrak{M}}(t_2)$. By Proposition 1.39, $\mathfrak{M}, s \models \varphi(t_2)$ iff $\mathfrak{M}, s' \models \varphi(x)$ iff $\mathfrak{M}, s \models \varphi(t_1)$. Since $\mathfrak{M} \models \varphi(t_1)$ therefore $\mathfrak{M} \models \varphi(t_2)$. □

Chapter 4

The Completeness Theorem

4.1 Introduction

The completeness theorem is one of the most fundamental results about logic. It comes in two formulations, the equivalence of which we'll prove. In its first formulation it says something fundamental about the relationship between semantic consequence and our proof system: if a sentence φ follows from some sentences Γ, then there is also a derivation that establishes $\Gamma \vdash \varphi$. Thus, the proof system is as strong as it can possibly be without proving things that don't actually follow. In its second formulation, it can be stated as a model existence result: every consistent set of sentences is satisfiable.

These aren't the only reasons the completeness theorem—or rather, its proof—is important. It has a number of important consequences, some of which we'll discuss separately. For instance, since any derivation that shows $\Gamma \vdash \varphi$ is finite and so can only use finitely many of the sentences in Γ, it follows by the completeness theorem that if φ is a consequence of Γ, it is already a consequence of a finite subset of Γ. This is called *compactness*. Equivalently, if every finite subset of Γ is consistent, then Γ itself must be consistent. It also follows from *the proof of* the completeness theorem that any satisfiable set of sentences has a finite or denumerable model. This result is called the Löwenheim-Skolem theorem.

4.2 Outline of the Proof

The proof of the completeness theorem is a bit complex, and upon first reading it, it is easy to get lost. So let us outline the proof. The first step is a shift of perspective, that allows us to see a route to a proof. When completeness is thought of as "whenever $\Gamma \models \varphi$ then $\Gamma \vdash \varphi$," it may be hard to even come up with an idea: for to show that $\Gamma \vdash \varphi$ we have to find a derivation, and it does not look like the hypothesis that $\Gamma \models \varphi$ helps us for this in any way. For some proof systems it is possible to directly construct a derivation, but we will take

a slightly different tack. The shift in perspective required is this: completeness can also be formulated as: "if Γ is consistent, it has a model." Perhaps we can use the information in Γ together with the hypothesis that it is consistent to construct a model. After all, we know what kind of model we are looking for: one that is as Γ describes it!

If Γ contains only atomic sentences, it is easy to construct a model for it: for atomic sentences are all of the form $P(a_1, \ldots, a_n)$ where the a_i are constant symbols. So all we have to do is come up with a domain $|\mathfrak{M}|$ and an interpretation for P so that $\mathfrak{M} \models P(a_1, \ldots, a_n)$. But nothing's easier than that: put $|\mathfrak{M}| = \mathbb{N}$, $c_i^{\mathfrak{M}} = i$, and for every $P(a_1, \ldots, a_n) \in \Gamma$, put the tuple $\langle k_1, \ldots, k_n \rangle$ into $P^{\mathfrak{M}}$, where k_i is the index of the constant symbol a_i (i.e., $a_i \equiv c_{k_i}$).

Now suppose Γ contains some sentence $\neg\psi$, with ψ atomic. We might worry that the construction of $\mathfrak{M}$ interferes with the possibility of making $\neg\psi$ true. But here's where the consistency of Γ comes in: if $\neg\psi \in \Gamma$, then $\psi \notin \Gamma$, or else Γ would be inconsistent. And if $\psi \notin \Gamma$, then according to our construction of $\mathfrak{M}$, $\mathfrak{M} \not\models \psi$, so $\mathfrak{M} \models \neg\psi$. So far so good.

Now what if Γ contains complex, non-atomic formulas? Say, it contains $\varphi \wedge \psi$. Then we should proceed as if both φ and ψ were in Γ. And if $\varphi \vee \psi \in \Gamma$, then we will have to make at least one of them true, i.e., proceed as if one of them was in$'\Gamma$.

This suggests the following idea: we add additional sentences to Γ so as to (a) keep the resulting set consistent and (b) make sure that for every possible atomic sentence φ, either φ is in the resulting set, or $\neg\varphi$, and (c) such that, whenever $\varphi \wedge \psi$ is in the set, so are both φ and ψ, if $\varphi \vee \psi$ is in the set, at least one of φ or ψ is also, etc. We keep doing this (potentially forever). Call the set of all sentences so added Γ^*. Then our construction above would provide us with a structure for which we could prove, by induction, that all sentences in Γ^* are true in $\mathfrak{M}$, and hence also all sentence in Γ since $\Gamma \subseteq \Gamma^*$.

There is one wrinkle in this plan: if $\exists x\, \varphi(x) \in \Gamma$ we would hope to be able to pick some constant symbol c and add $\varphi(c)$ in this process. But how do we know we can always do that? Perhaps we only have a few constant symbols in our language, and for each one of them we have $\neg\psi(c) \in \Gamma$. We can't also add $\psi(c)$, since this would make the set inconsistent, and we wouldn't know whether $\mathfrak{M}$ has to make $\psi(c)$ or $\neg\psi(c)$ true. Moreover, it might happen that Γ contains only sentences in a language that has no constant symbols at all (e.g., the language of set theory).

The solution to this problem is to simply add infinitely many constants at the beginning, plus sentences that connect them with the quantifiers in the right way. (Of course, we have to verify that this cannot introduce an inconsistency.)

Our original construction works well if we only have constant symbols in the atomic sentences. But the language might also contain function symbols. In that case, it might be tricky to find the right functions on $\mathbb{N}$ to assign to

these function symbols to make everything work. So here's another trick: instead of using i to interpret c_i, just take the set of constant symbols itself as the domain. Then $\mathfrak{M}$ can assign every constant symbol to itself: $c_i^{\mathfrak{M}} = c_i$. But why not go all the way: let $|\mathfrak{M}|$ be all *terms* of the language! If we do this, there is an obvious assignment of functions (that take terms as arguments and have terms as values) to function symbols: we assign to the function symbol f_i^n the function which, given n terms $t_1, \ldots, t_n$ as input, produces the term $f_i^n(t_1, \ldots, t_n)$ as value.

The last piece of the puzzle is what to do with $=$. The predicate symbol $=$ has a fixed interpretation: $\mathfrak{M} \models t = t'$ iff $\mathrm{Val}^{\mathfrak{M}}(t) = \mathrm{Val}^{\mathfrak{M}}(t')$. Now if we set things up so that the value of a term t is t itself, then this structure will make *no* sentence of the form $t = t'$ true unless t and t' are one and the same term. And of course this is a problem, since basically every interesting theory in a language with function symbols will have as theorems sentences $t = t'$ where t and t' are not the same term (e.g., in theories of arithmetic: $(\mathsf{o} + \mathsf{o}) = \mathsf{o}$). To solve this problem, we change the domain of $\mathfrak{M}$: instead of using terms as the objects in $|\mathfrak{M}|$, we use sets of terms, and each set is so that it contains all those terms which the sentences in Γ require to be equal. So, e.g., if Γ is a theory of arithmetic, one of these sets will contain: o, $(\mathsf{o} + \mathsf{o})$, $(\mathsf{o} \times \mathsf{o})$, etc. This will be the set we assign to o, and it will turn out that this set is also the value of all the terms in it, e.g., also of $(\mathsf{o} + \mathsf{o})$. Therefore, the sentence $(\mathsf{o} + \mathsf{o}) = \mathsf{o}$ will be true in this revised structure.

4.3 Maximally Consistent Sets of Sentences

Definition 4.1. A set Γ of sentences is *maximally consistent* iff

1. Γ is consistent, and
2. if $\Gamma \subsetneq \Gamma'$, then Γ' is inconsistent.

An alternate definition equivalent to the above is: a set Γ of sentences is *maximally consistent* iff

1. Γ is consistent, and
2. If $\Gamma \cup \{\varphi\}$ is consistent, then $\varphi \in \Gamma$.

In other words, one cannot add sentences not already in Γ to a maximally consistent set Γ without making the resulting larger set inconsistent.

Maximally consistent sets are important in the completeness proof since we can guarantee that every consistent set of sentences Γ is contained in a maximally consistent set Γ^*, and a maximally consistent set contains, for each sentence φ, either φ or its negation $\neg\varphi$. This is true in particular for atomic sentences, so from a maximally consistent set in a language suitably expanded by constant symbols, we can construct a structure where the interpretation of

predicate symbols is defined according to which atomic sentences are in Γ^*. This structure can then be shown to make all sentences in Γ^* (and hence also in Γ) true. The proof of this latter fact requires that $\neg\varphi \in \Gamma^*$ iff $\varphi \notin \Gamma^*$, $(\varphi \vee \psi) \in \Gamma^*$ iff $\varphi \in \Gamma^*$ or $\psi \in \Gamma^*$, etc.

Proposition 4.2. *Suppose Γ is maximally consistent. Then:*

1. *If $\Gamma \vdash \varphi$, then $\varphi \in \Gamma$.*
2. *For any φ, either $\varphi \in \Gamma$ or $\neg\varphi \in \Gamma$.*
3. *$(\varphi \wedge \psi) \in \Gamma$ iff both $\varphi \in \Gamma$ and $\psi \in \Gamma$.*
4. *$(\varphi \vee \psi) \in \Gamma$ iff either $\varphi \in \Gamma$ or $\psi \in \Gamma$.*
5. *$(\varphi \to \psi) \in \Gamma$ iff either $\varphi \notin \Gamma$ or $\psi \in \Gamma$.*

Proof. Let us suppose for all of the following that Γ is maximally consistent.

1. If $\Gamma \vdash \varphi$, then $\varphi \in \Gamma$.

   Suppose that $\Gamma \vdash \varphi$. Suppose to the contrary that $\varphi \notin \Gamma$: then since Γ is maximally consistent, $\Gamma \cup \{\varphi\}$ is inconsistent, hence $\Gamma \cup \{\varphi\} \vdash \bot$. By Proposition 3.17(1) Γ is inconsistent. This contradicts the assumption that Γ is consistent. Hence, it cannot be the case that $\varphi \notin \Gamma$, so $\varphi \in \Gamma$.

2. For any φ, either $\varphi \in \Gamma$ or $\neg\varphi \in \Gamma$.

   Suppose to the contrary that for some φ both $\varphi \notin \Gamma$ and $\neg\varphi \notin \Gamma$. Since Γ is maximally consistent, $\Gamma \cup \{\varphi\}$ and $\Gamma \cup \{\neg\varphi\}$ are both inconsistent, so $\Gamma \cup \{\varphi\} \vdash \bot$ and $\Gamma \cup \{\neg\varphi\} \vdash \bot$. By Proposition 3.17(3), Γ is inconsistent, a contradiction. Hence there cannot be such a sentence φ and, for every φ, $\varphi \in \Gamma$ or $\neg\varphi \in \Gamma$.

3. $(\varphi \wedge \psi) \in \Gamma$ iff both $\varphi \in \Gamma$ and $\psi \in \Gamma$:

   For the forward direction, suppose $(\varphi \wedge \psi) \in \Gamma$. Then $\Gamma \vdash \varphi \wedge \psi$. By Proposition 3.17(6), $\Gamma \vdash \varphi$ and $\Gamma \vdash \psi$. By (1), $\varphi \in \Gamma$ and $\psi \in \Gamma$, as required.

   For the reverse direction, let $\varphi \in \Gamma$ and $\psi \in \Gamma$. Then $\Gamma \vdash \varphi$ and $\Gamma \vdash \psi$. By Proposition 3.17(7), $\Gamma \vdash \varphi \wedge \psi$. By (1), $(\varphi \wedge \psi) \in \Gamma$.

4. $(\varphi \vee \psi) \in \Gamma$ iff either $\varphi \in \Gamma$ or $\psi \in \Gamma$.

   For the contrapositive of the forward direction, suppose that $\varphi \notin \Gamma$ and $\psi \notin \Gamma$. We want to show that $(\varphi \vee \psi) \notin \Gamma$. Since Γ is maximally consistent, $\Gamma \cup \{\varphi\} \vdash \bot$ and $\Gamma \cup \{\psi\} \vdash \bot$. By Proposition 3.17(4), $\Gamma \cup \{(\varphi \vee \psi)\}$ is inconsistent. Hence, $(\varphi \vee \psi) \notin \Gamma$, as required.

   For the reverse direction, suppose that $\varphi \in \Gamma$ or $\psi \in \Gamma$. Then $\Gamma \vdash \varphi$ or $\Gamma \vdash \psi$. By Proposition 3.17(5), $\Gamma \vdash \varphi \vee \psi$. By (1), $(\varphi \vee \psi) \in \Gamma$, as required.

5. $(\varphi \to \psi) \in \Gamma$ iff either $\varphi \notin \Gamma$ or $\psi \in \Gamma$:

   For the forward direction, let $(\varphi \to \psi) \in \Gamma$, and suppose to the contrary that $\varphi \in \Gamma$ and $\psi \notin \Gamma$. On these assumptions, $\Gamma \vdash \varphi \to \psi$ and $\Gamma \vdash \varphi$. By Proposition 3.17(8), $\Gamma \vdash \psi$. But then by (1), $\psi \in \Gamma$, contradicting the assumption that $\psi \notin \Gamma$.

   For the reverse direction, first consider the case where $\varphi \notin \Gamma$. By (2), $\neg\varphi \in \Gamma$ and hence $\Gamma \vdash \neg\varphi$. By Proposition 3.17(9), $\Gamma \vdash \varphi \to \psi$. Again by (1), we get that $(\varphi \to \psi) \in \Gamma$, as required.

   Now consider the case where $\psi \in \Gamma$. Then $\Gamma \vdash \psi$ and by Proposition 3.17(9), $\Gamma \vdash \varphi \to \psi$. By (1), $(\varphi \to \psi) \in \Gamma$.

□

4.4 Henkin Expansion

Part of the challenge in proving the completeness theorem is that the model we construct from a maximally consistent set Γ must make all the quantified formulas in Γ true. In order to guarantee this, we use a trick due to Leon Henkin. In essence, the trick consists in expanding the language by infinitely many constants and adding, for each formula with one free variable $\varphi(x)$ a formula of the form $\exists x\, \varphi \to \varphi(c)$, where c is one of the new constant symbols. When we construct the structure satisfying Γ, this will guarantee that each true existential sentence has a witness among the new constants.

Lemma 4.3. *If Γ is consistent in $\mathcal{L}$ and $\mathcal{L}'$ is obtained from $\mathcal{L}$ by adding a denumerable set of new constant symbols $d_1, d_2, \ldots$, then Γ is consistent in $\mathcal{L}'$.*

Definition 4.4. A set Γ of formulas of a language $\mathcal{L}$ is *saturated* if and only if for each formula $\varphi \in \mathrm{Frm}(\mathcal{L})$ and variable x there is a constant symbol c such that $\exists x\, \varphi \to \varphi(c) \in \Gamma$.

The following definition will be used in the proof of the next theorem.

Definition 4.5. Let $\mathcal{L}'$ be as in Lemma 4.3. Fix an enumeration $\langle \varphi_1, x_1 \rangle, \langle \varphi_2, x_2 \rangle, \ldots$ of all formula-variable pairs of $\mathcal{L}'$. We define the sentences θ_n by recursion on n. Assuming that $\theta_1, \ldots, \theta_n$ have already been defined, let c_{n+1} be the first new constant symbol among the d_i that does not occur in $\theta_1, \ldots, \theta_n$, and let θ_{n+1} be the formula $\exists x_{n+1}\, \varphi_{n+1}(x_{n+1}) \to \varphi_{n+1}(c_{n+1})$. This includes the case where $n = 0$ and the list of previous θ_i's is empty, i.e., θ_1 is $\exists x_1\, \varphi_1 \to \varphi_1(c_1)$.

Theorem 4.6. *Every consistent set Γ can be extended to a saturated consistent set Γ'.*

Proof. Given a consistent set of sentences Γ in a language $\mathcal{L}$, expand the language by adding a denumerable set of new constant symbols to form $\mathcal{L}'$. By the previous Lemma, Γ is still consistent in the richer language. Further, let θ_i

be as in the previous definition: then $\Gamma \cup \{\theta_1, \theta_2, \dots\}$ is saturated by construction. Let

$$\Gamma_0 = \Gamma$$
$$\Gamma_{n+1} = \Gamma_n \cup \{\theta_{n+1}\}$$

i.e., $\Gamma_n = \Gamma \cup \{\theta_1, \dots, \theta_n\}$, and let $\Gamma' = \bigcup_n \Gamma_n$. To show that Γ' is consistent it suffices to show, by induction on n, that each set Γ_n is consistent.

The induction basis is simply the claim that $\Gamma_0 = \Gamma$ is consistent, which is the hypothesis of the theorem. For the induction step, suppose that Γ_{n-1} is consistent but $\Gamma_n = \Gamma_{n-1} \cup \{\theta_n\}$ is inconsistent. Recall that θ_n is $\exists x_n\, \varphi_n(x_n) \to \varphi_n(c_n)$. where $\varphi(x)$ is a formula of $\mathcal{L}'$ with only the variable x_n free and not containing any constant symbols c_i where $i \geq n$.

If $\Gamma_{n-1} \cup \{\theta_n\}$ is inconsistent, then $\Gamma_{n-1} \vdash \neg\theta_n$, and hence both of the following hold:

$$\Gamma_{n-1} \vdash \exists x_n\, \varphi_n(x_n) \qquad \Gamma_{n-1} \vdash \neg\varphi_n(c_n)$$

Here c_n does not occur in Γ_{n-1} or $\varphi_n(x_n)$ (remember, it was added only with θ_n). By Theorem 3.18, from $\Gamma \vdash \neg\varphi_n(c_n)$, we obtain $\Gamma \vdash \forall x_n\, \neg\varphi_n(x_n)$. Thus we have that both $\Gamma_{n-1} \vdash \exists x_n\, \varphi_n$ and $\Gamma_{n-1} \vdash \forall x_n\, \neg\varphi_n(x_n)$, so Γ itself is inconsistent. (Note that $\forall x_n\, \neg\varphi_n(x_n) \vdash \neg\exists x_n\, \varphi_n(x_n)$.) Contradiction: Γ_{n-1} was supposed to be consistent. Hence $\Gamma_n \cup \{\theta_n\}$ is consistent. □

4.5 Lindenbaum's Lemma

Lemma 4.7 (Lindenbaum's Lemma)**.** *Every consistent set Γ can be extended to a maximally consistent saturated set Γ^*.*

Proof. Let Γ be consistent, and let Γ' be as in the proof of Theorem 4.6: we proved there that $\Gamma \cup \Gamma'$ is a consistent saturated set in the richer language $\mathcal{L}'$ (with the denumerable set of new constants). Let φ_0, φ_1, ... be an enumeration of all the formulas of $\mathcal{L}'$. Define $\Gamma_0 = \Gamma \cup \Gamma'$, and

$$\Gamma_{n+1} = \begin{cases} \Gamma_n \cup \{\varphi_n\} & \text{if } \Gamma_n \cup \{\varphi_n\} \text{ is consistent;} \\ \Gamma_n \cup \{\neg\varphi_n\} & \text{otherwise.} \end{cases}$$

Let $\Gamma^* = \bigcup_{n \geq 0} \Gamma_n$. Since $\Gamma' \subseteq \Gamma^*$, for each formula φ, Γ^* contains a formula of the form $\exists x\, \varphi \to \varphi(c)$ and thus is saturated.

Each Γ_n is consistent: Γ_0 is consistent by definition. If $\Gamma_{n+1} = \Gamma_n \cup \{\varphi\}$, this is because the latter is consistent. If it isn't, $\Gamma_{n+1} = \Gamma_n \cup \{\neg\varphi\}$, which must be consistent. If it weren't, i.e., both $\Gamma_n \cup \{\varphi\}$ and $\Gamma_n \cup \{\neg\varphi\}$ are inconsistent, then $\Gamma_n \vdash \neg\varphi$ and $\Gamma_n \vdash \varphi$, so Γ_n would be inconsistent contrary to induction hypothesis.

Every formula of $\mathrm{Frm}(\mathcal{L}')$ appears on the list used to define Γ^*. If $\varphi_n \notin \Gamma^*$, then that is because $\Gamma_n \cup \{\varphi_n\}$ was inconsistent. But that means that Γ^* is maximally consistent. □

4.6 Construction of a Model

We will begin by showing how to construct a structure which satisfies a maximally consistent, saturated set of sentences in a language $\mathcal{L}$ without $=$.

Definition 4.8. Let Γ^* be a maximally consistent, saturated set of sentences in a language $\mathcal{L}$. The *term model* $\mathfrak{M}(\Gamma^*)$ of Γ^* is the structure defined as follows:

1. The domain $|\mathfrak{M}(\Gamma^*)|$ is the set of all closed terms of $\mathcal{L}$.
2. The interpretation of a constant symbol c is c itself: $c^{\mathfrak{M}(\Gamma^*)} = c$.
3. The function symbol f is assigned the function

$$f^{\mathfrak{M}(\Gamma^*)}(t_1, \ldots, t_n) = f(\mathrm{Val}^{\mathfrak{M}(\Gamma^*)}(t_1), \ldots, \mathrm{Val}^{\mathfrak{M}(\Gamma^*)}(t_1))$$

4. If R is an n-place predicate symbol, then $\langle t_1, \ldots, t_n \rangle \in R^{\mathfrak{M}(\Gamma^*)}$ iff $R(t_1, \ldots, t_n) \in \Gamma^*$.

Lemma 4.9 (Truth Lemma)**.** *Suppose φ does not contain $=$. Then $\mathfrak{M}(\Gamma^*) \models \varphi$ iff $\varphi \in \Gamma^*$.*

Proof. We prove both directions simultaneously, and by induction on φ.

1. $\varphi \equiv R(t_1, \ldots, t_n)$: $\mathfrak{M}(\Gamma^*) \models R(t_1, \ldots, t_n)$ iff $\langle t_1, \ldots, t_n \rangle \in R^{\mathfrak{M}(\Gamma^*)}$ (by the definition of satisfaction) iff $R(t_1, \ldots, t_n) \in \Gamma^*$ (the construction of $\mathfrak{M}(\Gamma^*)$.
2. $\varphi \equiv \neg\psi$: $\mathfrak{M}(\Gamma^*) \models \varphi$ iff $\mathfrak{M}(\Gamma^*) \not\models \psi$ (by definition of satisfaction). By induction hypothesis, $\mathfrak{M}(\Gamma^*) \not\models \psi$ iff $\psi \notin \Gamma^*$. By Proposition 4.2(2), $\neg\psi \in \Gamma^*$ if $\psi \notin \Gamma^*$; and $\neg\psi \notin \Gamma^*$ if $\psi \in \Gamma^*$ since Γ^* is consistent.
3. $\varphi \equiv \psi \wedge \chi$: $\mathfrak{M}(\Gamma^*) \models \varphi$ iff we have both $\mathfrak{M}(\Gamma^*) \models \psi$ and $\mathfrak{M}(\Gamma^*) \models \chi$ (by definition of satisfaction) iff both $\psi \in \Gamma^*$ and $\chi \in \Gamma^*$ (by the induction hypothesis). By Proposition 4.2(3), this is the case iff $(\psi \wedge \chi) \in \Gamma^*$.
4. $\varphi \equiv \psi \vee \chi$: $\mathfrak{M}(\Gamma^*) \models \varphi$ iff at $\mathfrak{M}(\Gamma^*) \models \psi$ or $\mathfrak{M}(\Gamma^*) \models \chi$ (by definition of satisfaction) iff $\psi \in \Gamma^*$ or $\chi \in \Gamma^*$ (by induction hypothesis). This is the case iff $(\psi \vee \chi) \in \Gamma^*$ (by Proposition 4.2(4)).
5. $\varphi \equiv \psi \to \chi$: $\mathfrak{M}(\Gamma^*) \models \varphi$ iff $\mathfrak{M}(\Gamma^* \not\models \psi$ or $\mathfrak{M}(\Gamma^*) \models \chi$ (by definition of satisfaction) iff $\psi \notin \Gamma^*$ or $\chi \in \Gamma^*$ (by induction hypothesis). This is the case iff $(\psi \to \chi) \in \Gamma^*$ (by Proposition 4.2(5)).
6. $\varphi \equiv \forall x\, \psi(x)$: Suppose that $\mathfrak{M}(\Gamma^*) \models \varphi$, then for every variable assignment s, $\mathfrak{M}(\Gamma^*), s \models \psi(x)$. Suppose to the contrary that $\forall x\, \psi(x) \notin \Gamma^*$: Then by Proposition 4.2(2), $\neg\forall x\, \psi(x) \in \Gamma^*$. By saturation, $(\exists x\, \neg\psi(x) \to$

$\neg\psi(c)) \in \Gamma^*$ for some c, so by Proposition 4.2(1), $\neg\psi(c) \in \Gamma^*$. Since Γ^* is consistent, $\psi(c) \notin \Gamma^*$. By induction hypothesis, $\mathfrak{M}(\Gamma^*) \not\models \psi(c)$. Therefore, if s' is the variable assignment such that $s(x) = c$, then $\mathfrak{M}(\Gamma^*), s' \not\models \psi(x)$, contradicting the earlier result that $\mathfrak{M}(\Gamma^*), s \models \psi(x)$ for all s. Thus, we have $\varphi \in \Gamma^*$.

Conversely, suppose that $\forall x\, \psi(x) \in \Gamma^*$. By Theorem 3.19 and Proposition 4.2(1), $\psi(t) \in \Gamma^*$ for every term $t \in |\mathfrak{M}(\Gamma^*)|$. By inductive hypothesis, $\mathfrak{M}(\Gamma^*) \models \psi(t)$ for every term $t \in |\mathfrak{M}(\Gamma^*)|$. Let s be the variable assigment with $s(x) = t$. Then $\mathfrak{M}(\Gamma^*), s \models \psi(x)$ for any such s, hence $\mathfrak{M}(\Gamma^*) \models \varphi$.

7. $\varphi \equiv \exists x\, \psi(x)$: First suppose that $\mathfrak{M}(\Gamma^*) \models \varphi$. By the definition of satisfaction, for some variable assignment s, $\mathfrak{M}(\Gamma^*), s \models \psi(x)$. The value $s(x)$ is some term $t \in |\mathfrak{M}(\Gamma^*)|$. Thus, $\mathfrak{M}(\Gamma^*) \models \psi(t)$, and by our induction hypothesis, $\psi(t) \in \Gamma^*$. By Theorem 3.19 we have $\Gamma^* \vdash \exists x\, \psi(x)$. Then, by Proposition 4.2(1), we can conclude that $\varphi \in \Gamma^*$.

   Conversely, suppose that $\exists x\, \psi(x) \in \Gamma^*$. Because Γ^* is saturated, $(\exists x\, \psi(x) \to \psi(c)) \in \Gamma^*$. By Proposition 3.17(8) and Proposition 4.2(1), $\psi(c) \in \Gamma^*$. By inductive hypothesis, $\mathfrak{M}(\Gamma^*) \models \psi(c)$. Now consider the variable assignment with $s(x) = c^{\mathfrak{M}(\Gamma^*)}$. Then $\mathfrak{M}(\Gamma^*), s \models \psi(x)$. By definition of satisfaction, $\mathfrak{M}(\Gamma^*) \models \exists x\, \psi(x)$.

□

4.7 Identity

The construction of the term model given in the preceding section is enough to establish completeness for first-order logic for sets Γ that do not contain $=$. The term model satisfies every $\varphi \in \Gamma^*$ which does not contain $=$ (and hence all $\varphi \in \Gamma$). It does not work, however, if $=$ is present. The reason is that Γ^* then may contain a sentence $t = t'$, but in the term model the value of any term is that term itself. Hence, if t and t' are different terms, their values in the term model—i.e., t and t', respectively—are different, and so $t = t'$ is false. We can fix this, however, using a construction known as "factoring."

Definition 4.10. Let Γ^* be a maximally consistent set of sentences in $\mathcal{L}$. We define the relation $\approx$ on the set of closed terms of $\mathcal{L}$ by

$$t \approx t' \quad \text{iff} \quad t = t' \in \Gamma^*$$

Proposition 4.11. *The relation $\approx$ has the following properties:*

1. *$\approx$ is reflexive.*
2. *$\approx$ is symmetric.*

3. $\approx$ *is transitive.*

4. *If* $t \approx t'$, f *is a function symbol, and* $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ *are terms, then*

$$f(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n) \approx f(t_1, \ldots, t_{i-1}, t', t_{i+1}, \ldots, t_n).$$

5. *If* $t \approx t'$, R *is a function symbol, and* $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ *are terms, then*

$$R(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n) \in \Gamma^* \textit{ iff } R(t_1, \ldots, t_{i-1}, t', t_{i+1}, \ldots, t_n) \in \Gamma^*.$$

Proof. Since Γ^* is maximally consistent, $t = t' \in \Gamma^*$ iff $\Gamma^* \vdash t = t'$. Thus it is enough to show the following:

1. $\Gamma^* \vdash t = t$ for all terms t.
2. If $\Gamma^* \vdash t = t'$ then $\Gamma^* \vdash t' = t$.
3. If $\Gamma^* \vdash t = t'$ and $\Gamma^* \vdash t' = t''$, then $\Gamma^* \vdash t = t''$.
4. If $\Gamma^* \vdash t = t'$, then

$$\Gamma^* \vdash f(t_1, \ldots, t_{i-1}, t, t_{i+1}, , \ldots, t_n) = f(t_1, \ldots, t_{i-1}, t', t_{i+1}, \ldots, t_n)$$

   for every n-place function symbol f and terms $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$.
5. If $\Gamma^* \vdash t = t'$ and $\Gamma^* \vdash R(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n)$, then $\Gamma^* \vdash R(t_1, \ldots, t_{i-1}, t', t_{i+1}, \ldots, t_n)$ for every n-place predicate symbol R and terms $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$.

□

Definition 4.12. Suppose Γ^* is a maximally consistent set in a language $\mathcal{L}$, t is a term, and $\approx$ as in the previous definition. Then:

$$[t]_\approx = \{t' : t' \in \mathrm{Trm}(\mathcal{L}), t \approx t'\}$$

and $\mathrm{Trm}(\mathcal{L})/_\approx = \{[t]_\approx : t \in \mathrm{Trm}(\mathcal{L})\}$.

Definition 4.13. Let $\mathfrak{M} = \mathfrak{M}(\Gamma^*)$ be the term model for Γ^*. Then $\mathfrak{M}/_\approx$ is the following structure:

1. $|\mathfrak{M}/_\approx| = \mathrm{Trm}(\mathcal{L})/_\approx$.
2. $c^{\mathfrak{M}/_\approx} = [c]_\approx$
3. $f^{\mathfrak{M}/_\approx}([t_1]_\approx, \ldots, [t_n]_\approx) = [f(t_1, \ldots, t_n)]_\approx$
4. $\langle [t_1]_\approx, \ldots, [t_n]_\approx \rangle \in R^{\mathfrak{M}/_\approx}$ iff $\mathfrak{M} \models R(t_1, \ldots, t_n)$.

Note that we have defined $f^{\mathfrak{M}/\approx}$ and $R^{\mathfrak{M}/\approx}$ for elements of $\mathrm{Trm}(\mathcal{L})/\approx$ by referring to them as $[t]_\approx$, i.e., via *representatives* $t \in [t]_\approx$. We have to make sure that these definitions do not depend on the choice of these representatives, i.e., that for some other choices t' which determine the same equivalence classes ($[t]_\approx = [t']_\approx$), the definitions yield the same result. For instance, if R is a one-place predicate symbol, the last clause of the definition says that $[t]_\approx \in R^{\mathfrak{M}/\approx}$ iff $\mathfrak{M} \models R(t)$. If for some other term t' with $t \approx t'$, $\mathfrak{M} \not\models R(t)$, then the definition would require $[t']_\approx \notin R^{\mathfrak{M}/\approx}$. If $t \approx t'$, then $[t]_\approx = [t']_\approx$, but we can't have both $[t]_\approx \in R^{\mathfrak{M}/\approx}$ and $[t]_\approx \notin R^{\mathfrak{M}/\approx}$. However, Proposition 4.11 guarantees that this cannot happen.

Proposition 4.14. *$\mathfrak{M}/\approx$ is well defined, i.e., if $t_1, \ldots, t_n$, $t'_1, \ldots, t'_n$ are terms, and $t_i \approx t'_i$ then*

1. *$[f(t_1, \ldots, t_n)]_\approx = [f(t'_1, \ldots, t'_n)]_\approx$, i.e., $f(t_1, \ldots, t_n) \approx f(t'_1, \ldots, t'_n)$ and*
2. *$\mathfrak{M} \models R(t_1, \ldots, t_n)$ iff $\mathfrak{M} \models R(t'_1, \ldots, t'_n)$, i.e., $R(t_1, \ldots, t_n) \in \Gamma^*$ iff $R(t'_1, \ldots, t'_n) \in \Gamma^*$.*

Proof. Follows from Proposition 4.11. □

Lemma 4.15. *$\mathfrak{M}/\approx \models \varphi$ iff $\varphi \in \Gamma^*$ for all sentences φ.*

Proof. By induction on φ, just as in the proof of Lemma 4.9. The only case that needs additional attention is when $\varphi \equiv t = t'$.

$$\begin{aligned} \mathfrak{M}/\approx \models t = t' &\text{ iff } [t]_\approx = [t']_\approx \text{ (by definition of } \mathfrak{M}/\approx) \\ &\text{iff } t \approx t' \text{ (by definition of } [t]_\approx) \\ &\text{iff } t = t' \in \Gamma^* \text{ (by definition of } \approx). \end{aligned}$$

□

Note that while $\mathfrak{M}(\Gamma^*)$ is always enumerable and infinite, $\mathfrak{M}/\approx$ may be finite, since it may turn out that there are only finitely many classes $[t]_\approx$. This is to be expected, since Γ may contain sentences which require any structure in which they are true to be finite. For instance, $\forall x \forall y\, x = y$ is a consistent sentence, but is satisfied only in structures with a domain that contains exactly one element.

4.8 The Completeness Theorem

Theorem 4.16 (Completeness Theorem)**.** *Let Γ be a set of sentences. If Γ is consistent, it is satisfiable.*

Proof. Suppose Γ is consistent. By Lemma 4.7, there is a $\Gamma^* \supseteq \Gamma$ which is maximally consistent and saturated. If Γ does not contain $=$, then by Lemma 4.9, $\mathfrak{M}(\Gamma^*) \models \varphi$ iff $\varphi \in \Gamma^*$. From this it follows in particular that for all $\varphi \in \Gamma$, $\mathfrak{M}(\Gamma^*) \models \varphi$, so Γ is satisfiable. If Γ does contain $=$, then by Lemma 4.15, $\mathfrak{M}/{\approx} \models \varphi$ iff $\varphi \in \Gamma^*$ for all sentences φ. In particular, $\mathfrak{M}/{\approx} \models \varphi$ for all $\varphi \in \Gamma$, so Γ is satisfiable. □

Corollary 4.17 (Completeness Theorem, Second Version)**.** *For all Γ and φ sentences: if $\Gamma \models \varphi$ then $\Gamma \vdash \varphi$.*

Proof. Note that the Γ's in Corollary 4.17 and Theorem 4.16 are universally quantified. To make sure we do not confuse ourselves, let us restate Theorem 4.16 using a different variable: for any set of sentences Δ, if Δ is consistent, it is satisfiable. By contraposition, if Δ is not satisfiable, then Δ is inconsistent. We will use this to prove the corollary.

Suppose that $\Gamma \models \varphi$. Then $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable by Proposition 1.44. Taking $\Gamma \cup \{\neg\varphi\}$ as our Δ, the previous version of Theorem 4.16 gives us that $\Gamma \cup \{\neg\varphi\}$ is inconsistent. By Proposition 3.13, $\Gamma \vdash \varphi$. □

4.9 The Compactness Theorem

Definition 4.18. A set Γ of formulas is *finitely satisfiable* if and only if every finite $\Gamma_0 \subseteq \Gamma$ is satisfiable.

Theorem 4.19 (Compactness Theorem)**.** *The following hold for any sentences Γ and φ:*

1. *$\Gamma \models \varphi$ iff there is a finite $\Gamma_0 \subseteq \Gamma$ such that $\Gamma_0 \models \varphi$.*
2. *Γ is satisfiable if and only if it is finitely satisfiable.*

Proof. We prove (2). If Γ is satisfiable, then there is a structure $\mathfrak{M}$ such that $\mathfrak{M} \models \varphi$ for all $\varphi \in \Gamma$. Of course, this $\mathfrak{M}$ also satisfies every finite subset of Γ, so Γ is finitely satisfiable.

Now suppose that Γ is finitely satisfiable. Then every finite subset $\Gamma_0 \subseteq \Gamma$ is satisfiable. By soundness, every finite subset is consistent. Then Γ itself must be consistent. For assume it is not, i.e., $\Gamma \vdash \bot$. But derivations are finite, and so already some finite subset $\Gamma_0 \subseteq \Gamma$ must be inconsistent (cf. Proposition 3.15). But we just showed they are all consistent, a contradiction. Now by completeness, since Γ is consistent, it is satisfiable. □

4.10 The Löwenheim-Skolem Theorem

Theorem 4.20. *If Γ is consistent then it has a denumerable model, i.e., it is satisfiable in a structure whose domain is either finite or infinite but enumerable.*

Proof. If Γ is consistent, the structure $\mathfrak{M}$ delivered by the proof of the completeness theorem has a domain $|\mathfrak{M}|$ whose cardinality is bounded by that of the set of the terms of the language $\mathcal{L}$. So $\mathfrak{M}$ is at most denumerable. □

Theorem 4.21. *If Γ is consistent set of sentences in the language of first-order logic without identity, then it has a denumerable model, i.e., it is satisfiable in a structure whose domain is infinite and enumerable.*

Proof. If Γ is consistent and contains no sentences in which identity appears, then the structure $\mathfrak{M}$ delivered by the proof of the completness theorem has a domain $|\mathfrak{M}|$ whose cardinality is identical to that of the set of the terms of the language $\mathcal{L}$. So $\mathfrak{M}$ is denumerably infinite. □

Chapter 5

Beyond First-order Logic

5.1 Overview

First-order logic is not the only system of logic of interest: there are many extensions and variations of first-order logic. A logic typically consists of the formal specification of a language, usually, but not always, a deductive system, and usually, but not always, an intended semantics. But the technical use of the term raises an obvious question: what do logics that are not first-order logic have to do with the word "logic," used in the intuitive or philosophical sense? All of the systems described below are designed to model reasoning of some form or another; can we say what makes them logical?

No easy answers are forthcoming. The word "logic" is used in different ways and in different contexts, and the notion, like that of "truth," has been analyzed from numerous philosophical stances. For example, one might take the goal of logical reasoning to be the determination of which statements are necessarily true, true a priori, true independent of the interpretation of the nonlogical terms, true by virtue of their form, or true by linguistic convention; and each of these conceptions requires a good deal of clarification. Even if one restricts one's attention to the kind of logic used in mathematical, there is little agreement as to its scope. For example, in the *Principia Mathematica*, Russell and Whitehead tried to develop mathematics on the basis of logic, in the *logicist* tradition begun by Frege. Their system of logic was a form of higher-type logic similar to the one described below. In the end they were forced to introduce axioms which, by most standards, do not seem purely logical (notably, the axiom of infinity, and the axiom of reducibility), but one might nonetheless hold that some forms of higher-order reasoning should be accepted as logical. In contrast, Quine, whose ontology does not admit "propositions" as legitimate objects of discourse, argues that second-order and higher-order logic are really manifestations of set theory in sheep's clothing; in other words, systems involving quantification over predicates are not purely logical.

For now, it is best to leave such philosophical issues for a rainy day, and

simply think of the systems below as formal idealizations of various kinds of reasoning, logical or otherwise.

5.2 Many-Sorted Logic

In first-order logic, variables and quantifiers range over a single domain. But it is often useful to have multiple (disjoint) domains: for example, you might want to have a domain of numbers, a domain of geometric objects, a domain of functions from numbers to numbers, a domain of abelian groups, and so on.

Many-sorted logic provides this kind of framework. One starts with a list of "sorts"—the "sort" of an object indicates the "domain" it is supposed to inhabit. One then has variables and quantifiers for each sort, and (usually) an identity predicate for each sort. Functions and relations are also "typed" by the sorts of objects they can take as arguments. Otherwise, one keeps the usual rules of first-order logic, with versions of the quantifier-rules repeated for each sort.

For example, to study international relations we might choose a language with two sorts of objects, French citizens and German citizens. We might have a unary relation, "drinks wine," for objects of the first sort; another unary relation, "eats wurst," for objects of the second sort; and a binary relation, "forms a multinational married couple," which takes two arguments, where the first argument is of the first sort and the second argument is of the second sort. If we use variables a, b, c to range over French citizens and x, y, z to range over German citizens, then

$$\forall a \forall x[(MarriedTo(a, x) \rightarrow (DrinksWine(a) \vee \neg EatsWurst(x))]]$$

asserts that if any French person is married to a German, either the French person drinks wine or the German doesn't eat wurst.

Many-sorted logic can be embedded in first-order logic in a natural way, by lumping all the objects of the many-sorted domains together into one first-order domain, using unary predicate symbols to keep track of the sorts, and relativizing quantifiers. For example, the first-order language corresponding to the example above would have unary predicate symbolss "$German$" and "$French$," in addition to the other relations described, with the sort requirements erased. A sorted quantifier $\forall x\, \varphi$, where x is a variable of the German sort, translates to

$$\forall x\, (German(x) \rightarrow \varphi).$$

We need to add axioms that insure that the sorts are separate—e.g., $\forall x \neg(German(x) \wedge French(x))$—as well as axioms that guarantee that "drinks wine" only holds of objects satisfying the predicate $French(x)$, etc. With these conventions and axioms, it is not difficult to show that many-sorted sentences translate to first-order sentences, and many-sorted derivations translate to first-order deriva-

tions. Also, many-sorted structures "translate" to corresponding first-order structures and vice-versa, so we also have a completeness theorem for many-sorted logic.

5.3 Second-Order logic

The language of second-order logic allows one to quantify not just over a domain of individuals, but over relations on that domain as well. Given a first-order language $\mathcal{L}$, for each k one adds variables R which range over k-ary relations, and allows quantification over those variables. If R is a variable for a k-ary relation, and $t_1, \ldots, t_k$ are ordinary (first-order) terms, $R(t_1, \ldots, t_k)$ is an atomic formula. Otherwise, the set of formulas is defined just as in the case of first-order logic, with additional clauses for second-order quantification. Note that we only have the identity predicate for first-order terms: if R and S are relation variables of the same arity k, we can define $R = S$ to be an abbreviation for

$$\forall x_1 \ldots \forall x_k \, (R(x_1, \ldots, x_k) \leftrightarrow S(x_1, \ldots, x_k)).$$

The rules for second-order logic simply extend the quantifier rules to the new second order variables. Here, however, one has to be a little bit careful to explain how these variables interact with the predicate symbols of $\mathcal{L}$, and with formulas of $\mathcal{L}$ more generally. At the bare minimum, relation variables count as terms, so one has inferences of the form

$$\varphi(R) \vdash \exists R \, \varphi(R)$$

But if $\mathcal{L}$ is the language of arithmetic with a constant relation symbol $<$, one would also expect the following inference to be valid:

$$x < y \vdash \exists R \, R(x, y)$$

or for a given formula φ,

$$\varphi(x_1, \ldots, x_k) \vdash \exists R \, R(x_1, \ldots, x_k)$$

More generally, we might want to allow inferences of the form

$$\varphi[\lambda \vec{x}. \, \psi(\vec{x}) / R] \exists R \, \varphi$$

where $\varphi[\lambda \vec{x}. \, \psi(\vec{x}) / R]$ denotes the result of replacing every atomic formula of the form $Rt_1, \ldots, t_k$ in φ by $\psi(t_1, \ldots, t_k)$. This last rule is equivalent to having a *comprehension schema*, i.e., an axiom of the form

$$\exists R \, \forall x_1, \ldots, x_k \, (\varphi(x_1, \ldots, x_k) \leftrightarrow R(x_1, \ldots, x_k)),$$

one for each formula φ in the second-order language, in which R is not a free variable. (Exercise: show that if R is allowed to occur in φ, this schema is inconsistent!)

When logicians refer to the "axioms of second-order logic" they usually mean the minimal extension of first-order logic by second-order quantifier rules together with the comprehension schema. But it is often interesting to study weaker subsystems of these axioms and rules. For example, note that in its full generality the axiom schema of comprehension is *impredicative*: it allows one to assert the existence of a relation $R(x_1, \ldots, x_k)$ that is "defined" by a formula with second-order quantifiers; and these quantifiers range over the set of all such relations—a set which includes R itself! Around the turn of the twentieth century, a common reaction to Russell's paradox was to lay the blame on such definitions, and to avoid them in developing the foundations of mathematics. If one prohibits the use of second-order quantifiers in the formula φ, one has a *predicative* form of comprehension, which is somewhat weaker.

From the semantic point of view, one can think of a second-order structure as consisting of a first-order structure for the language, coupled with a set of relations on the domain over which the second-order quantifiers range (more precisely, for each k there is a set of relations of arity k). Of course, if comprehension is included in the proof system, then we have the added requirement that there are enough relations in the "second-order part" to satisfy the comprehension axioms—otherwise the proof system is not sound! One easy way to insure that there are enough relations around is to take the second-order part to consist of *all* the relations on the first-order part. Such a structure is called *full*, and, in a sense, is really the "intended structure" for the language. If we restrict our attention to full structures we have what is known as the *full* second-order semantics. In that case, specifying a structure boils down to specifying the first-order part, since the contents of the second-order part follow from that implicitly.

To summarize, there is some ambiguity when talking about second-order logic. In terms of the proof system, one might have in mind either

1. A "minimal" second-order proof system, together with some comprehension axioms.
2. The "standard" second-order proof system, with full comprehension.

In terms of the semantics, one might be interested in either

1. The "weak" semantics, where a structure consists of a first-order part, together with a second-order part big enough to satisfy the comprehension axioms.
2. The "standard" second-order semantics, in which one considers full structures only.

When logicians do not specify the proof system or the semantics they have in mind, they are usually refering to the second item on each list. The advantage to using this semantics is that, as we will see, it gives us categorical descriptions of many natural mathematical structures; at the same time, the proof system is quite strong, and sound for this semantics. The drawback is that the proof system is *not* complete for the semantics; in fact, *no* effectively given proof system is complete for the full second-order semantics. On the other hand, we will see that the proof system *is* complete for the weakened semantics; this implies that if a sentence is not provable, then there is *some* structure, not necessarily the full one, in which it is false.

The language of second-order logic is quite rich. One can identify unary relations with subsets of the domain, and so in particular you can quantify over these sets; for example, one can express induction for the natural numbers with a single axiom

$$\forall R\,((R(\mathsf{o}) \wedge \forall x\,(R(x) \to R(x'))) \to \forall x\,R(x)).$$

If one takes the language of arithmetic to have symbols $\mathsf{o}, \prime, +, \times$ and $<$, one can add the following axioms to describe their behavior:

1. $\forall x\,\neg x' = \mathsf{o}$
2. $\forall x\,\forall y\,(s(x) = s(y) \to x = y)$
3. $\forall x\,(x + 0) = x$
4. $\forall x\,\forall y\,(x + y') = (x + y)'$
5. $\forall x\,(x \times 0) = 0$
6. $\forall x\,\forall y\,(x \times y') = ((x \times y) + x)$
7. $\forall x\,\forall y\,(x < y \leftrightarrow \exists z\,y = (x + z'))$

It is not difficult to show that these axioms, together with the axiom of induction above, provide a categorical description of the structure $\mathfrak{N}$, the standard model of arithmetic, provided we are using the full second-order semantics. Given any structure $\mathfrak{A}$ in which these axioms are true, define a function f from $\mathbb{N}$ to the domain of $\mathfrak{A}$ using ordinary recursion on $\mathbb{N}$, so that $f(0) = \mathsf{o}^{\mathfrak{A}}$ and $f(x+1) = \prime^{\mathfrak{A}}(f(x))$. Using ordinary induction on $\mathbb{N}$ and the fact that axioms (1) and (2) hold in $\mathfrak{A}$, we see that f is injective. To see that f is surjective, let P be the set of elements of $|\mathfrak{A}|$ that are in the range of f. Since $\mathfrak{A}$ is full, P is in the second-order domain. By the construction of f, we know that $\mathsf{o}^{\mathfrak{A}}$ is in P, and that P is closed under $\prime^{\mathfrak{A}}$. The fact that the induction axiom holds in $\mathfrak{A}$ (in particular, for P) guarantees that P is equal to the entire first-order domain of $\mathfrak{A}$. This shows that f is a bijection. Showing that f is a homomorphism is no more difficult, using ordinary induction on $\mathbb{N}$ repeatedly.

In set-theoretic terms, a function is just a special kind of relation; for example, a unary function f can be identified with a binary relation R satisfying $\forall x \exists y\, R(x, y)$. As a result, one can quantify over functions too. Using the full semantics, one can then define the class of infinite structures to be the class of structures $\mathfrak{A}$ for which there is an injective function from the domain of $\mathfrak{A}$ to a proper subset of itself:

$$\exists f\, (\forall x\, \forall y\, (f(x) = f(y) \to x = y) \wedge \exists y\, \forall x\, f(x) \neq y).$$

The negation of this sentence then defines the class of finite structures.

In addition, one can define the class of well-orderings, by adding the following to the definition of a linear ordering:

$$\forall P\, (\exists x\, P(x) \to \exists x\, (P(x) \wedge \forall y\, (y < x \to \neg P(y)))).$$

This asserts that every nonempty set has a least element, modulo the identification of "set" with "one-place relation". For another example, one can express the notion of connectedness for graphs, by saying that there is no nontrivial separation of the vertices into disconnected parts:

$$\neg \exists A\, (\exists x\, A(x) \wedge \exists y\, \neg A(y) \wedge \forall [w][\forall z\, ((A(w) \wedge \neg A(z)) \to \neg R(w, z))]).$$

For yet another example, you might try as an exercise to define the class of finite structures whose domain has even size. More strikingly, one can provide a categorical description of the real numbers as a complete ordered field containing the rationals.

In short, second-order logic is much more expressive than first-order logic. That's the good news; now for the bad. We have already mentioned that there is no effective proof system that is complete for the full second-order semantics. For better or for worse, many of the properties of first-order logic are absent, including compactness and the Löwenheim-Skolem theorems.

On the other hand, if one is willing to give up the full second-order semantics in terms of the weaker one, then the minimal second-order proof system is complete for this semantics. In other words, if we read $\vdash$ as "proves in the minimal system" and $\models$ as "logically implies in the weaker semantics", we can show that whenever $\Gamma \models \varphi$ then $\Gamma \vdash \varphi$. If one wants to include specific comprehension axioms in the proof system, one has to restrict the semantics to second-order structures that satisfy these axioms: for example, if Δ consists of a set of comprehension axioms (possibly all of them), we have that if $\Gamma \cup \Delta \models \varphi$, then $\Gamma \cup \Delta \vdash \varphi$. In particular, if φ is not provable using the comprehension axioms we are considering, then there is a model of $\neg\varphi$ in which these comprehension axioms nonetheless hold.

The easiest way to see that the completeness theorem holds for the weaker semantics is to think of second-order logic as a many-sorted logic, as follows. One sort is interpreted as the ordinary "first-order" domain, and then for each

k we have a domain of "relations of arity k." We take the language to have built-in relation symbols "$true_k(R, x_1, \ldots, x_k)$" which is meant to assert that R holds of $x_1, \ldots, x_k$, where R is a variable of the sort "k-ary relation" and $x_1, \ldots, x_k$ are objects of the first-order sort.

With this identification, the weak second-order semantics is essentially the usual semantics for many-sorted logic; and we have already observed that many-sorted logic can be embedded in first-order logic. Modulo the translations back and forth, then, the weaker conception of second-order logic is really a form of first-order logic in disguise, where the domain contains both "objects" and "relations" governed by the appropriate axioms.

5.4 Higher-Order logic

Passing from first-order logic to second-order logic enabled us to talk about sets of objects in the first-order domain, within the formal language. Why stop there? For example, third-order logic should enable us to deal with sets of sets of objects, or perhaps even sets which contain both objects and sets of objects. And fourth-order logic will let us talk about sets of objects of that kind. As you may have guessed, one can iterate this idea arbitrarily.

In practice, higher-order logic is often formulated in terms of functions instead of relations. (Modulo the natural identifications, this difference is inessential.) Given some basic "sorts" A, B, C, ...(which we will now call "types"), we can create new ones by stipulating

> If σ and τ are finite types then so is $\sigma \to \tau$.

Think of types as syntactic "labels," which classify the objects we want in our domain; $\sigma \to \tau$ describes those objects that are functions which take objects of type σ to objects of type τ. For example, we might want to have a type Ω of truth values, "true" and "false," and a type $\mathbb{N}$ of natural numbers. In that case, you can think of objects of type $\mathbb{N} \to \Omega$ as unary relations, or subsets of $\mathbb{N}$; objects of type $\mathbb{N} \to \mathbb{N}$ are functions from natural numers to natural numbers; and objects of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ are "functionals," that is, higher-type functions that take functions to numbers.

As in the case of second-order logic, one can think of higher-order logic as a kind of many-sorted logic, where there is a sort for each type of object we want to consider. But it is usually clearer just to define the syntax of higher-type logic from the ground up. For example, we can define a set of finite types inductively, as follows:

1. $\mathbb{N}$ is a finite type.
2. If σ and τ are finite types, then so is $\sigma \to \tau$.
3. If σ and τ are finite types, so is $\sigma \times \tau$.

Intuitively, $\mathbb{N}$ denotes the type of the natural numbers, $\sigma \to \tau$ denotes the type of functions from σ to τ, and $\sigma \times \tau$ denotes the type of pairs of objects, one from σ and one from τ. We can then define a set of terms inductively, as follows:

1. For each type σ, there is a stock of variables $x, y, z, \ldots$ of type σ
2. $\mathsf{0}$ is a term of type $\mathbb{N}$
3. S (successor) is a term of type $\mathbb{N} \to \mathbb{N}$
4. If s is a term of type σ, and t is a term of type $\mathbb{N} \to (\sigma \to \sigma)$, then R_{st} is a term of type $\mathbb{N} \to \sigma$
5. If s is a term of type $\tau \to \sigma$ and t is a term of type τ, then $s(t)$ is a term of type σ
6. If s is a term of type σ and x is a variable of type τ, then $\lambda x.\, s$ is a term of type $\tau \to \sigma$.
7. If s is a term of type σ and t is a term of type τ, then $\langle s, t \rangle$ is a term of type $\sigma \times \tau$.
8. If s is a term of type $\sigma \times \tau$ then $p_1(s)$ is a term of type σ and $p_2(s)$ is a term of type τ.

Intuitively, R_{st} denotes the function defined recursively by

$$\begin{aligned} R_{st}(0) &= s \\ R_{st}(x+1) &= t(x, R_{st}(x)), \end{aligned}$$

$\langle s, t \rangle$ denotes the pair whose first component is s and whose second component is t, and $p_1(s)$ and $p_2(s)$ denote the first and second elements ("projections") of s. Finally, $\lambda x.\, s$ denotes the function f defined by

$$f(x) = s$$

for any x of type σ; so item (6) gives us a form of comprehension, enabling us to define functions using terms. Formulas are built up from identity predicate statements $s = t$ between terms of the same type, the usual propositional connectives, and higher-type quantification. One can then take the axioms of the system to be the basic equations governing the terms defined above, together with the usual rules of logic with quantifiers and identity predicate.

If one augments the finite type system with a type Ω of truth values, one has to include axioms which govern its use as well. In fact, if one is clever, one can get rid of complex formulas entirely, replacing them with terms of type Ω! The proof system can then be modified accordingly. The result is essentially the *simple theory of types* set forth by Alonzo Church in the 1930s.

As in the case of second-order logic, there are different versions of higher-type semantics that one might want to use. In the full version, variables of type $\sigma \to \tau$ range over the set of *all* functions from the objects of type σ to objects of type τ. As you might expect, this semantics is too strong to admit a complete, effective proof system. But one can consider a weaker semantics, in which a structure consists of sets of elements T_τ for each type τ, together with appropriate operations for application, projection, etc. If the details are carried out correctly, one can obtain completeness theorems for the kinds of proof systems described above.

Higher-type logic is attractive because it provides a framework in which we can embed a good deal of mathematics in a natural way: starting with $\mathbb{N}$, one can define real numbers, continuous functions, and so on. It is also particularly attractive in the context of intuitionistic logic, since the types have clear "constructive" intepretations. In fact, one can develop constructive versions of higher-type semantics (based on intuitionistic, rather than classical logic) that clarify these constructive interpretations quite nicely, and are, in many ways, more interesting than the classical counterparts.

5.5 Intuitionistic logic

In constrast to second-order and higher-order logic, intuitionistic first-order logic represents a restriction of the classical version, intended to model a more "constructive" kind of reasoning. The following examples may serve to illustrate some of the underlying motivations.

Suppose someone came up to you one day and announced that they had determined a natural number x, with the property that if x is prime, the Riemann hypothesis is true, and if x is composite, the Riemann hypothesis is false. Great news! Whether the Riemann hypothesis is true or not is one of the big open questions of mathematics, and here they seem to have reduced the problem to one of calculation, that is, to the determination of whether a specific number is prime or not.

What is the magic value of x? They describe it as follows: x is the natural number that is equal to 7 if the Riemann hypothesis is true, and 9 otherwise.

Angrily, you demand your money back. From a classical point of view, the description above does in fact determine a unique value of x; but what you really want is a value of x that is given *explicitly*.

To take another, perhaps less contrived example, consider the following question. We know that it is possible to raise an irrational number to a rational power, and get a rational result. For example, $\sqrt{2}^2 = 2$. What is less clear is whether or not it is possible to raise an irrational number to an *irrational* power, and get a rational result. The following theorem answers this in the affirmative:

Theorem 5.1. *There are irrational numbers a and b such that a^b is rational.*

Proof. Consider $\sqrt{2}^{\sqrt{2}}$. If this is rational, we are done: we can let $a = b = \sqrt{2}$. Otherwise, it is irrational. Then we have

$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = \sqrt{2}^2 = 2,$$

which is certainly rational. So, in this case, let a be $\sqrt{2}^{\sqrt{2}}$, and let b be $\sqrt{2}$. □

Does this constitute a valid proof? Most mathematicians feel that it does. But again, there is something a little bit unsatisfying here: we have proved the existence of a pair of real numbers with a certain property, without being able to say *which* pair of numbers it is. It is possible to prove the same result, but in such a way that the pair a, b *is* given in the proof: take $a = \sqrt{3}$ and $b = \log_3 4$. Then

$$a^b = \sqrt{3}^{\log_3 4} = 3^{1/2\cdot\log_3 4} = (3^{\log_3 4})^{1/2} = 4^{1/2} = 2,$$

since $3^{\log_3 x} = x$.

Intuitionistic logic is designed to model a kind of reasoning where moves like the one in the first proof are disallowed. Proving the existence of an x satisfying $\varphi(x)$ means that you have to give a specific x, and a proof that it satisfies φ, like in the second proof. Proving that φ or ψ holds requires that you can prove one or the other.

Formally speaking, intuitionistic first-order logic is what you get if you omit restrict a proof system for first-order logic in a certain way. Similarly, there are intuitionistic versions of second-order or higher-order logic. From the mathematical point of view, these are just formal deductive systems, but, as already noted, they are intended to model a kind of mathematical reasoning. One can take this to be the kind of reasoning that is justified on a certain philosophical view of mathematics (such as Brouwer's intuitionism); one can take it to be a kind of mathematical reasoning which is more "concrete" and satisfying (along the lines of Bishop's constructivism); and one can argue about whether or not the formal description captures the informal motivation. But whatever philosophical positions we may hold, we can study intuitionistic logic as a formally presented logic; and for whatever reasons, many mathematical logicians find it interesting to do so.

There is an informal constructive interpretation of the intuitionist connectives, usually known as the Brouwer-Heyting-Kolmogorov interpretation. It runs as follows: a proof of $\varphi \wedge \psi$ consists of a proof of φ paired with a proof of ψ; a proof of $\varphi \vee \psi$ consists of either a proof of φ, or a proof of ψ, where we have explicit information as to which is the case; a proof of $\varphi \to \psi$ consists of a procedure, which transforms a proof of φ to a proof of ψ; a proof of $\forall x\, \varphi(x)$ consists of a procedure which returns a proof of $\varphi(x)$ for any value of x; and a proof of $\exists x\, \varphi(x)$ consists of a value of x, together with a proof that this value satisfies φ. One can describe the interpretation in computational terms known as the "Curry-Howard isomorphism" or the "formulas-as-types

paradigm": think of a formula as specifying a certain kind of data type, and proofs as computational objects of these data types that enable us to see that the corresponding formula is true.

Intuitionistic logic is often thought of as being classical logic "minus" the law of the excluded middle. This following theorem makes this more precise.

Theorem 5.2. *Intuitionistically, the following axiom schemata are equivalent:*

1. $(\varphi \to \bot) \to \neg\varphi$.
2. $\varphi \vee \neg\varphi$
3. $\neg\neg\varphi \to \varphi$

Obtaining instances of one schema from either of the others is a good exercise in intuitionistic logic.

The first deductive systems for intuitionistic propositional logic, put forth as formalizations of Brouwer's intuitionism, are due, independently, to Kolmogorov, Glivenko, and Heyting. The first formalization of intuitionistic first-order logic (and parts of intuitionist mathematics) is due to Heyting. Though a number of classically valid schemata are not intuitionistically valid, many are.

The *double-negation translation* describes an important relationship between classical and intuitionist logic. It is defined inductively follows (think of φ^N as the "intuitionist" translation of the classical formula φ):

$$\begin{aligned}
\varphi^N &\equiv \neg\neg\varphi \quad \text{for atomic formulas } \varphi \\
(\varphi \wedge \psi)^N &\equiv (\varphi^N \wedge \psi^N) \\
(\varphi \vee \psi)^N &\equiv \neg\neg(\varphi^N \vee \psi^N) \\
(\varphi \to \psi)^N &\equiv (\varphi^N \to \psi^N) \\
(\forall x\, \varphi)^N &\equiv \forall x\, \varphi^N \\
(\exists x\, \varphi)^N &\equiv \neg\neg\exists x\, \varphi^N
\end{aligned}$$

Kolmogorov and Glivenko had versions of this translation for propositional logic; for predicate logic, it is due to Gödel and Gentzen, independently. We have

Theorem 5.3. *1. $\varphi \leftrightarrow \varphi^N$ is provable classically*

2. If φ is provable classically, then φ^N is provable intuitionistically.

We can now envision the following dialogue. Classical mathematician: "I've proved φ!" Intuitionist mathematician: "Your proof isn't valid. What you've really proved is φ^N." Classical mathematician: "Fine by me!" As far as

the classical mathematician is concerned, the intuitionist is just splitting hairs, since the two are equivalent. But the intuitionist insists there is a difference.

Note that the above translation concerns pure logic only; it does not address the question as to what the appropriate *nonlogical* axioms are for classical and intuitionistic mathematics, or what the relationship is between them. But the following slight extension of the theorem above provides some useful information:

Theorem 5.4. *If Γ proves φ classically, Γ^N proves φ^N intuitionistically.*

In other words, if φ is provable from some hypotheses classically, then φ^N is provable from their double-negation translations.

To show that a sentence or propositional formula is intuitionistically valid, all you have to do is provide a proof. But how can you show that it is not valid? For that purpose, we need a semantics that is sound, and preferrably complete. A semantics due to Kripke nicely fits the bill.

We can play the same game we did for classical logic: define the semantics, and prove soundness and completeness. It is worthwhile, however, to note the following distinction. In the case of classical logic, the semantics was the "obvious" one, in a sense implicit in the meaning of the connectives. Though one can provide some intuitive motivation for Kripke semantics, the latter does not offer the same feeling of inevitability. In addition, the notion of a classical structure is a natural mathematical one, so we can either take the notion of a structure to be a tool for studying classical first-order logic, or take classical first-order logic to be a tool for studying mathematical structures. In contrast, Kripke structures can only be viewed as a logical construct; they don't seem to have independent mathematical interest.

A Kripke structure for a propositional langauge consists of a partial order $\mathfrak{P}$ with a least element, and an "monotone" assignment of propositional variables to the elements of $\mathfrak{P}$. The intuition is that the elements of $\mathfrak{P}$ represent "worlds," or "states of knowledge"; an element $p \geq q$ represents a "possible future state" of q; and the propositional variables assigned to p are the propositions that are known to be true in state p. The forcing relation $\mathfrak{P}, p \Vdash \varphi$ then extends this relationship to arbitrary formulas in the language; read $\mathfrak{P}, p \Vdash \varphi$ as "φ is true in state p." The relationship is defined inductively, as follows:

1. $\mathfrak{P}, p \Vdash p_i$ iff p_i is one of the propositional variables assigned to p.
2. $\mathfrak{P}, p \nVdash \bot$.
3. $\mathfrak{P}, p \Vdash (\varphi \wedge \psi)$ iff $\mathfrak{P}, p \Vdash \varphi$ and $\mathfrak{P}, p \Vdash \psi$.
4. $\mathfrak{P}, p \Vdash (\varphi \vee \psi)$ iff $\mathfrak{P}, p \Vdash \varphi$ or $\mathfrak{P}, p \Vdash \psi$.
5. $\mathfrak{P}, p \Vdash (\varphi \to \psi)$ iff, whenever $q \geq p$ and $\mathfrak{P}, q \Vdash \varphi$, then $\mathfrak{P}, q \Vdash \psi$.

It is a good exercise to try to show that $\neg(p \wedge q) \rightarrow (\neg p \vee \neg q)$ is not intuitionistically valid, by cooking up a Kripke structure that provides a counterexample.

5.6 Modal Logics

Consider the following example of a conditional sentence:

> If Jeremy is alone in that room, then he is drunk and naked and dancing on the chairs.

This is an example of a conditional assertion that may be materially true but nonetheless misleading, since it seems to suggest that there is a stronger link between the antecedent and conclusion other than simply that either the antecedent is false or the consequent true. That is, the wording suggests that the claim is not only true in this particular world (where it may be trivially true, because Jeremy is not alone in the room), but that, moreover, the conclusion *would have* been true *had* the antecedent been true. In other words, one can take the assertion to mean that the claim is true not just in this world, but in any "possible" world; or that it is *necessarily* true, as opposed to just true in this particular world.

Modal logic was designed to make sense of this kind of necessity. One obtains modal propositional logic from ordinary propositional logic by adding a box operator; which is to say, if φ is a formula, so is $\Box\varphi$. Intuitively, $\Box\varphi$ asserts that φ is *necessarily* true, or true in any possible world. $\Diamond\varphi$ is usually taken to be an abbreviation for $\neg\Box\neg\varphi$, and can be read as asserting that φ is *possibly* true. Of course, modality can be added to predicate logic as well.

Kripke structures can be used to provide a semantics for modal logic; in fact, Kripke first designed this semantics with modal logic in mind. Rather than restricting to partial orders, more generally one has a set of "possible worlds," P, and a binary "accessibility" relation $R(x, y)$ between worlds. Intuitively, $R(p, q)$ asserts that the world q is compatible with p; i.e., if we are "in" world p, we have to entertain the possibility that the world could have been like q.

Modal logic is sometimes called an "intensional" logic, as opposed to an "extensional" one. The intended semantics for an extensional logic, like classical logic, will only refer to a single world, the "actual" one; while the semantics for an "intensional" logic relies on a more elaborate ontology. In addition to structureing necessity, one can use modality to structure other linguistic constructions, reinterpreting $\Box$ and $\Diamond$ according to the application. For example:

1. In provability logic, $\Box\varphi$ is read "φ is provable" and $\Diamond\varphi$ is read "φ is consistent."

2. In epistemic logic, one might read $\Box\varphi$ as "I know φ" or "I believe φ."

3. In temporal logic, one can read $\Box\varphi$ as "φ is always true" and $\Diamond\varphi$ as "φ is sometimes true."

One would like to augment logic with rules and axioms dealing with modality. For example, the system **S4** consists of the ordinary axioms and rules of propositional logic, together with the following axioms:

$$\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$$
$$\Box\varphi \to \varphi$$
$$\Box\varphi \to \Box\Box\varphi$$

as well as a rule, "from φ conclude $\Box\varphi$." **S5** adds the following axiom:

$$\Diamond\varphi \to \Box\Diamond\varphi$$

Variations of these axioms may be suitable for different applications; for example, S5 is usually taken to characterize the notion of logical necessity. And the nice thing is that one can usually find a semantics for which the proof system is sound and complete by restricting the accessibility relation in the Kripke structures in natural ways. For example, **S4** corresponds to the class of Kripke structures in which the accessibility relation is reflexive and transitive. **S5** corresponds to the class of Kripke structures in which the accessibility relation is *universal*, which is to say that every world is accessible from every other; so $\Box\varphi$ holds if and only if φ holds in every world.

5.7 Other Logics

As you may have gathered by now, it is not hard to design a new logic. You too can create your own a syntax, make up a deductive system, and fashion a semantics to go with it. You might have to be a bit clever if you want the proof system to be complete for the semantics, and it might take some effort to convince the world at large that your logic is truly interesting. But, in return, you can enjoy hours of good, clean fun, exploring your logic's mathematical and computational properties.

Recent decades have witnessed a veritable explosion of formal logics. Fuzzy logic is designed to model reasoning about vague properties. Probabilistic logic is designed to model reasoning about uncertainty. Default logics and nonmonotonic logics are designed to model defeasible forms of reasoning, which is to say, "reasonable" inferences that can later be overturned in the face of new information. There are epistemic logics, designed to model reasoning about knowledge; causal logics, designed to model reasoning about causal relationships; and even "deontic" logics, which are designed to model reasoning about moral and ethical obligations. Depending on whether the primary

motivation for introducing these systems is philosophical, mathematical, or computational, you may find such creatures studies under the rubric of mathematical logic, philosophical logic, artificial intelligence, cognitive science, or elsewhere.

The list goes on and on, and the possibilities seem endless. We may never attain Leibniz' dream of reducing all of human reason to calculation—but that can't stop us from trying.

Part II

Computability

Chapter 6

Recursive Functions

6.1 Introduction

In order to develop a mathematical theory of computability, one has to first of all develop a *model* of computability. We now think of computability as the kind of thing that computers do, and computers work with symbols. But at the beginning of the development of theories of computability, the paradigmatic example of computation was *numerical* computation. Mathematicians were always interested in number-theoretic functions, i.e., functions $f : \mathbb{N}^n \to \mathbb{N}$ that can be computed. So it is not surprising that at the beginning of the theory of computability, it was such functions that were studied. The most familiar examples of computable numerical functions, such as addition, multiplication, exponentiation (of natural numbers) share an interesting feature: they can be defined *recursively*. It is thus quite natural to attempt a general definition of *computable function* on the basis of recursive definitions. Among the many possible ways to define number-theoretic functions recursively, one particulalry simple pattern of definition here becomes central: so-called *primitive recursion*.

In addition to computable functions, we might be interested in computable sets and relations. A set is computable if we can compute the answer to whether or not a given number is an element of the set, and a relation is computable iff we can compute whether or not a tuple $\langle n_1, \ldots, n_k \rangle$ is an element of the relation. By considering the *characteristic function* of a set or relation, discussion of computable sets and relations can be subsumed under that of computable functions. Thus we can define primitive recursive relations as well, e.g., the relation "n evenly divides m" is a primitive recursive relation.

Primitive recursive functions—those that can be defined using just primitive recursion—are not, however, the only computable number-theoretic functions. Many generalizations of primitive recursion have been considered, but the most powerful and widely-accepted additional way of computing functions is by unbounded search. This leads to the definition of *partial recur-*

sive functions, and a related definition to *general recursive functions*. General recursive functions are computable and total, and the definition characterizes exactly the partial recursive functions that happen to be total. Recursive functions can simulate every other model of computation (Turing machines, lambda calculus, etc.) and so represent one of the many accepted models of computation.

6.2 Primitive Recursion

Suppose we specify that a certain function l from $\mathbb{N}$ to $\mathbb{N}$ satisfies the following two clauses:

$$\begin{aligned} l(0) &= 1 \\ l(x+1) &= 2 \cdot l(x). \end{aligned}$$

It is pretty clear that there is only one function, l, that meets these two criteria. This is an instance of a *definition by primitive recursion*. We can define even more fundamental functions like addition and multiplication by

$$\begin{aligned} f(x,0) &= x \\ f(x,y+1) &= f(x,y)+1 \end{aligned}$$

and

$$\begin{aligned} g(x,0) &= 0 \\ g(x,y+1) &= f(g(x,y),x). \end{aligned}$$

Exponentiation can also be defined recursively, by

$$\begin{aligned} h(x,0) &= 1 \\ h(x,y+1) &= g(h(x,y),x). \end{aligned}$$

We can also compose functions to build more complex ones; for example,

$$\begin{aligned} k(x) &= x^x + (x+3) \cdot x \\ &= f(h(x,x), g(f(x,3),x)). \end{aligned}$$

Remember that the *arity* of a function is the number of arguments. For convenience, we will consider a constant, like 7, to be a 0-ary function. (Send it zero arguments, and it returns 7.) The set of *primitive recursive functions* is the set of functions from $\mathbb{N}$ to $\mathbb{N}$ that you get if you start with 0 and the successor function, $S(x) = x+1$, and iterate the two operations above, primitive recursion and composition. The idea is that primitive recursive functions are defined in a straightforward and explicit way, so that it is intuitively clear that each one can be computed using finite means.

Definition 6.1. If f is a k-ary function and $g_0, \ldots, g_{k-1}$ are l-ary functions on the natural numbers, the *composition* of f with $g_0, \ldots, g_{k-1}$ is the l-ary function h defined by

$$h(x_0, \ldots, x_{l-1}) = f(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{k-1}(x_0, \ldots, x_{l-1})).$$

Definition 6.2. If $f(z_0, \ldots, z_{k-1})$ is a k-ary function and $g(x, y, z_0, \ldots, z_{k-1})$ is a $k+2$-ary function, then the function defined by *primitive recursion from f and g* is the $k+1$-ary function h, defined by the equations

$$\begin{aligned} h(0, z_0, \ldots, z_{k-1}) &= f(z_0, \ldots, z_{k-1}) \\ h(x+1, z_0, \ldots, z_{k-1}) &= g(x, h(x, z_0, \ldots, z_{k-1}), z_0, \ldots, z_{k-1}) \end{aligned}$$

In addition to the constant, 0, and the successor function, $S(x)$, we will include among primitive recursive functions the projection functions,

$$P^n_i(x_0, \ldots, x_{n-1}) = x_i,$$

for each natural number n and $i < n$. In the end, we have the following:

Definition 6.3. The set of primitive recursive functions is the set of functions of various arities from the set of natural numbers to the set of natural numbers, defined inductively by the following clauses:

1. The constant, 0, is primitive recursive.
2. The successor function, S, is primitive recursive.
3. Each projection function P^n_i is primitive recursive.
4. If f is a k-ary primitive recursive function and $g_0, \ldots, g_{k-1}$ are l-ary primitive recursive functions, then the composition of f with $g_0, \ldots, g_{k-1}$ is primitive recursive.
5. If f is a k-ary primitive recursive function and g is a $k+2$-ary primitive recursive function, then the function defined by primitive recursion from f and g is primitive recursive.

Put more concisely, the set of primitive recursive functions is the smallest set containing the constant 0, the successor function, and projection functions, and closed under composition and primitive recursion.

Another way of describing the set of primitive recursive functions keeps track of the "stage" at which a function enters the set. Let S_0 denote the set of starting functions: zero, successor, and the projections. Once S_i has been defined, let S_{i+1} be the set of all functions you get by applying a single instance of composition or primitive recursion to functions in S_i. Then

$$S = \bigcup_{i \in \mathbb{N}} S_i$$

is the set of primitive recursive functions

Our definition of composition may seem too rigid, since $g_0, \ldots, g_{k-1}$ are all required to have the same arity, l. But adding the projection functions provides the desired flexibility. For example, suppose f and g are ternary functions and h is the binary function defined by

$$h(x, y) = f(x, g(x, x, y), y).$$

Then the definition of h can be rewritten with the projection functions, as

$$h(x, y) = f(P^2_0(x, y), g(P^2_0(x, y), P^2_0(x, y), P^2_1(x, y)), P^2_1(x, y)).$$

Then h is the composition of f with P^2_0, l, P^2_1, where

$$l(x, y) = g(P^2_0(x, y), P^2_0(x, y), P^2_1(x, y)),$$

i.e., l is the composition of g with P^2_0, P^2_0, P^2_1.

For another example, let us consider one of the informal examples given above, namely, addition. This is described recursively by the following two equations:

$$\begin{aligned} x + 0 &= x \\ x + (y + 1) &= S(x + y). \end{aligned}$$

In other words, addition is the function g defined recursively by the equations

$$\begin{aligned} g(0, x) &= x \\ g(y + 1, x) &= S(g(y, x)). \end{aligned}$$

But even this is not a strict primitive recursive definition; we need to put it in the form

$$\begin{aligned} g(0, x) &= k(x) \\ g(y + 1, x) &= h(y, g(y, x), x) \end{aligned}$$

for some 1-ary primitive recursive function k and some 3-ary primitive recursive function h. We can take k to be P^1_0, and we can define h using composition,

$$h(y, w, x) = S(P^3_1(y, w, x)).$$

The function h, being the composition of basic primitive recursive functions, is primitive recursive; and hence so is g. (Note that, strictly speaking, we have defined the function $g(y, x)$ meeting the recursive specification of $x + y$; in other words, the variables are in a different order. Luckily, addition is commutative, so here the difference is not important; otherwise, we could define the function g' by

$$g'(x, y) = g(P^2_1(y, x)), P^2_0(y, x)) = g(y, x),$$

using composition.)

One advantage to having the precise description of the primitive recursive functions is that we can be systematic in describing them. For example, we can assign a "notation" to each such function, as follows. Use symbols 0, S, and P^n_i for zero, successor, and the projections. Now suppose f is defined by composition from a k-ary function h and l-ary functions $g_0, \ldots, g_{k-1}$, and we have assigned notations $H, G_0, \ldots, G_{k-1}$ to the latter functions. Then, using a new symbol $\mathrm{Comp}_{k,l}$, we can denote the function f by $\mathrm{Comp}_{k,l}[H, G_0, \ldots, G_{k-1}]$. For the functions defined by primitive recursion, we can use analogous notations of the form $\mathrm{Rec}_k[G, H]$, where k denotes that arity of the function being defined. With this setup, we can denote the addition function by

$$\mathrm{Rec}_2[P^1_0, \mathrm{Comp}_{1,3}[S, P^3_1]].$$

Having these notations sometimes proves useful.

6.3 Primitive Recursive Functions are Computable

Suppose a function h is defined by primitive recursion

$$\begin{aligned} h(0, \vec{z}) &= f(\vec{z}) \\ h(x+1, \vec{z}) &= g(x, h(x, \vec{z}), \vec{z}) \end{aligned}$$

and suppose the functions f and g are computable. Then $h(0, \vec{z})$ can obviously be computed, since it is just $f(\vec{z})$ which we assume is computable. $h(1, \vec{z})$ can then also be computed, since $1 = 0 + 1$ and so $h(1, \vec{z})$ is just

$$g(0, h(0, \vec{z}), \vec{z}) = g(0, f(\vec{z}), \vec{z}).$$

We can go on in this way and compute

$$\begin{aligned} h(2, \vec{z}) &= g(1, g(0, f(\vec{z}), \vec{z}), \vec{z}) \\ h(3, \vec{z}) &= g(2, g(1, g(0, f(\vec{z}), \vec{z}), \vec{z}), \vec{z}) \\ h(4, \vec{z}) &= g(3, g(2, g(1, g(0, f(\vec{z}), \vec{z}), \vec{z}), \vec{z}), \vec{z}) \\ &\vdots \end{aligned}$$

Thus, to compute $h(x, \vec{z})$ in general, successively compute $h(0, \vec{z})$, $h(1, \vec{z})$, ..., until we reach $h(x, \vec{z})$.

Thus, primitive recursion yields a new computable function if the functions f and g are computable. Composition of functions also results in a computable function if the functions f and g_i are computable.

Since the basic functions 0, S, and P^n_i are computable, and composition and primitive recursion yield computable functions from computable functions, his means that every primitive recursive function is computable.

6.4 Examples of Primitive Recursive Functions

Here are some examples of primitive recursive functions:

1. Constants: for each natural number n, n is a 0-ary primitive recursive function, since it is equal to $S(S(\dots S(0)))$.
2. The identity function: $\mathrm{id}(x) = x$, i.e. P^1_0
3. Addition, $x + y$
4. Multiplication, $x \cdot y$
5. Exponentiation, x^y (with 0^0 defined to be 1)
6. Factorial, $x!$
7. The predecessor function, $\mathrm{pred}(x)$, defined by
$$\mathrm{pred}(0) = 0, \quad \mathrm{pred}(x+1) = x$$
8. Truncated subtraction, $x \mathbin{\dot{-}} y$, defined by
$$x \mathbin{\dot{-}} 0 = x, \quad x \mathbin{\dot{-}} (y+1) = \mathrm{pred}(x \mathbin{\dot{-}} y)$$
9. Maximum, $\max(x, y)$, defined by
$$\max(x, y) = x + (y \mathbin{\dot{-}} x)$$
10. Minimum, $\min(x, y)$
11. Distance between x and y, $|x - y|$

The set of primitive recursive functions is further closed under the following two operations:

1. Finite sums: if $f(x, \vec{z})$ is primitive recursive, then so is the function
$$g(y, \vec{z}) = \sum_{x=0}^{y} f(x, \vec{z}).$$
2. Finite products: if $f(x, \vec{z})$ is primitive recursive, then so is the function
$$h(y, \vec{z}) = \prod_{x=0}^{y} f(x, \vec{z}).$$

For example, finite sums are defined recursively by the equations

$$g(0, \vec{z}) = f(0, \vec{z}), \quad g(y+1, \vec{z}) = g(y, \vec{z}) + f(y+1, \vec{z}).$$

We can also define boolean operations, where 1 stands for true, and 0 for false:

1. Negation, $\text{not}(x) = 1 \dot{-} x$
2. Conjunction, $\text{and}(x, y) = x \cdot y$

Other classical boolean operations like $\text{or}(x, y)$ and $\text{ifthen}(x, y)$ can be defined from these in the usual way.

6.5 Primitive Recursive Relations

Definition 6.4. A relation $R(\vec{x})$ is said to be primitive recursive if its characteristic function,

$$\chi_R(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive.

In other words, when one speaks of a primitive recursive relation $R(\vec{x})$, one is referring to a relation of the form $\chi_R(\vec{x}) = 1$, where χ_R is a primitive recursive function which, on any input, returns either 1 or 0. For example, the relation $\text{Zero}(x)$, which holds if and only if $x = 0$, corresponds to the function χ_{Zero}, defined using primitive recursion by

$$\chi_{\text{Zero}}(0) = 1, \quad \chi_{\text{Zero}}(x + 1) = 0.$$

It should be clear that one can compose relations with other primitive recursive functions. So the following are also primitive recursive:

1. The equality relation, $x = y$, defined by $\text{Zero}(|x - y|)$
2. The less-than relation, $x \leq y$, defined by $\text{Zero}(x \dot{-} y)$

Furthermore, the set of primitive recursive relations is closed under boolean operations:

1. Negation, $\neg P$
2. Conjunction, $P \wedge Q$
3. Disjunction, $P \vee Q$
4. Implication, $P \to Q$

are all primitive recursive, if P and Q are.

One can also define relations using bounded quantification:

1. Bounded universal quantification: if $R(x, \vec{z})$ is a primitive recursive relation, then so is the relation

   $$\forall x < y \, R(x, \vec{z})$$

   which holds if and only if $R(x, \vec{z})$ holds for every x less than y.

2. Bounded existential quantification: if $R(x, \vec{z})$ is a primitive recursive relation, then so is

$$\exists x < y \, R(x, \vec{z}).$$

By convention, we take expressions of the form $\forall x < 0 \, R(x, \vec{z})$ to be true (for the trivial reason that there *are* no x less than 0) and $\exists x < 0 \, R(x, \vec{z})$ to be false. A universal quantifier functions just like a finite product; it can also be defined directly by

$$g(0, \vec{z}) = 1, \quad g(y+1, \vec{z}) = \chi_{\text{and}}(g(y, \vec{z}), \chi_R(y, \vec{z})).$$

Bounded existential quantification can similarly be defined using or. Alternatively, it can be defined from bounded universal quantification, using the equivalence, $\exists x < y \, \varphi(x) \leftrightarrow \neg \forall x < y \, \neg \varphi(x)$. Note that, for example, a bounded quantifier of the form $\exists x \leq y$ is equivalent to $\exists x < y + 1$.

Another useful primitive recursive function is:

1. The conditional function, $\text{cond}(x, y, z)$, defined by

$$\text{cond}(x, y, z) = \begin{cases} y & \text{if } x = 0 \\ z & \text{otherwise} \end{cases}$$

This is defined recursively by

$$\text{cond}(0, y, z) = y, \quad \text{cond}(x+1, y, z) = z.$$

One can use this to justify:

1. Definition by cases: if $g_0(\vec{x}), \ldots, g_m(\vec{x})$ are functions, and $R_1(\vec{x}), \ldots, R_{m-1}(\vec{x})$ are relations, then the function f defined by

$$f(\vec{x}) = \begin{cases} g_0(\vec{x}) & \text{if } R_0(\vec{x}) \\ g_1(\vec{x}) & \text{if } R_1(\vec{x}) \text{ and not } R_0(\vec{x}) \\ \vdots & \\ g_{m-1}(\vec{x}) & \text{if } R_{m-1}(\vec{x}) \text{ and none of the previous hold} \\ g_m(\vec{x}) & \text{otherwise} \end{cases}$$

   is also primitive recursive.

When $m = 1$, this is just the function defined by

$$f(\vec{x}) = \text{cond}(\chi_{\neg R_0}(\vec{x}), g_0(\vec{x}), g_1(\vec{x})).$$

For m greater than 1, one can just compose definitions of this form.

6.6 Bounded Minimization

Proposition 6.5. *If $R(x, \vec{z})$ is primitive recursive, so is the function $m_R(y, \vec{z})$ which returns the least x less than y such that $R(x, \vec{z})$ holds, if there is one, and 0 otherwise. We will write the function m_R as*

$$\min x < y \; R(x, \vec{z}),$$

Proof. Note than there can be no $x < 0$ such that $R(x, \vec{z})$ since there is no $x < 0$ at all. So $m_R(x, 0) = 0$.

In case the bound is $y + 1$ we have three cases: (a) There is an $x < y$ such that $R(x, \vec{z})$, in which case $m_R(y + 1, \vec{z}) = m_R(y, \vec{z})$. (b) There is no such x but $R(y, \vec{z})$ holds, then $m_R(y + 1, \vec{z}) = y$. (c) There is no $x < y + 1$ such that $R(x, \vec{z})$, then $m_R(y + 1, \vec{z}) = 0$. So,

$$m_R(0, \vec{z}) = 0$$

$$m_R(y + 1, \vec{z}) = \begin{cases} m_R(y, \vec{z}) & \text{if } \exists x < y \, R(x, \vec{z}) \\ y & \text{otherwise, provided } R(y, \vec{z}) \\ 0 & \text{otherwise.} \end{cases}$$

□

The choice of "0 otherwise" is somewhat arbitrary. It is in fact even easier to recursively define the function m'_R which returns the least x less than y such that $R(x, \vec{z})$ holds, and $y + 1$ otherwise. When we use min, however, we will always know that the least x such that $R(x, \vec{z})$ exists and is less than y. Thus, in practice, we will not have to worry about the possibility that if $\min x < y \; R(x, \vec{z}) = 0$ we do not know if that value indicates that $R(0, \vec{z})$ or that for no $x < y$, $R(x, \vec{z})$. As with bounded quantification, $\min x \leq y \; \ldots$ can be understood as $\min x < y + 1 \; \ldots$.

All this provides us with a good deal of machinery to show that natural functions and relations are primitive recursive. For example, the following are all primitive recursive:

1. The relation "x divides y", written $x \mid y$, defined by

$$x \mid y \Leftrightarrow \exists z \leq y \, (x \cdot z) = y.$$

2. The relation $\text{Prime}(x)$, which holds iff x is prime, defined by

$$\text{Prime}(x) \Leftrightarrow (x \geq 2 \wedge \forall y \leq x \, (y \mid x \rightarrow y = 1 \vee y = x)).$$

3. The function nextPrime(x), which returns the first prime number larger than x, defined by

$$\text{nextPrime(x)} = \min y \leq x! + 1 \; (y > x \wedge \text{Prime}(y))$$

Here we are relying on Euclid's proof of the fact that there is always a prime number between x and $x! + 1$.

4. The function $p(x)$, returning the xth prime, defined by $p(0) = 2, p(x + 1) = \text{nextPrime}(p(x))$. For convenience we will write this as p_x (starting with 0; i.e. $p_0 = 2$).

6.7 Sequences

The set of primitive recursive functions is remarkably robust. But we will be able to do even more once we have developed an adequate means of handling *sequences*. We will identify finite sequences of natural numbers with natural numbers in the following way: the sequence $\langle a_0, a_1, a_2, \ldots, a_k \rangle$ corresponds to the number

$$p_0^{a_0+1} \cdot p_1^{a_1+1} \cdot p_2^{a_2+1} \cdot \ldots \cdot p_k^{a_k+1}.$$

We add one to the exponents to guarantee that, for example, the sequences $\langle 2, 7, 3 \rangle$ and $\langle 2, 7, 3, 0, 0 \rangle$ have distinct numeric codes. We will take both 0 and 1 to code the empty sequence; for concreteness, let $\emptyset$ denote 0.

Let us define the following functions:

1. $\text{len}(s)$, which returns the length of the sequence s:

$$\text{len}(s) = \begin{cases} 0 & \text{if } s = 0 \text{ or } s = 1 \\ \min i < s \, (p_i \mid s \wedge \forall j < s \, (j > i \rightarrow p_j \not\mid s)) + 1 & \text{otherwise} \end{cases}$$

   Note that we need to bound the search on i; clearly s provides an acceptable bound.

2. $\text{append}(s, a)$, which returns the result of appending a to the sequence s:

$$\text{append}(s, a) = \begin{cases} 2^{a+1} & \text{if } s = 0 \text{ or } s = 1 \\ s \cdot p_{\text{len}(s)}^{a+1} & \text{otherwise} \end{cases}$$

3. $\text{element}(s, i)$, which returns the ith element of s (where the initial element is called the 0th), or 0 if i is greater than or equal to the length of s:

$$\text{element}(s, i) = \begin{cases} 0 & \text{if } i \geq \text{len}(s) \\ \min j < s \, (p_i^{j+2} \not\mid s) - 1 & \text{otherwise} \end{cases}$$

Instead of using the official names for the functions defined above, we introduce a more compact notation. We will use $(s)_i$ instead of $\text{element}(s, i)$, and $\langle s_0, \ldots, s_k \rangle$ to abbreviate $\text{append}(\text{append}(\ldots \text{append}(\emptyset, s_0) \ldots), s_k)$. Note that if s has length k, the elements of s are $(s)_0, \ldots, (s)_{k-1}$.

It will be useful for us to be able to bound the numeric code of a sequence in terms of its length and its largest element. Suppose s is a sequence of length k, each element of which is less than equal to some number x. Then s has at most k prime factors, each at most p_{k-1}, and each raised to at most $x+1$ in the prime factorization of s. In other words, if we define

$$\text{sequenceBound}(x,k) = p_{k-1}^{k\cdot(x+1)},$$

then the numeric code of the sequence s described above is at most $\text{sequenceBound}(x,k)$.

Having such a bound on sequences gives us a way of defining new functions using bounded search. For example, suppose we want to define the function $\text{concat}(s,t)$, which concatenates two sequences. One first option is to define a "helper" function $\text{hconcat}(s,t,n)$ which concatenates the first n symbols of t to s. This function can be defined by primitive recursion, as follows:

1. $\text{hconcat}(s,t,0) = s$
2. $\text{hconcat}(s,t,n+1) = \text{append}(\text{hconcat}(s,t,n),(t)_n)$

Then we can define concat by

$$\text{concat}(s,t) = \text{hconcat}(s,t,\text{len}(t)).$$

But using bounded search, we can be lazy. All we need to do is write down a primitive recursive *specification* of the object (number) we are looking for, and a bound on how far to look. The following works:

$$\begin{aligned}\text{concat}(s,t) = \min v < \text{sequenceBound}(s+t, \text{len}(s)+\text{len}(t))& \\ (\text{len}(v) = \text{len}(s)+\text{len}(t) \wedge& \\ \forall i < \text{len}(s)\,((v)_i = (s)_i) \wedge \forall j < \text{len}(t)\,((v)_{\text{len}(s)+j} = (t)_j))&\end{aligned}$$

We will write $s \frown t$ instead of $\text{concat}(s,t)$.

6.8 Other Recursions

Using pairing and sequencing, we can justify more exotic (and useful) forms of primitive recursion. For example, it is often useful to define two functions simultaneously, such as in the following definition:

$$\begin{aligned} f_0(0,\vec{z}) &= k_0(\vec{z}) \\ f_1(0,\vec{z}) &= k_1(\vec{z}) \\ f_0(x+1,\vec{z}) &= h_0(x, f_0(x,\vec{z}), f_1(x,\vec{z}), \vec{z}) \\ f_1(x+1,\vec{z}) &= h_1(x, f_0(x,\vec{z}), f_1(x,\vec{z}), \vec{z}) \end{aligned}$$

This is an instance of *simultaneous recursion*. Another useful way of defining functions is to give the value of $f(x+1,\vec{z})$ in terms of *all* the values $f(0,\vec{z})$, $\ldots$, $f(x,\vec{z})$, as in the following definition:

$$\begin{aligned} f(0,\vec{z}) &= g(\vec{z}) \\ f(x+1,\vec{z}) &= h(x,\langle f(0,\vec{z}),\ldots,f(x,\vec{z})\rangle,\vec{z}). \end{aligned}$$

The following schema captures this idea more succinctly:

$$f(x,\vec{z}) = h(x,\langle f(0,\vec{z}),\ldots,f(x-1,\vec{z})\rangle)$$

with the understanding that the second argument to h is just the empty sequence when x is 0. In either formulation, the idea is that in computing the "successor step," the function f can make use of the entire sequence of values computed so far. This is known as a *course-of-values* recursion. For a particular example, it can be used to justify the following type of definition:

$$f(x,\vec{z}) = \begin{cases} h(x,f(k(x,\vec{z}),\vec{z}),\vec{z}) & \text{if } k(x,\vec{z}) < x \\ g(x,\vec{z}) & \text{otherwise} \end{cases}$$

In other words, the value of f at x can be computed in terms of the value of f at *any* previous value, given by k.

You should think about how to obtain these functions using ordinary primitive recursion. One final version of primitive recursion is more flexible in that one is allowed to change the *parameters* (side values) along the way:

$$\begin{aligned} f(0,\vec{z}) &= g(\vec{z}) \\ f(x+1,\vec{z}) &= h(x,f(x,k(\vec{z})),\vec{z}) \end{aligned}$$

This, too, can be simulated with ordinary primitive recursion. (Doing so is tricky. For a hint, try unwinding the computation by hand.)

Finally, notice that we can always extend our "universe" by defining additional objects in terms of the natural numbers, and defining primitive recursive functions that operate on them. For example, we can take an integer to be given by a pair $\langle m,n\rangle$ of natural numbers, which, intuitively, represents the integer $m-n$. In other words, we say

$$\text{Integer}(x) \Leftrightarrow \text{length}(x) = 2$$

and then we define the following:

1. $\text{iequal}(x,y)$
2. $\text{iplus}(x,y)$
3. $\text{iminus}(x,y)$
4. $\text{itimes}(x,y)$

Similarly, we can define a rational number to be a pair $\langle x,y\rangle$ of integers with $y \neq 0$, representing the value x/y. And we can define qequal, qplus, qminus, qtimes, qdivides, and so on.

6.9 Non-Primitive Recursive Functions

The primitive recursive functions do not exhaust the intuitively computable functions. It should be intuitively clear that we can make a list of all the unary primitive recursive functions, f_0, f_1, f_2, ... such that we can effectively compute the value of f_x on input y; in other words, the function $g(x, y)$, defined by

$$g(x, y) = f_x(y)$$

is computable. But then so is the function

$$\begin{aligned} h(x) &= g(x, x) + 1 \\ &= f_x(x) + 1. \end{aligned}$$

For each primitive recursive function f_i, the value of h and f_i differ at i. So h is computable, but not primitive recursive; and one can say the same about g. This is a an "effective" version of Cantor's diagonalization argument.

One can provide more explicit examples of computable functions that are not primitive recursive. For example, let the notation $g^n(x)$ denote $g(g(\ldots g(x)))$, with n g's in all; and define a sequence $g_0, g_1, \ldots$ of functions by

$$\begin{aligned} g_0(x) &= x + 1 \\ g_{n+1}(x) &= g_n^x(x) \end{aligned}$$

You can confirm that each function g_n is primitive recursive. Each successive function grows much faster than the one before; $g_1(x)$ is equal to $2x$, $g_2(x)$ is equal to $2^x \cdot x$, and $g_3(x)$ grows roughly like an exponential stack of x 2's. Ackermann's function is essentially the function $G(x) = g_x(x)$, and one can show that this grows faster than any primitive recursive function.

Let us return to the issue of enumerating the primitive recursive functions. Remember that we have assigned symbolic notations to each primitive recursive function; so it suffices to enumerate notations. We can assign a natural number $\#(F)$ to each notation F, recursively, as follows:

$$\begin{aligned} \#(0) &= \langle 0 \rangle \\ \#(S) &= \langle 1 \rangle \\ \#(P_i^n) &= \langle 2, n, i \rangle \\ \#(\mathrm{Comp}_{k,l}[H, G_0, \ldots, G_{k-1}]) &= \langle 3, k, l, \#(H), \#(G_0), \ldots, \#(G_{k-1}) \rangle \\ \#(\mathrm{Rec}_l[G, H]) &= \langle 4, l, \#(G), \#(H) \rangle \end{aligned}$$

Here I am using the fact that every sequence of numbers can be viewed as a natural number, using the codes from the last section. The upshot is that every code is assigned a natural number. Of course, some sequences (and hence some numbers) do not correspond to notations; but we can let f_i be the unary primitive recursive function with notation coded as i, if i codes such a

notation; and the constant 0 function otherwise. The net result is that we have an explicit way of enumerating the unary primitive recursive functions.

(In fact, some functions, like the constant zero function, will appear more than once on the list. This is not just an artifact of our coding, but also a result of the fact that the constant zero function has more than one notation. We will later see that one can not computably avoid these repetitions; for example, there is no computable function that decides whether or not a given notation represents the constant zero function.)

We can now take the function $g(x, y)$ to be given by $f_x(y)$, where f_x refers to the enumeration we have just described. How do we know that $g(x, y)$ is computable? Intuitively, this is clear: to compute $g(x, y)$, first "unpack" x, and see if it a notation for a unary function; if it is, compute the value of that function on input y.

You may already be convinced that (with some work!) one can write a program (say, in Java or C++) that does this; and now we can appeal to the Church-Turing thesis, which says that anything that, intuitively, is computable can be computed by a Turing machine.

Of course, a more direct way to show that $g(x, y)$ is computable is to describe a Turing machine that computes it, explicitly. This would, in particular, avoid the Church-Turing thesis and appeals to intuition. But, as noted above, working with Turing machines directly is unpleasant. Soon we will have built up enough machinery to show that $g(x, y)$ is computable, appealing to a model of computation that can be *simulated* on a Turing machine: namely, the recursive functions.

6.10 Partial Recursive Functions

To motivate the definition of the recursive functions, note that our proof that there are computable functions that are not primitive recursive actually establishes much more. The argument was simple: all we used was the fact was that it is possible to enumerate functions $f_0, f_1, \ldots$ such that, as a function of x and y, $f_x(y)$ is computable. So the argument applies to *any class of functions that can be enumerated in such a way*. This puts us in a bind: we would like to describe the computable functions explicitly; but any explicit description of a collection of computable functions cannot be exhaustive!

The way out is to allow *partial* functions to come into play. We will see that it *is* possible to enumerate the partial computable functions. In fact, we already pretty much know that this is the case, since it is possible to enumerate Turing machines in a systematic way. We will come back to our diagonal argument later, and explore why it does not go through when partial functions are included.

The question is now this: what do we need to add to the primitive recursive functions to obtain all the partial recursive functions? We need to do two

things:

1. Modify our definition of the primitive recursive functions to allow for partial functions as well.

2. *Add* something to the definition, so that some new partial functions are included.

The first is easy. As before, we will start with zero, successor, and projections, and close under composition and primitive recursion. The only difference is that we have to modify the definitions of composition and primitive recursion to allow for the possibility that some of the terms in the definition are not defined. If f and g are partial functions, we will write $f(x) \downarrow$ to mean that f is defined at x, i.e., x is in the domain of f; and $f(x) \uparrow$ to mean the opposite, i.e., that f is not defined at x. We will use $f(x) \simeq g(x)$ to mean that either $f(x)$ and $g(x)$ are both undefined, or they are both defined and equal. We will use these notations for more complicated terms as well. We will adopt the convention that if h and $g_0, \ldots, g_k$ all are partial functions, then

$$h(g_0(\vec{x}), \ldots, g_k(\vec{x}))$$

is defined if and only if each g_i is defined at $\vec{x}$, and h is defined at $g_0(\vec{x})$, $\ldots$, $g_k(\vec{x})$. With this understanding, the definitions of composition and primitive recursion for partial functions is just as above, except that we have to replace "$=$" by "$\simeq$".

What we will add to the definition of the primitive recursive functions to obtain partial functions is the *unbounded search operator*. If $f(x, \vec{z})$ is any partial function on the natural numbers, define $\mu x\ f(x, \vec{z})$ to be

> the least x such that $f(0, \vec{z}), f(1, \vec{z}), \ldots, f(x, \vec{z})$ are all defined, and $f(x, \vec{z}) = 0$, if such an x exists

with the understanding that $\mu x\ f(x, \vec{z})$ is undefined otherwise. This defines $\mu x\ f(x, \vec{z})$ uniquely.

Note that our definition makes no reference to Turing machines, or algorithms, or any specific computational model. But like composition and primitive recursion, there is an operational, computational intuition behind unbounded search. When it comes to the computability of a partial function, arguments where the function is undefined correspond to inputs for which the computation does not halt. The procedure for computing $\mu x\ f(x, \vec{z})$ will amount to this: compute $f(0, \vec{z}), f(1, \vec{z}), f(2, \vec{z})$ until a value of 0 is returned. If any of the intermediate computations do not halt, however, neither does the computation of $\mu x\ f(x, \vec{z})$.

If $R(x, \vec{z})$ is any relation, $\mu x\ R(x, \vec{z})$ is defined to be $\mu x\ (1 \dot{-} \chi_R(x, \vec{z}))$. In other words, $\mu x\ R(x, \vec{z})$ returns the least value of x such that $R(x, \vec{z})$ holds. So, if $f(x, \vec{z})$ is a total function, $\mu x\ f(x, \vec{z})$ is the same as $\mu x\ (f(x, \vec{z}) = 0)$. But note

that our original definition is more general, since it allows for the possibility that $f(x, \vec{z})$ is not everywhere defined (whereas, in contrast, the characteristic function of a relation is always total).

Definition 6.6. The set of *partial recursive functions* is the smallest set of partial functions from the natural numbers to the natural numbers (of various arities) containing zero, successor, and projections, and closed under composition, primitive recursion, and unbounded search.

Of course, some of the partial recursive functions will happen to be total, i.e., defined for every argument.

Definition 6.7. The set of *recursive functions* is the set of partial recursive functions that are total.

A recursive function is sometimes called "total recursive" to emphasize that it is defined everywhere.

6.11 The Normal Form Theorem

Theorem 6.8 (Kleene's Normal Form Theorem)**.** *There is a primitive recursive relation $T(e, x, s)$ and a primitive recursive function $U(s)$, with the following property: if f is any partial recursive function, then for some e,*

$$f(x) \simeq U(\mu s\ T(e, x, s))$$

for every x.

The proof of the normal form theorem is involved, but the basic idea is simple. Every partial recursive function has an *index* e, intuitively, a number coding its program or definition. If $f(x) \downarrow$, the computation can be recorded systematically and coded by some number s, and that s codes the computation of f on input x can be checked primitive recursively using only x and the definition e. This means that T is primitive recursive. Given the full record of the computation s, the "upshot" of s is the value of $f(x)$, and it can be obtained from s primitive recursively as well.

The normal form theorem shows that only a single unbounded search is required for the definition of any partial recursive function. We can use the numbers e as "names" of partial recursive functions, and write φ_e for the function f defined by the equation in the theorem. Note that any partial recursive function can have more than one index—in fact, every partial recursive function has infinitely many indices.

6.12 The Halting Problem

The *halting problem* in general is the problem of deciding, given the specification e (e.g., program) of a computable function and a number n, whether the computation of the function on input n halts, i.e., produces a result. Famously, Alan Turing proved that this problem itself cannot be solved by a computable function, i.e., the function

$$h(e,n) = \begin{cases} 1 & \text{if computation } e \text{ halts on input } n \\ 0 & \text{otherwise,} \end{cases}$$

is not computable.

In the context of partial recursive functions, the role of the specification of a program may be played by the index e given in Kleene's normal form theorem. If f is a partial recursive function, any e for which the equation in the normal form theorem holds, is an index of f. Given a number e, the normal form theorem states that

$$\varphi_e(x) \simeq U(\mu s\ T(e,x,s))$$

is partial recursive, and for every partial recursive $f\colon \mathbb{N} \to \mathbb{N}$, there is an $e \in \mathbb{N}$ such that $\varphi_e(x) \simeq f(x)$ for all $x \in \mathbb{N}$. In fact, for each such f there is not just one, but infinitely many such e. The *halting function* h is defined by

$$h(e,x) = \begin{cases} 1 & \text{if } \varphi_e(x) \downarrow \\ 0 & \text{otherwise.} \end{cases}$$

Note that $h(e,x) = 0$ if $\varphi_e(x) \uparrow$, but also when e is not the index of a partial recursive function at all.

Theorem 6.9. *The halting function h is not partial recursive.*

Proof. If h were partial recursive, we could define

$$d(y) = \begin{cases} 1 & \text{if } h(y,y) = 0 \\ \mu x\ x \neq x & \text{otherwise.} \end{cases}$$

From this definition it follows that

1. $d(y) \downarrow$ iff $\varphi_y(y) \uparrow$ or y is not the index of a partial recursive function.
2. $d(y) \uparrow$ iff $\varphi_y(y) \downarrow$.

If h were partial recursive, then d would be partial recursive as well. Thus, by the Kleene normal form theorem, it has an index e_d. Consider the value of $h(e_d, e_d)$. There are two possible cases, 0 and 1.

1. If $h(e_d, e_d) = 1$ then $\varphi_{e_d}(e_d) \downarrow$. But $\varphi_{e_d} \simeq d$, and $d(e_d)$ is defined iff $h(e_d, e_d) = 0$. So $h(e_d, e_d) \neq 1$.

2. If $h(e_d, e_d) = 0$ then either e_d is not the index of a partial recursive function, or it is and $\varphi_{e_d}(e_d) \uparrow$. But again, $\varphi_{e_d} \simeq d$, and $d(e_d)$ is undefined iff $\varphi_{e_d}(e_d) \downarrow$.

The upshot is that e_d cannot, after all, be the index of a partial recursive function. But if h were partial recursive, d would be too, and so our definition of e_d as an index of it would be admissible. We must conclude that h cannot be partial recursive. □

6.13 General Recursive Functions

There is another way to obtain a set of total functions. Say a total function $f(x, \vec{z})$ is *regular* if for every sequence of natural numbers $\vec{z}$, there is an x such that $f(x, \vec{z}) = 0$. In other words, the regular functions are exactly those functions to which one can apply unbounded search, and end up with a total function. One can, conservatively, restrict unbounded search to regular functions:

Definition 6.10. The set of *general recursive functions* is the smallest set of functions from the natural numbers to the natural numbers (of various arities) containing zero, successor, and projections, and closed under composition, primitive recursion, and unbounded search applied to *regular* functions.

Clearly every general recursive function is total. The difference between Definition 6.10 and Definition 6.7 is that in the latter one is allowed to use partial recursive functions along the way; the only requirement is that the function you end up with at the end is total. So the word "general," a historic relic, is a misnomer; on the surface, Definition 6.10 is *less* general than Definition 6.7. But, fortunately, the difference is illusory; though the definitions are different, the set of general recursive functions and the set of recursive functions are one and the same.

Chapter 7

The Lambda Calculus

7.1 Introduction

The lambda calculus was originally designed by Alonzo Church in the early 1930s as a basis for constructive logic, and *not* as a model of the computable functions. But soon after the Turing computable functions, the recursive functions, and the general recursive functions were shown to be equivalent, lambda computability was added to the list. The fact that this initially came as a small surprise makes the characterization all the more interesting.

Lambda notiation is a convenient way of referring to a function directly by a symbolic expression which defines it, instead of defining a name for it. Instead of saying "let f be the function defined by $f(x) = x + 3$," one can say, "let f be the function $\lambda x.\,(x + 3)$." In other words, $\lambda x.\,(x + 3)$ is just a *name* for the function that adds three to its argument. In this expression, x is a dummy variable, or a placeholder: the same function can just as well be denoted by $\lambda y.\,(y + 3)$. The notation works even with other parameters around. For example, suppose $g(x, y)$ is a function of two variables, and k is a natural number. Then $\lambda x.\,g(x, k)$ is the function which maps any x to $g(x, k)$.

This way of defining a function from a symbolic expression is known as *lambda abstraction*. The flip side of lambda abstraction is *application*: assuming one has a function f (say, defined on the natural numbers), one can *apply* it to any value, like 2. In conventional notation, of course, we write $f(2)$ for the result.

What happens when you combine lambda abstraction with application? Then the resulting expression can be simplified, by "plugging" the applicand in for the abstracted variable. For example,

$$(\lambda x.\,(x + 3))(2)$$

can be simplified to $2 + 3$.

Up to this point, we have done nothing but introduce new notations for

conventional notions. The lambda calculus, however, represents a more radical departure from the set-theoretic viewpoint. In this framework:

1. Everything denotes a function.
2. Functions can be defined using lambda abstraction.
3. Anything can be applied to anything else.

For example, if F is a term in the lambda calculus, $F(F)$ is always assumed to be meaningful. This liberal framework is known as the *untyped* lambda calculus, where "untyped" means "no restriction on what can be applied to what."

There is also a *typed* lambda calculus, which is an important variation on the untyped version. Although in many ways the typed lambda calculus is similar to the untyped one, it is much easier to reconcile with a classical set-theoretic framework, and has some very different properties.

Research on the lambda calculus has proved to be central in theoretical computer science, and in the design of programming languages. LISP, designed by John McCarthy in the 1950s, is an early example of a language that was influenced by these ideas.

7.2 The Syntax of the Lambda Calculus

One starts with a sequence of variables x, y, z, ... and some constant symbols a, b, c, ... The set of terms is defined inductively, as follows:

1. Each variable is a term.
2. Each constant is a term.
3. If M and N are terms, so is (MN).
4. If M is a term and x is a variable, then $(\lambda x.\, M)$ is a term.

The system without any constants at all is called the *pure* lambda calculus.

We will follow a few notational conventions:

1. When parentheses are left out, application takes place from left to right. For example, if M, N, P, and Q are terms, then $MNPQ$ abbreviates $(((MN)P)Q)$.
2. Again, when parentheses are left out, lambda abstraction is to be given the widest scope possible. From example, $\lambda x.\, MNP$ is read $\lambda x.\, (MNP)$.
3. A lambda can be used to abstract multiple variables. For example, $\lambda xyz.\, M$ is short for $\lambda x.\, \lambda y.\, \lambda z.\, M$.

For example,

$$\lambda xy.\, xxyx\lambda z.\, xz$$

abbreviates

$$\lambda x.\, \lambda y.\, ((((xx)y)x)\lambda z.\, (xz)).$$

You should memorize these conventions. They will drive you crazy at first, but you will get used to them, and after a while they will drive you less crazy than having to deal with a morass of parentheses.

Two terms that differ only in the names of the bound variables are called α-equivalent; for example, $\lambda x.\, x$ and $\lambda y.\, y$. It will be convenient to think of these as being the "same" term; in other words, when we say that M and N are the same, we also mean "up to renamings of the bound variables." Variables that are in the scope of a λ are called "bound", while others are called "free." There are no free variables in the previous example; but in

$$(\lambda z.\, yz)x$$

y and x are free, and z is bound.

7.3 Reduction of Lambda Terms

What can one do with lambda terms? Simplify them. If M and N are any lambda terms and x is any variable, we can use $M[N/x]$ to denote the result of substituting N for x in M, after renaming any bound variables of M that would interfere with the free variables of N after the substitution. For example,

$$(\lambda w.\, xxw)[yyz/x] = \lambda w.\, (yyz)(yyz)w.$$

Alternative notations for substitution are $[N/x]M$, $M[N/x]$, and also $M[x/N]$. Beware!

Intuitively, $(\lambda x.\, M)N$ and $M[N/x]$ have the same meaning; the act of replacing the first term by the second is called β-conversion. More generally, if it is possible convert a term P to P' by β-conversion of some subterm, one says *P β-reduces to P' in one step*. If P can be converted to P' with any number of one-step reductions (possibly none), then *P β-reduces* to P'. A term that cannot be β-reduced any further is called *β-irreducible*, or β-normal. I will say "reduces" instead of "β-reduces," etc., when the context is clear.

Let us consider some examples.

1. We have

$$\begin{aligned}(\lambda x.\, xxy)\lambda z.\, z &\rhd_1 (\lambda z.\, z)(\lambda z.\, z)y \\ &\rhd_1 (\lambda z.\, z)y \\ &\rhd_1 y\end{aligned}$$

2. "Simplifying" a term can make it more complex:

$$\begin{aligned}(\lambda x.\, xxy)(\lambda x.\, xxy) &\rhd_1 (\lambda x.\, xxy)(\lambda x.\, xxy)y \\ &\rhd_1 (\lambda x.\, xxy)(\lambda x.\, xxy)yy \\ &\rhd_1 \dots\end{aligned}$$

3. It can also leave a term unchanged:

$$(\lambda x.\, xx)(\lambda x.\, xx) \rhd_1 (\lambda x.\, xx)(\lambda x.\, xx)$$

4. Also, some terms can be reduced in more than one way; for example,

$$(\lambda x.\, (\lambda y.\, yx)z)v \rhd_1 (\lambda y.\, yv)z$$

   by contracting the outermost application; and

$$(\lambda x.\, (\lambda y.\, yx)z)v \rhd_1 (\lambda x.\, zx)v$$

   by contracting the innermost one. Note, in this case, however, that both terms further reduce to the same term, zv.

The final outcome in the last example is not a coincidence, but rather illustrates a deep and important property of the lambda calculus, known as the "Church-Rosser property."

7.4 The Church-Rosser Property

Theorem 7.1. *Let M, N_1, and N_2 be terms, such that $M \rhd N_1$ and $M \rhd N_2$. Then there is a term P such that $N_1 \rhd P$ and $N_2 \rhd P$.*

Corollary 7.2. *Suppose M can be reduced to normal form. Then this normal form is unique.*

Proof. If $M \rhd N_1$ and $M \rhd N_2$, by the previous theorem there is a term P such that N_1 and N_2 both reduce to P. If N_1 and N_2 are both in normal form, this can only happen if $N_1 = P = N_2$. □

Finally, we will say that two terms M and N are *β-equivalent*, or just *equivalent*, if they reduce to a common term; in other words, if there is some P such that $M \rhd P$ and $N \rhd P$. This is written $M \equiv N$. Using Theorem 7.1, you can check that $\equiv$ is an equivalence relation, with the additional property that for every M and N, if $M \rhd N$ or $N \rhd M$, then $M \equiv N$. (In fact, one can show that $\equiv$ is the *smallest* equivalence relation having this property.)

7.5 Representability by Lambda Terms

How can the lambda calculus serve as a model of computation? At first, it is not even clear how to make sense of this statement. To talk about computability on the natural numbers, we need to find a suitable representation for such numbers. Here is one that works surprisingly well.

Definition 7.3. For each natural number n, define the *numeral* $\overline{n}$ to be the lambda term $\lambda x.\, \lambda y.\, (x(x(x(\ldots x(y)))))$, where there are n x's in all.

The terms $\overline{n}$ are "iterators": on input f, $\overline{n}$ returns the function mapping y to $f^n(y)$. Note that each numeral is normal. We can now say what it means for a lambda term to "compute" a function on the natural numbers.

Definition 7.4. Let $f(x_0, \ldots, x_{n-1})$ be an n-ary partial function from $\mathbb{N}$ to $\mathbb{N}$. We say a lambda term X *represents* f if for every sequence of natural numbers $m_0, \ldots, m_{n-1}$,

$$X\overline{m_0}\overline{m_1}\ldots\overline{m_{n-1}} \triangleright \overline{f(m_0, m_1, \ldots, m_{n-1})}$$

if $f(m_0, \ldots, m_{n-1})$ is defined, and $X\overline{m_0}\overline{m_1}\ldots\overline{m_{n-1}}$ has no normal form otherwise.

Theorem 7.5. *A function f is a partial computable function if and only if it is represented by a lambda term.*

This theorem is somewhat striking. As a model of computation, the lambda calculus is a rather simple calculus; the only operations are lambda abstraction and application! From these meager resources, however, it is possible to implement any computational procedure.

7.6 Lambda Representable Functions are Computable

Theorem 7.6. *If a partial function f is represented by a lambda term, it is computable.*

Proof. Suppose a function f, is represented by a lambda term X. Let us describe an informal procedure to compute f. On input m_0, …, m_{n-1}, write down the term $X\overline{m}_0 \ldots \overline{m}_{n-1}$. Build a tree, first writing down all the one-step reductions of the original term; below that, write all the one-step reductions of those (i.e., the two-step reductions of the original term); and keep going. If you ever reach a numeral, return that as the answer; otherwise, the function is undefined.

An appeal to Church's thesis tells us that this function is computable. A better way to prove the theorem would be to give a recursive description of this search procedure. For example, one could define a sequence primitive recursive functions and relations, "IsASubterm," "Substitute," "ReducesToInOneStep,"

"ReductionSequence," "Numeral," etc. The partial recursive procedure for computing $f(m_0, \ldots, m_{n-1})$ is then to search for a sequence of one-step reductions starting with $X\overline{m_0}\ldots\overline{m_{n-1}}$ and ending with a numeral, and return the number corresponding to that numeral. The details are long and tedious but otherwise routine. □

7.7 Computable Functions are Lambda Representable

Theorem 7.7. *Every computable partial function if representable by a lambda term.*

Proof. Wwe need to show that every partial computable function f is represented by a lambda term $\overline{f}$. By Kleene's normal form theorem, it suffices to show that every primitive recursive function is represented by a lambda term, and then that the functions so represented are closed under suitable compositions and unbounded search. To show that every primitive recursive function is represented by a lambda term, it suffices to show that the initial functions are represented, and that the partial functions that are represented by lambda terms are closed under composition, primitive recursion, and unbounded search. □

We will use a more conventional notation to make the rest of the proof more readable. For example, we will write $M(x, y, z)$ instead of $Mxyz$. While this is suggestive, you should remember that terms in the untyped lambda calculus do not have associated arities; so, for the same term M, it makes just as much sense to write $M(x, y)$ and $M(x, y, z, w)$. But using this notation indicates that we are treating M as a function of three variables, and helps make the intentions behind the definitions clearer. In a similar way, we will say "define M by $M(x, y, z) = \ldots$" instead of "define M by $M = \lambda x.\, \lambda y.\, \lambda z.\, \ldots$."

7.8 The Basic Primitive Recursive Functions are Lambda Representable

Lemma 7.8. *The functions* 0*, S, and* P_i^n *are lambda representable.*

Proof. Zero, $\overline{0}$, is just $\lambda x.\, \lambda y.\, y$.

The successor function $\overline{S}$, is defined by $\overline{S}(u) = \lambda x.\, \lambda y.\, x(uxy)$. You should think about why this works; for each numeral $\overline{n}$, thought of as an iterator, and each function f, $S(\overline{n}, f)$ is a function that, on input y, applies f n times starting with y, and then applies it once more.

There is nothing to say about projections: $\overline{P_i^n}(x_0, \ldots, x_{n-1}) = x_i$. In other words, by our conventions, $\overline{P_i^n}$ is the lambda term $\lambda x_0.\, \ldots \lambda x_{n-1}.\, x_i$. □

7.9 Lambda Representable Functions Closed under Composition

Lemma 7.9. *The lambda representable functions are closed under composition.*

Proof. Suppose f is defined by composition from $h, g_0, \ldots, g_{k-1}$. Assuming h, $g_0, \ldots, g_{k-1}$ are represented by $\overline{h}, \overline{g_0}, \ldots, \overline{g_{k-1}}$, respectively, we need to find a term $\overline{f}$ representing f. But we can simply define $\overline{f}$ by

$$\overline{f}(x_0, \ldots, x_{l-1}) = \overline{h}(\overline{g_0}(x_0, \ldots, x_{l-1}), \ldots, \overline{g_{k-1}}(x_0, \ldots, x_{l-1})).$$

In other words, the language of the lambda calculus is well suited to represent composition. □

7.10 Lambda Representable Functions Closed under Primitive Recursion

When it comes to primitive recursion, we finally need to do some work. We will have to proceed in stages. As before, on the assumption that we already have terms $\overline{g}$ and $\overline{h}$ representing functions g and h, respectively, we want a term $\overline{f}$ representing the function f defined by

$$\begin{aligned} f(0, \vec{z}) &= g(\vec{z}) \\ f(x+1, \vec{z}) &= h(z, f(x, \vec{z}), \vec{z}). \end{aligned}$$

So, in general, given lambda terms G' and H', it suffices to find a term F such that

$$\begin{aligned} F(\overline{0}, \vec{z}) &\equiv G'(\vec{z}) \\ F(\overline{n+1}, \vec{z}) &\equiv H'(\overline{n}, F(\overline{n}, \vec{z}), \vec{z}) \end{aligned}$$

for every natural number n; the fact that G' and H' represent g and h means that whenever we plug in numerals $\overline{\vec{m}}$ for $\vec{z}$, $F(\overline{n+1}, \overline{\vec{m}})$ will normalize to the right answer.

But for this, it suffices to find a term F satisfying

$$\begin{aligned} F(\overline{0}) &\equiv G \\ F(\overline{n+1}) &\equiv H(\overline{n}, F(\overline{n})) \end{aligned}$$

for every natural number n, where

$$\begin{aligned} G &= \lambda \vec{z}.\, G'(\vec{z}) \text{ and} \\ H(u, v) &= \lambda \vec{z}.\, H'(u, v(u, \vec{z}), \vec{z}). \end{aligned}$$

In other words, with lambda trickery, we can avoid having to worry about the extra parameters $\vec{z}$—they just get absorbed in the lambda notation.

Before we define the term F, we need a mechanism for handling ordered pairs. This is provided by the next lemma.

Lemma 7.10. *There is a lambda term D such that for each pair of lambda terms M and N, $D(M,N)(\overline{0}) \triangleright M$ and $D(M,N)(\overline{1}) \triangleright N$.*

Proof. First, define the lambda term K by

$$K(y) = \lambda x.\, y.$$

In other words, K is the term $\lambda y.\, \lambda x.\, y$. Looking at it differently, for every M, $K(M)$ is a constant function that returns M on any input.

Now define $D(x,y,z)$ by $D(x,y,z) = z(K(y))x$. Then we have

$$\begin{aligned} D(M,N,\overline{0}) &\triangleright \overline{0}(K(N))M \triangleright M \text{ and} \\ D(M,N,\overline{1}) &\triangleright \overline{1}(K(N))M \triangleright K(N)M \triangleright N, \end{aligned}$$

as required. □

The idea is that $D(M,N)$ represents the pair $\langle M,N\rangle$, and if P is assumed to represent such a pair, $P(\overline{0})$ and $P(\overline{1})$ represent the left and right projections, $(P)_0$ and $(P)_1$. We will use the latter notations.

Lemma 7.11. *The lambda representable functions are closed under primitive recursion.*

Proof. We need to show that given any terms, G and H, we can find a term F such that

$$\begin{aligned} F(\overline{0}) &\equiv G \\ F(\overline{n+1}) &\equiv H(\overline{n}, F(\overline{n})) \end{aligned}$$

for every natural number n. The idea is roughly to compute sequences of *pairs*

$$\langle \overline{0}, F(\overline{0})\rangle, \langle \overline{1}, F(\overline{1})\rangle, \ldots,$$

using numerals as iterators. Notice that the first pair is just $\langle \overline{0}, G\rangle$. Given a pair $\langle \overline{n}, F(\overline{n})\rangle$, the next pair, $\langle \overline{n+1}, F(\overline{n+1})\rangle$ is supposed to be equivalent to $\langle \overline{n+1}, H(\overline{n}, F(\overline{n}))\rangle$. We will design a lambda term T that makes this one-step transition.

The details are as follows. Define $T(u)$ by

$$T(u) = \langle S((u)_0), H((u)_0, (u)_1)\rangle.$$

Now it is easy to verify that for any number n,

$$T(\langle \overline{n}, M\rangle) \triangleright \langle \overline{n+1}, H(\overline{n}, M)\rangle.$$

As suggested above, given G and H, define $F(u)$ by

$$F(u) = (u(T, \langle \overline{0}, G \rangle))_1.$$

In other words, on input $\overline{n}$, F iterates T n times on $\langle \overline{0}, G \rangle$, and then returns the second component. To start with, we have

1. $\overline{0}(T, \langle \overline{0}, G \rangle) \equiv \langle \overline{0}, G \rangle$
2. $F(\overline{0}) \equiv G$

By induction on n, we can show that for each natural number one has the following:

1. $\overline{n+1}(T, \langle \overline{0}, G \rangle) \equiv \langle \overline{n+1}, F(\overline{n+1}) \rangle$
2. $F(\overline{n+1}) \equiv H(\overline{n}, F(\overline{n}))$

For the second clause, we have

$$\begin{aligned} F(\overline{n+1}) &\triangleright (\overline{n+1}(T, \langle \overline{0}, G \rangle))_1 \\ &\equiv (T(\overline{n}(T, \langle \overline{0}, G \rangle)))_1 \\ &\equiv (T(\langle \overline{n}, F(\overline{n}) \rangle))_1 \\ &\equiv (\langle \overline{n+1}, H(\overline{n}, F(\overline{n})) \rangle)_1 \\ &\equiv H(\overline{n}, F(\overline{n})). \end{aligned}$$

Here we have used the induction hypothesis on the second-to-last line. For the first clause, we have

$$\begin{aligned} \overline{n+1}(T, \langle \overline{0}, G \rangle) &\equiv T(\overline{n}(T, \langle \overline{0}, G \rangle)) \\ &\equiv T(\langle \overline{n}, F(\overline{n}) \rangle) \\ &\equiv \langle \overline{n+1}, H(\overline{n}, F(\overline{n})) \rangle \\ &\equiv \langle \overline{n+1}, F(\overline{n+1}) \rangle. \end{aligned}$$

Here we have used the second clause in the last line. So we have shown $F(\overline{0}) \equiv G$ and, for every n, $F(\overline{n+1}) \equiv H(\overline{n}, F(\overline{n}))$, which is exactly what we needed. □

7.11 Fixed-Point Combinators

Suppose you have a lambda term g, and you want another term k with the property that k is β-equivalent to gk. Define terms

$$\mathrm{diag}(x) = xx$$

and

$$l(x) = g(\mathrm{diag}(x))$$

using our notational conventions; in other words, l is the term $\lambda x.\, g(xx)$. Let k be the term ll. Then we have

$$\begin{aligned} k &= (\lambda x.\, g(xx))(\lambda x.\, g(xx)) \\ &\rhd g((\lambda x.\, g(xx))(\lambda x.\, g(xx))) \\ &= gk. \end{aligned}$$

If one takes

$$Y = \lambda g.\, ((\lambda x.\, g(xx))(\lambda x.\, g(xx)))$$

then Yg and $g(Yg)$ reduce to a common term; so $Yg \equiv_\beta g(Yg)$. This is known as "Curry's combinator." If instead one takes

$$Y = (\lambda xg.\, g(xxg))(\lambda xg.\, g(xxg))$$

then in fact Yg reduces to $g(Yg)$, which is a stronger statement. This latter version of Y is known as "Turing's combinator."

7.12 Lambda Representable Functions Closed under Minimization

Lemma 7.12. *Suppose $f(x, y)$ is primitive recursive. Let g be defined by*

$$g(x) \simeq \mu y\, f(x, y).$$

Then g is represented by a lambda term.

Proof. The idea is roughly as follows. Given x, we will use the fixed-point lambda term Y to define a function $h_x(n)$ which searches for a y starting at n; then $g(x)$ is just $h_x(0)$. The function h_x can be expressed as the solution of a fixed-point equation:

$$h_x(n) \simeq \begin{cases} n & \text{if } f(x, n) = 0 \\ h_x(n+1) & \text{otherwise.} \end{cases}$$

Here are the details. Since f is primitive recursive, it is represented by some term F. Remember that we also have a lambda term D, such that $D(M, N, \bar{0}) \rhd M$ and $D(M, N, \bar{1}) \rhd N$. Fixing x for the moment, to represent h_x we want to find a term H (depending on x) satisfying

$$H(\overline{n}) \equiv D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})).$$

We can do this using the fixed-point term Y. First, let U be the term

$$\lambda h.\, \lambda z.\, D(z, (h(Sz)), F(x, z)),$$

and then let H be the term YU. Notice that the only free variable in H is x. Let us show that H satisfies the equation above.

By the definition of Y, we have

$$H = YU \equiv U(YU) = U(H).$$

In particular, for each natural number n, we have

$$\begin{aligned} H(\overline{n}) &\equiv U(H, \overline{n}) \\ &\rhd D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})), \end{aligned}$$

as required. Notice that if you substitute a numeral $\overline{m}$ for x in the last line, the expression reduces to $\overline{n}$ if $F(\overline{m}, \overline{n})$ reduces to $\overline{0}$, and it reduces to $H(S(\overline{n}))$ if $F(\overline{m}, \overline{n})$ reduces to any other numeral.

To finish off the proof, let G be $\lambda x. H(\overline{0})$. Then G represents g; in other words, for every m, $G(\overline{m})$ reduces to reduces to $\overline{g(m)}$, if $g(m)$ is defined, and has no normal form otherwise. □

Chapter 8

Computability Theory

8.1 Introduction

The branch of logic known as *Computability Theory* deals with issues having to do with the computability, or relative computability, of functions and sets. It is a evidence of Kleene's influence that the subject used to be known as *Recursion Theory*, and today, both names are commonly used.

Let us call a function $f\colon \mathbb{N} \nrightarrow \mathbb{N}$ *partial computable* if it can be computed in some model of computation. If f is total we will simply say that f is *computable*. A relation R with computable characteristic function χ_R is also called computable. If f and g are partial functions, we will write $f(x)\downarrow$ to mean that f is defined at x, i.e., x is in the domain of f; and $f(x)\uparrow$ to mean the opposite, i.e., that f is not defined at x. We will use $f(x) \simeq g(x)$ to mean that either $f(x)$ and $g(x)$ are both undefined, or they are both defined and equal.

One can explore the subject without having to refer to a specific model of computation. To do this, one shows that there is a universal partial computable function, $\mathrm{Un}(k, x)$. This allows us to enumerate the partial computable functions. We will adopt the notation φ_k to denote the k-th unary partial computable function, defined by $\varphi_k(x) \simeq \mathrm{Un}(k, x)$. (Kleene used $\{k\}$ for this purpose, but this notation has not been used as much recently.) Slightly more generally, we can uniformly enumerate the partial computable functions of arbitrary arities, and we will use φ^n_k to denote the k-th n-ary partial recursive function.

Recall that if $f(\vec{x}, y)$ is a total or partial function, then $\mu y\ f(\vec{x}, y)$ is the function of $\vec{x}$ that returns the least y such that $f(\vec{x}, y) = 0$, assuming that all of $f(\vec{x}, 0), \ldots, f(\vec{x}, y-1)$ are defined; if there is no such y, $\mu y\ f(\vec{x}, y)$ is undefined. If $R(\vec{x}, y)$ is a relation, $\mu y\ R(\vec{x}, y)$ is defined to be the least y such that $R(\vec{x}, y)$ is true; in other words, the least y such that *one minus* the characteristic function of R is equal to zero at $\vec{x}, y$.

To show that a function is computable, there are two ways one can proceed:

1. Rigorously: describe a Turing machine or partial recursive function explicitly, and show that it computes the function you have in mind;
2. Informally: describe an algorithm that computes it, and appeal to Church's thesis.

There is no fine line between the two; a detailed description of an algorithm should provide enough information so that it is relatively clear how one could, in principle, design the right Turing machine or sequence of partial recursive definitions. Fully rigorous definitions are unlikely to be informative, and we will try to find a happy medium between these two approaches; in short, we will try to find intuitive yet rigorous proofs that the precise definitions could be obtained.

8.2 Coding Computations

In every model of computation, it is possible to do the following:

1. Describe the *definitions* of computable functions in a systematic way. For instance, you can think of Turing machine specifications, recursive definitions, or programs in a programming language as providing these definitions.
2. Describe the complete record of the computation of a function given by some definition for a given input. For instance, a Turing machine computation can be described by the sequence of configurations (state of the machine, contents of the tape) for each step of computation.
3. Test whether a putative record of a computation is in fact the record of how a computable function with a given definition would be computed for a given input.
4. Extract from such a description of the complete record of a computation the value of the function for a given input. For instance, the contents of the tape in the very last step of a halting Turing machine computation is the value.

Using coding, it is possible to assign to each description of a computable function a numerical *index* in such a way that the instructions can be recovered from the index in a computable way. Similarly, the complete record of a computation can be coded by a single number as well. The resulting arithmetical relation "s codes the record of computation of the function with index e for input x" and the function "output of computation sequence with code s" are then computable; in fact, they are primitive recursive.

This fundamental fact is very powerful, and allows us to prove a number of striking and important results about computability, independently of the model of computation chosen.

8.3 The Normal Form Theorem

Theorem 8.1 (Kleene's Normal Form Theorem)**.** *There are a primitive recursive relation $T(k, x, s)$ and a primitive recursive function $U(s)$, with the following property: if f is any partial computable function, then for some k,*

$$f(x) \simeq U(\mu s\ T(k, x, s))$$

for every x.

Proof Sketch. For any model of computation one can rigorously define a description of the computable function f and code such description using a natural number k. One can also rigorously define a notion of "computation sequence" which records the process of computing the function with index k for input x. These computation sequences can likewise be coded as numbers s. This can be done in such a way that (a) it is decidable whether a number s codes the computation sequence of the function with index k on input x and (b) what the end result of the computation sequence coded by s is. In fact, the relation in (a) and the function in (b) are primitive recursive. □

In order to give a rigorous proof of the Normal Form Theorem, we would have to fix a model of computation and carry out the coding of descriptions of computable functions and of computation sequences in detail, and verify that the relation T and function U are primitive recursive. For most applications, it suffices that T and U are computable and that U is total.

It is probably best to remember the proof of the normal form theorem in slogan form: $\mu s\ T(k, x, s)$ searches for a computation sequence of the function with index k on input x, and U returns the output of the computation sequence if one can be found.

T and U can be used to define the enumeration φ_0, φ_1, φ_2, ... From now on, we will assume that we have fixed a suitable choice of T and U, and take the equation

$$\varphi_e(x) \simeq U(\mu s\ T(e, x, s))$$

to be the *definition* of φ_e.

Here is another useful fact:

Theorem 8.2. *Every partial computable function has infinitely many indices.*

Again, this is intuitively clear. Given any (description of) a computable function, one can come up with a different description which computes the same function (input-output pair) but does so, e.g., by first doing something that has no effect on the computation (say, test if $0 = 0$, or count to 5, etc.). The index of the altered description will always be different from the original index. Both are indices of the same function, just computed slightly differently.

8.4 The *s-m-n* Theorem

The next theorem is known as the "*s-m-n* theorem," for a reason that will be clear in a moment. The hard part is understanding just what the theorem says; once you understand the statement, it will seem fairly obvious.

Theorem 8.3. *For each pair of natural numbers n and m, there is a primitive recursive function s^m_n such that for every sequence $x, a_0, \ldots, a_{m-1}, y_0, \ldots, y_{n-1}$, we have*

$$\varphi^n_{s^m_n(x,a_0,\ldots,a_{m-1})}(y_0,\ldots,y_{n-1}) \simeq \varphi^{m+n}_x(a_0,\ldots,a_{m-1},y_0,\ldots,y_{n-1}).$$

It is helpful to think of s^m_n as acting on *programs*. That is, s^m_n takes a program, x, for an $(m+n)$-ary function, as well as fixed inputs $a_0, \ldots, a_{m-1}$; and it returns a program, $s^m_n(x, a_0, \ldots, a_{m-1})$, for the n-ary function of the remaining arguments. It you think of x as the description of a Turing machine, then $s^m_n(x, a_0, \ldots, a_{m-1})$ is the Turing machine that, on input $y_0, \ldots, y_{n-1}$, prepends $a_0, \ldots, a_{m-1}$ to the input string, and runs x. Each s^m_n is then just a primitive recursive function that finds a code for the appropriate Turing machine.

8.5 The Universal Partial Computable Function

Theorem 8.4. *There is a universal partial computable function $\mathrm{Un}(k, x)$. In other words, there is a function $\mathrm{Un}(k, x)$ such that:*

1. *$\mathrm{Un}(k, x)$ is partial computable.*
2. *If $f(x)$ is any partial computable function, then there is a natural number k such that $f(x) \simeq \mathrm{Un}(k, x)$ for every x.*

Proof. Let $\mathrm{Un}(k, x) \simeq U(\mu s\, T(k, x, s))$ in Kleene's normal form theorem. □

This is just a precise way of saying that we have an effective enumeration of the partial computable functions; the idea is that if we write f_k for the function defined by $f_k(x) = \mathrm{Un}(k, x)$, then the sequence $f_0, f_1, f_2, \ldots$ includes all the partial computable functions, with the property that $f_k(x)$ can be computed "uniformly" in k and x. For simplicity, we am using a binary function that is universal for unary functions, but by coding sequences of numbers we can easily generalize this to more arguments. For example, note that if $f(x, y, z)$ is a 3-place partial recursive function, then the function $g(x) \simeq f((x)_0, (x)_1, (x)_2)$ is a unary recursive function.

8.6 No Universal Computable Function

Theorem 8.5. *There is no universal computable function. In other words, the universal function $\mathrm{Un}'(k, x) = \varphi_k(x)$ is not computable.*

Proof. This theorem says that there is no *total* computable function that is universal for the total computable functions. The proof is a simple diagonalization: if $\mathrm{Un}'(k, x)$ were total and computable, then

$$d(x) = \mathrm{Un}'(x, x) + 1$$

would also be total and computable. However, for every k, $d(k)$ is not equal to $\mathrm{Un}'(k, k)$. □

Theorem Theorem 8.4 above shows that we can get around this diagonalization argument, but only at the expense of allowing partial functions. It is worth trying to understand what goes wrong with the diagonalization argument, when we try to apply it in the partial case. In particular, the function $h(x) = \mathrm{Un}(x, x) + 1$ *is* partial recursive. Suppose h is the k-th function in the enumeration; what can we say about $h(k)$?

8.7 The Halting Problem

Since, in our construction, $\mathrm{Un}(k, x)$ is defined if and only if the computation of the function coded by k produces a value for input x, it is natural to ask if we can decide whether this is the case. And in fact, it is not. For the Turing machine model of computation, this means that whether a given Turing machine halts on a given input is computationally undecidable. The following theorem is therefore known as the "undecidability of the halting problem." I will provide two proofs below. The first continues the thread of our previous discussion, while the second is more direct.

Theorem 8.6. *Let*

$$h(k, x) = \begin{cases} 1 & \textit{if } \mathrm{Un}(k, x) \textit{ is defined} \\ 0 & \textit{otherwise.} \end{cases}$$

Then h is not computable.

Proof. If h were computable, we would have a universal computable function, as follows. Suppose h is computable, and define

$$\mathrm{Un}'(k, x) = \begin{cases} fnUn(k, x) & \text{if } h(k, x) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

But now $\mathrm{Un}'(k, x)$ is a total function, and is computable if h is. For instance, we could define g using primitive recursion, by

$$\begin{aligned} g(0, k, x) &\simeq 0 \\ g(y + 1, k, x) &\simeq \mathrm{Un}(k, x); \end{aligned}$$

then

$$\mathrm{Un}'(k, x) \simeq g(h(k, x), k, x).$$

And since $\mathrm{Un}'(k, x)$ agrees with $\mathrm{Un}(k, x)$ wherever the latter is defined, Un' is universal for those partial computable functions that happen to be total. But this contradicts Theorem 8.5. □

Proof. Suppose $h(k, x)$ were computable. Define the function g by

$$g(x) = \begin{cases} 0 & \text{if } h(x, x) = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function g is partial computable; for example, one can define it as $\mu y\ h(x, x) = 0$. So, for some k, $g(x) \simeq \mathrm{Un}(k, x)$ for every x. Is g defined at k? If it is, then, by the definition of g, $h(k, k) = 0$. By the definition of f, this means that $\mathrm{Un}(k, k)$ is undefined; but by our assumption that $g(k) \simeq \mathrm{Un}(k, x)$ for every x, this means that $g(k)$ is undefined, a contradiction. On the other hand, if $g(k)$ is undefined, then $h(k, k) \neq 0$, and so $h(k, k) = 1$. But this means that $\mathrm{Un}(k, k)$ is defined, i.e., that $g(k)$ is defined. □

We can describe this argument in terms of Turing machines. Suppose there were a Turing machine H that took as input a description of a Turing machine K and an input x, and decided whether or not K halts on input x. Then we could build another Turing machine G which takes a single input x, calls H to decide if machine x halts on input x, and does the opposite. In other words, if H reports that x halts on input x, G goes into an infinite loop, and if H reports that x doesn't halt on input x, then G just halts. Does G halt on input G? The argument above shows that it does if and only if it doesn't—a contradiction. So our supposition that there is a such Turing machine H, is false.

8.8 Comparison with Russell's Paradox

It is instructive to compare and contrast the arguments in this section with Russell's paradox:

1. Russell's paradox: let $S = \{x : x \notin x\}$. Then $x \in S$ if and only if $x \notin S$, a contradiction.

   Conclusion: There is no such set S. Assuming the existence of a "set of all sets" is inconsistent with the other axioms of set theory.

2. A modification of Russell's paradox: let F be the "function" from the set of all functions to $\{0, 1\}$, defined by

$$F(f) = \begin{cases} 1 & \text{if } f \text{ is in the domain of } f, \text{ and } f(f) = 0 \\ 0 & \text{otherwise} \end{cases}$$

A similar argument shows that $F(F) = 0$ if and only if $F(F) = 1$, a contradiction.

Conclusion: F is not a function. The "set of all functions" is too big to be the domain of a function.

3. The diagonalization argument: let f_0, f_1, ... be the enumeration of the partial computable functions, and let $G\colon \mathbb{N} \to \{0,1\}$ be defined by

$$G(x) = \begin{cases} 1 & \text{if } f_x(x) \downarrow = 0 \\ 0 & \text{otherwise} \end{cases}$$

If G is computable, then it is the function f_k for some k. But then $G(k) = 1$ if and only if $G(k) = 0$, a contradiction.

Conclusion: G is not computable. Note that according to the axioms of set theory, G is still a function; there is no paradox here, just a clarification.

That talk of partial functions, computable functions, partial computable functions, and so on can be confusing. The set of all partial functions from $\mathbb{N}$ to $\mathbb{N}$ is a big collection of objects. Some of them are total, some of them are computable, some are both total and computable, and some are neither. Keep in mind that when we say "function," by default, we mean a total function. Thus we have:

1. computable functions
2. partial computable functions that are not total
3. functions that are not computable
4. partial functions that are neither total nor computable

To sort this out, it might help to draw a big square representing all the partial functions from $\mathbb{N}$ to $\mathbb{N}$, and then mark off two overlapping regions, corresponding to the total functions and the computable partial functions, respectively. It is a good exercise to see if you can describe an object in each of the resulting regions in the diagram.

8.9 Computable Sets

We can extend the notion of computability from computable functions to computable sets:

Definition 8.7. Let S be a set of natural numbers. Then S is *computable* iff its characteristic function is. In other words, S is computable iff the function

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

is computable. Similarly, a relation $R(x_0, \ldots, x_{k-1})$ is computable if and only if its characteristic function is.

Computable sets are also called *decidable*.

Notice that we now have a number of notions of computability: for partial functions, for functions, and for sets. Do not get them confused! The Turing machine computing a partial function returns the output of the function, for input values at which the function is defined; the Turing machine computing a set returns either 1 or 0, after deciding whether or not the input value is in the set or not.

8.10 Computably Enumerable Sets

Definition 8.8. A set is *computably enumerable* if it is empty or the range of a computable function.

Computably enumarable sets are also called *recursively enumerable* instead. This is the original terminology, and today both are commonly used, as well as the abbreviations "c.e." and "r.e."

You should think about what the definition means, and why the terminology is appropriate. The idea is that if S is the range of the computable function f, then

$$S = \{f(0), f(1), f(2), \ldots\},$$

and so f can be seen as "enumerating" the elements of S. Note that according to the definition, f need not be an increasing function, i.e., the enumeration need not be in increasing order. In fact, f need not even be injective, so that the constant function $f(x) = 0$ enumerates the set $\{0\}$.

Any computable set is computably enumerable. To see this, suppose S is computable. If S is empty, then by definition it is computably enumerable. Otherwise, let a be any element of S. Define f by

$$f(x) = \begin{cases} x & \text{if } \chi_S(x) = 1 \\ a & \text{otherwise.} \end{cases}$$

Then f is a computable function, and S is the range of f.

8.11 Equivalent Defininitions of Computably Enumerable Sets

The following gives a number of important equivalent statements of what it means to be computably enumerable.

Theorem 8.9. *Let S be a set of natural numbers. Then the following are equivalent:*

1. *S is computably enumerable.*

2. *S is the range of a* partial *computable function.*

3. *S is empty or the range of a primitive recursive function.*

4. *S is the* domain *of a partial computable function.*

The first three clauses say that we can equivalently take any nonempty computably enumerable set to be enumerated by either a computable function, a partial computable function, or a primitive recursive function. The fourth clause tells us that if S is computably enumerable, then for some index e,

$$S = \{x : \varphi_e(x) \downarrow\}.$$

In other words, S is the set of inputs on for which the computation of φ_e halts. For that reason, computably enumerable sets are sometimes called *semi-decidable*: if a number is in the set, you eventually get a "yes," but if it isn't, you never get a "no"!

Proof. Since every primitive recursive function is computable and every computable function is partial computable, (3) implies (1) and (1) implies (2). (Note that if S is empty, S is the range of the partial computable function that is nowhere defined.) If we show that (2) implies (3), we will have shown the first three clauses equivalent.

So, suppose S is the range of the partial computable function φ_e. If S is empty, we are done. Otherwise, let a be any element of S. By Kleene's normal form theorem, we can write

$$\varphi_e(x) = U(\mu s\ T(e, x, s)).$$

In particular, $\varphi_e(x) \downarrow$ and $= y$ if and only if there is an s such that $T(e, x, s)$ and $U(s) = y$. Define $f(z)$ by

$$f(z) = \begin{cases} U((z)_1) & \text{if } T(e, (z)_0, (z)_1) \\ a & \text{otherwise.} \end{cases}$$

Then f is primitive recursive, because T and U are. Expressed in terms of Turing machines, if z codes a pair $\langle (z)_0, (z)_1 \rangle$ such that $(z)_1$ is a halting computation of machine e on input $(z)_0$, then f returns the output of the computation; otherwise, it returns a.We need to show that S is the range of f, i.e., for any natural number y, $y \in S$ if and only if it is in the range of f. In the forwards direction, suppose $y \in S$. Then y is in the range of φ_e, so for some x and s, $T(e, x, s)$ and $U(s) = y$; but then $y = f(\langle x, s \rangle)$. Conversely, suppose y is in the range of f. Then either $y = a$, or for some z, $T(e, (z)_0, (z)_1)$ and $U((z)_1) = y$. Since, in the latter case, $\varphi_e(x) \downarrow = y$, either way, y is in S.

(The notation $\varphi_e(x) \downarrow = y$ means "$\varphi_e(x)$ is defined and equal to y." We could just as well use $\varphi_e(x) = y$, but the extra arrow is sometimes helpful in reminding us that we are dealing with a partial function.)

To finish up the proof of Theorem 8.9, it suffices to show that (1) and (4) are equivalent. First, let us show that (1) implies (4). Suppose S is the range of a computable function f, i.e.,

$$S = \{y : \text{for some } x, f(x) = y\}.$$

Let

$$g(y) = \mu x\, f(x) = y.$$

Then g is a partial computable function, and $g(y)$ is defined if and only if for some x, $f(x) = y$. In other words, the domain of g is the range of f. Expressed in terms of Turing machines: given a Turing machine F that enumerates the elements of S, let G be the Turing machine that semi-decides S by searching through the outputs of F to see if a given element is in the set.

Finally, to show (4) implies (1), suppose that S is the domain of the partial computable function φ_e, i.e.,

$$S = \{x : \varphi_e(x) \downarrow\}.$$

If S is empty, we are done; otherwise, let a be any element of S. Define f by

$$f(z) = \begin{cases} (z)_0 & \text{if } T(e, (z)_0, (z)_1) \\ a & \text{otherwise.} \end{cases}$$

Then, as above, a number x is in the range of f if and only if $\varphi_e(x) \downarrow$, i.e., if and only if $x \in S$. Expressed in terms of Turing machines: given a machine M_e that semi-decides S, enumerate the elements of S by running through all possible Turing machine computations, and returning the inputs that correspond to halting computations. □

The fourth clause of Theorem 8.9 provides us with a convenient way of enumerating the computably enumerable sets: for each e, let W_e denote the domain of φ_e. Then if A is any computably enumerable set, $A = W_e$, for some e.

The following provides yet another characterization of the computably enumerable sets.

Theorem 8.10. *A set S is computably enumerable if and only if there is a computable relation $R(x, y)$ such that*

$$S = \{x : \exists y\, R(x, y)\}.$$

Proof. In the forward direction, suppose S is computably enumerable. Then for some e, $S = W_e$. For this value of e we can write S as

$$S = \{x : \exists y\, T(e, x, y)\}.$$

In the reverse direction, suppose $S = \{x : \exists y\, R(x,y)\}$. Define f by

$$f(x) \simeq \mu y\ AtomRx, y.$$

Then f is partial computable, and S is the domain of f. □

8.12 Computably Enumerable Sets are Closed under Union and Intersection

The following theorem gives some closure properties on the set of computably enumerable sets.

Theorem 8.11. *Suppose A and B are computably enumerable. Then so are $A \cap B$ and $A \cup B$.*

Proof. Theorem 8.9 allows us to use various characterizations of the computably enumerable sets. By way of illustration, we will provide a few different proofs.

For the first proof, suppose A is enumerated by a computable function f, and B is enumerated by a computable function g. Let

$$h(x) = \mu y\ (f(y) = x \vee g(y) = x) \text{ and}$$
$$j(x) = \mu y\ (f((y)_0) = x \wedge g((y)_1) = x).$$

Then $A \cup B$ is the domain of h, and $A \cap B$ is the domain of j.

Here is what is going on, in computational terms: given procedures that enumerate A and B, we can semi-decide if an element x is in $A \cup B$ by looking for x in either enumeration; and we can semi-decide if an element x is in $A \cap B$ for looking for x in both enumerations at the same time.

For the second proof, suppose again that A is enumerated by f and B is enumerated by g. Let

$$k(x) = \begin{cases} f(x/2) & \text{if } x \text{ is even} \\ g((x-1)/2) & \text{if } x \text{ is odd.} \end{cases}$$

Then k enumerates $A \cup B$; the idea is that k just alternates between the enumerations offered by f and g. Enumerating $A \cap B$ is tricker. If $A \cap B$ is empty, it is trivially computably enumerable. Otherwise, let c be any element of $A \cap B$, and define l by

$$l(x) = \begin{cases} f((x)_0) & \text{if } f((x)_0) = g((x)_1) \\ c & \text{otherwise.} \end{cases}$$

In computational terms, l runs through pairs of elements in the enumerations of f and g, and outputs every match it finds; otherwise, it just stalls by outputting c.

For the last proof, suppose A is the *domain* of the partial function $m(x)$ and B is the domain of the partial function $n(x)$. Then $A \cap B$ is the domain of the partial function $m(x) + n(x)$.

In computational terms, if A is the set of values for which m halts and B is the set of values for which n halts, $A \cap B$ is the set of values for which both procedures halt.

Expressing $A \cup B$ as a set of halting values is more difficult, because one has to simulate m and n in parallel. Let d be an index for m and let e be an index for n; in other words, $m = \varphi_d$ and $n = \varphi_e$. Then $A \cup B$ is the domain of the function

$$p(x) = \mu y \ (T(d, x, y) \vee T(e, x, y)).$$

In computational terms, on input x, p searches for either a halting computation for m or a halting computation for n, and halts if it finds either one. □

8.13 Computably Enumerable Sets not Closed under Complement

Suppose A is computably enumerable. Is the complement of A, $\overline{A} = \mathbb{N} \setminus A$, necessarily computably enumerable as well? The following theorem and corollary show that the answer is "no."

Theorem 8.12. *Let A be any set of natural numbers. Then A is computable if and only if both A and $\overline{A}$ are computably enumerable.*

Proof. The forwards direction is easy: if A is computable, then $\overline{A}$ is computable as well ($\chi_A = 1 \dot{-} \chi_{\overline{A}}$), and so both are computably enumerable.

In the other direction, suppose A and $\overline{A}$ are both computably enumerable. Let A be the domain of φ_d, and let $\overline{A}$ be the domain of φ_e. Define h by

$$h(x) = \mu s \ (T(d, x, s) \vee T(e, x, s)).$$

In other words, on input x, h searches for either a halting computation of φ_d or a halting computation of φ_e. Now, if $x \in A$, it will succeed in the first case, and if $x \in \overline{A}$, it will succeed in the second case. So, h is a total computable function. But now we have that for every x, $x \in A$ if and only if $T(e, x, h(x))$, i.e., if φ_e is the one that is defined. Since $T(e, x, h(x))$ is a computable relation, A is computable. □

It is easier to understand what is going on in informal computational terms: to decide A, on input x search for halting computations of φ_e and φ_f. One of them is bound to halt; if it is φ_e, then x is in A, and otherwise, x is in $\overline{A}$.

Corollary 8.13. *$\overline{K_0}$ is not computably enumerable.*

Proof. We know that K_0 is computably enumerable, but not computable. If $\overline{K_0}$ were computably enumerable, then K_0 would be computable by Theorem 8.12. □

8.14 Reducibility

We now know that there is at least one set, K_0, that is computably enumerable but not computable. It should be clear that there are others. The method of reducibility provides a powerful method of showing that other sets have these properties, without constantly having to return to first principles.

Generally speaking, a "reduction" of a set A to a set B is a method of transforming answers to whether or not elements are in B into answers as to whether or not elements are in A. We will focus on a notion called "many-one reducibility," but there are many other notions of reducibility available, with varying properties. Notions of reducibility are also central to the study of computational complexity, where efficiency issues have to be considered as well. For example, a set is said to be "NP-complete" if it is in NP and every NP problem can be reduced to it, using a notion of reduction that is similar to the one described below, only with the added requirement that the reduction can be computed in polynomial time.

We have already used this notion implicitly. Define the set K by

$$K = \{x : \varphi_x(x) \downarrow\},$$

i.e., $K = \{x : x \in W_x\}$. Our proof that the halting problem in unsolvable, Theorem 8.6, shows most directly that K is not computable. Recall that K_0 is the set

$$K_0 = \{\langle e, x\rangle : \varphi_e(x) \downarrow\}.$$

i.e. $K_0 = \{\langle x, e\rangle : x \in W_e\}$. It is easy to extend any proof of the uncomputability of K to the uncomputability of K_0: if K_0 were computable, we could decide whether or not an element x is in K simply by asking whether or not the pair $\langle x, x\rangle$ is in K_0. The function f which maps x to $\langle x, x\rangle$ is an example of a *reduction* of K to K_0.

Definition 8.14. Let A and B be sets. Then A is said to be *many-one reducible* to B, written $A \leq_m B$, if there is a computable function f such that for every natural number x,

$$x \in A \quad \text{if and only if} \quad f(x) \in B.$$

If A is many-one reducible to B and vice-versa, then A and B are said to be *many-one equivalent*, written $A \equiv_m B$.

If the function f in the definition above happens to be injective, A is said to be *one-one reducible* to B. Most of the reductions described below meet this stronger requirement, but we will not use this fact.

It is true, but by no means obvious, that one-one reducibility really is a stronger requirement than many-one reducibility. In other words, there are infinite sets A and B such that A is many-one reducible to B but not one-one reducible to B.

8.15 Properties of Reducibility

The intuition behind writing $A \leq_m B$ is that A is "no harder than" B. The following two propositions support this intuition.

Proposition 8.15. *If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$.*

Proof. Composing a reduction of A to B with a reduction of B to C yields a reduction of A to C. (You should check the details!) □

Proposition 8.16. *Let A and B be any sets, and suppose A is many-one reducible to B.*

1. *If B is computably enumerable, so is A.*
2. *If B is computable, so is A.*

Proof. Let f be a many-one reduction from A to B. For the first claim, just check that if B is the domain of a partial function g, then A is the domain of $g \circ f$:

$$\begin{aligned} x \in A &\text{iff } f(x) \in B \\ &\text{iff } g(f(x)) \downarrow . \end{aligned}$$

For the second claim, remember that if B is computable then B and $\overline{B}$ are computably enumerable. It is not hard to check that f is also a many-one reduction of $\overline{A}$ to $\overline{B}$, so, by the first part of this proof, A and $\overline{A}$ are computably enumerable. So A is computable as well. (Alternatively, you can check that $\chi_A = \chi_B \circ f$; so if χ_B is computable, then so is χ_A.) □

A more general notion of reducibility called *Turing reducibility* is useful in other contexts, especially for proving undecidability results. Note that by Corollary 8.13, the complement of K_0 is not reducible to K_0, since it is not computably enumerable. But, intuitively, if you knew the answers to questions about K_0, you would know the answer to questions about its complement as well. A set A is said to be Turing reducible to B if one can determine answers to questions in A using a computable procedure that can ask questions about B. This is more liberal than many-one reducibility, in which (1) you are only allowed to ask one question about B, and (2) a "yes" answer has to translate to a "yes" answer to the question about A, and similarly for "no." It is still the case that if A is Turing reducible to B and B is computable then A is

computable as well (though, as we have seen, the analogous statement does not hold for computable enumerability).

You should think about the various notions of reducibility we have discussed, and understand the distinctions between them. We will, however, only deal with many-one reducibility in this chapter. Incidentally, both types of reducibility discussed in the last paragraph have analogues in computational complexity, with the added requirement that the Turing machines run in polynomial time: the complexity version of many-one reducibility is known as *Karp reducibility*, while the complexity version of Turing reducibility is known as *Cook reducibility*.

8.16 Complete Computably Enumerable Sets

Definition 8.17. A set A is a *complete computably enumerable set* (under many-one reducibility) if

1. A is computably enumerable, and
2. for any other computably enumerable set B, $B \leq_m A$.

In other words, complete computably enumerable sets are the "hardest" computably enumerable sets possible; they allow one to answer questions about *any* computably enumerable set.

Theorem 8.18. *K, K_0, and K_1 are all complete computably enumerable sets.*

Proof. To see that K_0 is complete, let B be any computably enumerable set. Then for some index e,

$$B = W_e = \{x : \varphi_e(x) \downarrow\}.$$

Let f be the function $f(x) = \langle e, x\rangle$. Then for every natural number x, $x \in B$ if and only if $f(x) \in K_0$. In other words, f reduces B to K_0.

To see that K_1 is complete, note that in the proof of Proposition 8.19 we reduced K_0 to it. So, by Proposition 8.15, any computably enumerable set can be reduced to K_1 as well.

K can be reduced to K_0 in much the same way. □

So, it turns out that all the examples of computably enumerable sets that we have considered so far are either computable, or complete. This should seem strange! Are there any examples of computably enumerable sets that are neither computable nor complete? The answer is yes, but it wasn't until the middle of the 1950s that this was established by Friedberg and Muchnik, independently.

8.17 An Example of Reducibility

Let us consider an application of Proposition 8.16.

Proposition 8.19. *Let*

$$K_1 = \{e : \varphi_e(0) \downarrow\}.$$

Then K_1 is computably enumerable but not computable.

Proof. Since $K_1 = \{e : \exists s\, T(e, 0, s)\}$, K_1 is computably enumerable by Theorem 8.10.

To show that K_1 is not computable, let us show that K_0 is reducible to it.

This is a little bit tricky, since using K_1 we can only ask questions about computations that start with a particular input, 0. Suppose you have a smart friend who can answer questions of this type (friends like this are known as "oracles"). Then suppose someone comes up to you and asks you whether or not $\langle e, x\rangle$ is in K_0, that is, whether or not machine e halts on input x. One thing you can do is build another machine, e_x, that, for *any* input, ignores that input and instead runs e on input x. Then clearly the question as to whether machine e halts on input x is equivalent to the question as to whether machine e_x halts on input 0 (or any other input). So, then you ask your friend whether this new machine, e_x, halts on input 0; your friend's answer to the modified question provides the answer to the original one. This provides the desired reduction of K_0 to K_1.

Using the universal partial computable function, let f be the 3-ary function defined by

$$f(x, y, z) \simeq \varphi_x(y).$$

Note that f ignores its third input entirely. Pick an index e such that $f = \varphi^3_e$; so we have

$$\varphi^3_e(x, y, z) \simeq \varphi_x(y).$$

By the *s-m-n* theorem, there is a function $s(e, x, y)$ such that, for every z,

$$\begin{aligned}\varphi_{s(e,x,y)}(z) &\simeq \varphi^3_e(x, y, z)\\ &\simeq \varphi_x(y).\end{aligned}$$

In terms of the informal argument above, $s(e, x, y)$ is an index for the machine that, for any input z, ignores that input and computes $\varphi_x(y)$.

In particular, we have

$$\varphi_{s(e,x,y)}(0) \downarrow \quad \text{if and only if} \quad \varphi_x(y) \downarrow .$$

In other words, $\langle x, y\rangle \in K_0$ if and only if $s(e, x, y) \in K_1$. So the function g defined by

$$g(w) = s(e, (w)_0, (w)_1)$$

is a reduction of K_0 to K_1. □

8.18 Totality is Undecidable

Let us consider one more example of using the *s-m-n* theorem to show that something is noncomputable. Let Tot be the set of indices of total computable functions, i.e.

$$\text{Tot} = \{x : \text{for every } y,\ \varphi_x(y) \downarrow\}.$$

Proposition 8.20. Tot *is not computable.*

Proof. To see that Tot is not computable, it suffices to show that K is reducible to it. Let $h(x, y)$ be defined by

$$h(x,y) \simeq \begin{cases} 0 & \text{if } x \in K \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that $h(x, y)$ does not depend on y at all. It should not be hard to see that h is partial computable: on input x, y, the we compute h by first simulating the function φ_x on input x; if this computation halts, $h(x, y)$ outputs 0 and halts. So $h(x, y)$ is just $Z(\mu s\ T(x, x, s))$, where Z is the constant zero function.

Using the *s-m-n* theorem, there is a primitive recursive function $k(x)$ such that for every x and y,

$$\varphi_{k(x)}(y) = \begin{cases} 0 & \text{if } x \in K \\ \text{undefined} & \text{otherwise} \end{cases}$$

So $\varphi_{k(x)}$ is total if $x \in K$, and undefined otherwise. Thus, k is a reduction of K to Tot. □

It turns out that Tot is not even computably enumerable—its complexity lies further up on the "arithmetic hierarchy." But we will not worry about this strengthening here.

8.19 Rice's Theorem

If you think about it, you will see that the specifics of Tot do not play into the proof of Proposition 8.20. We designed $h(x, y)$ to act like the constant function $j(y) = 0$ exactly when x is in K; but we could just as well have made it act like any other partial computable function under those circumstances. This observation lets us state a more general theorem, which says, roughly, that no nontrivial property of computable functions is decidable.

Keep in mind that $\varphi_0, \varphi_1, \varphi_2, \ldots$ is our standard enumeration of the partial computable functions.

Theorem 8.21 (Rice's Theorem)**.** *Let C be any set of partial computable functions, and let $A = \{n : \varphi_n \in C\}$. If A is computable, then either C is $\emptyset$ or C is the set of all the partial computable functions.*

An *index set* is a set A with the property that if n and m are indices which "compute" the same function, then either both n and m are in A, or neither is. It is not hard to see that the set A in the theorem has this property. Conversely, if A is an index set and C is the set of functions computed by these indices, then $A = \{n : \varphi_n \in C\}$.

With this terminology, Rice's theorem is equivalent to saying that no nontrivial index set is decidable. To understand what the theorem says, it is helpful to emphasize the distinction between *programs* (say, in your favorite programming language) and the functions they compute. There are certainly questions about programs (indices), which are syntactic objects, that are computable: does this program have more than 150 symbols? Does it have more than 22 lines? Does it have a "while" statement? Does the string "hello world" every appear in the argument to a "print" statement? Rice's theorem says that no nontrivial question about the program's *behavior* is computable. This includes questions like these: does the program halt on input 0? Does it ever halt? Does it ever output an even number?

Proof of Rice's theorem. Suppose C is neither $\emptyset$ nor the set of all the partial computable functions, and let A be the set of indices of functions in C. We will show that if A were computable, we could solve the halting problem; so A is not computable.

Without loss of generality, we can assume that the function f which is nowhere defined is not in C (otherwise, switch C and its complement in the argument below). Let g be any function in C. The idea is that if we could decide A, we could tell the difference between indices computing f, and indices computing g; and then we could use that capability to solve the halting problem.

Here's how. Using the universal computation predicate, we can define a function

$$h(x,y) \simeq \begin{cases} \text{undefined} & \text{if } \varphi_x(x) \uparrow \\ g(y) & \text{otherwise.} \end{cases}$$

To compute h, first we try to compute $\varphi_x(x)$; if that computation halts, we go on to compute $g(y)$; and if *that* computation halts, we return the output. More formally, we can write

$$h(x,y) \simeq P^2_0(g(y), \mathrm{Un}(x,x)).$$

where $P^2_0(z_0, z_1) = z_0$ is the 2-place projection function returning the 0-th argument, which is computable.

Then h is a composition of partial computable functions, and the right side is defined and equal to $g(y)$ just when $\mathrm{Un}(x,x)$ and $g(y)$ are both defined.

Notice that for a fixed x, if $\varphi_x(x)$ is undefined, then $h(x,y)$ is undefined for every y; and if $\varphi_x(x)$ is defined, then $h(x,y) \simeq g(y)$. So, for any fixed value of x, either $h(x,y)$ acts just like f or it acts just like g, and deciding whether or

not $\varphi_x(x)$ is defined amounts to deciding which of these two cases holds. But this amounts to deciding whether or not $h_x(y) \simeq h(x,y)$ is in C or not, and if A were computable, we could do just that.

More formally, since h is partial computable, it is equal to the function φ_k for some index k. By the *s-m-n* theorem there is a primitive recursive function s such that for each x, $\varphi_{s(k,x)}(y) = h_x(y)$. Now we have that for each x, if $\varphi_x(x) \downarrow$, then $\varphi_{s(k,x)}$ is the same function as g, and so $s(k,x)$ is in A. On the other hand, if $\varphi_x(x) \uparrow$, then $\varphi_{s(k,x)}$ is the same function as f, and so $s(k,x)$ is not in A. In other words we have that for every x, $x \in K$ if and only if $s(k,x) \in A$. If A were computable, K would be also, which is a contradiction. So A is not computable. □

Rice's theorem is very powerful. The following immediate corollary shows some sample applications.

Corollary 8.22. *The following sets are undecidable.*

1. $\{x : 17 \text{ is in the range of } \varphi_x\}$
2. $\{x : \varphi_x \text{ is constant}\}$
3. $\{x : \varphi_x \text{ is total}\}$
4. $\{x : \text{whenever } y < y', \varphi_x(y) \downarrow, \text{ and if } \varphi_x(y') \downarrow, \text{ then } \varphi_x(y) < \varphi_x(y')\}$

Proof. These are all nontrivial index sets. □

8.20 The Fixed-Point Theorem

Let's consider the halting problem again. As temporary notation, let us write $\ulcorner \varphi_x(y) \urcorner$ for $\langle x, y \rangle$; think of this as representing a "name" for the value $\varphi_x(y)$. With this notation, we can reword one of our proofs that the halting problem is undecidable.

Question: is there a computable function h, with the following property? For every x and y,

$$h(\ulcorner \varphi_x(y) \urcorner) = \begin{cases} 1 & \text{if } \varphi_x(y) \downarrow \\ 0 & \text{otherwise.} \end{cases}$$

Answer: No; otherwise, the partial function

$$g(x) \simeq \begin{cases} 0 & \text{if } h(\ulcorner \varphi_x(x) \urcorner) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

would be computable, and so have some index e. But then we have

$$\varphi_e(e) \simeq \begin{cases} 0 & \text{if } h(\ulcorner \varphi_e(e) \urcorner) = 0 \\ \text{undefined} & \text{otherwise,} \end{cases}$$

in which case $\varphi_e(e)$ is defined if and only if it isn't, a contradiction.

Now, take a look at the equation with φ_e. There is an instance of self-reference there, in a sense: we have arranged for the value of $\varphi_e(e)$ to depend on $\ulcorner \varphi_e(e) \urcorner$, in a certain way. The fixed-point theorem says that we *can* do this, in general—not just for the sake of proving contradictions.

Lemma 8.23 gives two equivalent ways of stating the fixed-point theorem. Logically speaking, the fact that the statements are equivalent follows from the fact that they are both true; but what we really mean is that each one follows straightforwardly from the other, so that they can be taken as alternative statements of the same theorem.

Lemma 8.23. *The following statements are equivalent:*

1. *For every partial computable function $g(x, y)$, there is an index e such that for every y,*
$$\varphi_e(y) \simeq g(e, y).$$
2. *For every computable function $f(x)$, there is an index e such that for every y,*
$$\varphi_e(y) \simeq \varphi_{f(e)}(y).$$

Proof. (1) $\Rightarrow$ (2): Given f, define g by $g(x, y) \simeq \mathrm{Un}(f(x), y)$. Use (1) to get an index e such that for every y,
$$\begin{aligned} \varphi_e(y) &= \mathrm{Un}(f(e), y) \\ &= \varphi_{f(e)}(y). \end{aligned}$$

(2) $\Rightarrow$ (1): Given g, use the *s-m-n* theorem to get f such that for every x and y, $\varphi_{f(x)}(y) \simeq g(x, y)$. Use (2) to get an index e such that
$$\begin{aligned} \varphi_e(y) &= \varphi_{f(e)}(y) \\ &= g(e, y). \end{aligned}$$

This concludes the proof. □

Before showing that statement (1) is true (and hence (2) as well), consider how bizarre it is. Think of e as being a computer program; statement (1) says that given any partial computable $g(x, y)$, you can find a computer program e that computes $g_e(y) \simeq g(e, y)$. In other words, you can find a computer program that computes a function that references the program itself.

Theorem 8.24. *The two statements in Lemma 8.23 are true. Specifically, for every partial computable function $g(x, y)$, there is an index e such that for every y,*
$$\varphi_e(y) \simeq g(e, y).$$

Proof. The ingredients are already implicit in the discussion of the halting problem above. Let $\mathrm{diag}(x)$ be a computable function which for each x returns an index for the function $f_x(y) \simeq \varphi_x(x, y)$, i.e.

$$\varphi_{\mathrm{diag}(x)}(y) \simeq \varphi_x(x, y).$$

Think of diag as a function that transforms a program for a 2-ary function into a program for a 1-ary function, obtained by fixing the original program as its first argument. The function diag can be defined formally as follows: first define s by

$$s(x, y) \simeq \mathrm{Un}^2(x, x, y),$$

where Un^2 is a 3-ary function that is universal for partial computable 2-ary functions. Then, by the *s-m-n* theorem, we can find a primitive recursive function diag satisfying

$$\varphi_{\mathrm{diag}(x)}(y) \simeq s(x, y).$$

Now, define the function l by

$$l(x, y) \simeq g(\mathrm{diag}(x), y).$$

and let $\ulcorner l \urcorner$ be an index for l. Finally, let $e = \mathrm{diag}(\ulcorner l \urcorner)$. Then for every y, we have

$$\begin{aligned}
\varphi_e(y) &\simeq \varphi_{\mathrm{diag}(\ulcorner l \urcorner)}(y) \\
&\simeq \varphi_{\ulcorner l \urcorner}(\ulcorner l \urcorner, y) \\
&\simeq l(\ulcorner l \urcorner, y) \\
&\simeq g(\mathrm{diag}(\ulcorner l \urcorner), y) \\
&\simeq g(e, y),
\end{aligned}$$

as required. □

What's going on? Suppose you are given the task of writing a computer program that prints itself out. Suppose further, however, that you are working with a programming language with a rich and bizarre library of string functions. In particular, suppose your programming language has a function diag which works as follows: given an input string s, diag locates each instance of the symbol 'x' occuring in s, and replaces it by a quoted version of the original string. For example, given the string

```
hello x world
```

as input, the function returns

```
hello 'hello x world' world
```

as output. In that case, it is easy to write the desired program; you can check that

```
print(diag('print(diag(x))'))
```

does the trick. For more common programming languages like C++ and Java, the same idea (with a more involved implementation) still works.

We are only a couple of steps away from the proof of the fixed-point theorem. Suppose a variant of the print function print(x, y) accepts a string x and another numeric argument y, and prints the string x repeatedly, y times. Then the "program"

```
getinput(y); print(diag('getinput(y); print(diag(x), y)'), y)
```

prints itself out y times, on input y. Replacing the getinput—print—diag skeleton by an arbitrary funtion $g(x, y)$ yields

```
g(diag('g(diag(x), y)'), y)
```

which is a program that, on input y, runs g on the program itself and y. Thinking of "quoting" with "using an index for," we have the proof above.

For now, it is o.k. if you want to think of the proof as formal trickery, or black magic. But you should be able to reconstruct the details of the argument given above. When we prove the incompleteness theorems (and the related "fixed-point theorem") we will discuss other ways of understanding why it works.

The same idea can be used to get a "fixed point" combinator. Suppose you have a lambda term g, and you want another term k with the property that k is β-equivalent to gk. Define terms

$$\text{diag}(x) = xx$$

and

$$l(x) = g(\text{diag}(x))$$

using our notational conventions; in other words, l is the term $\lambda x.\, g(xx)$. Let k be the term ll. Then we have

$$\begin{aligned} k &= (\lambda x.\, g(xx))(\lambda x.\, g(xx)) \\ &\triangleright g((\lambda x.\, g(xx))(\lambda x.\, g(xx))) \\ &= gk. \end{aligned}$$

If one takes

$$Y = \lambda g.\, ((\lambda x.\, g(xx))(\lambda x.\, g(xx)))$$

then Yg and $g(Yg)$ reduce to a common term; so $Yg \equiv_\beta g(Yg)$. This is known as "Curry's combinator." If instead one takes

$$Y = (\lambda xg.\, g(xxg))(\lambda xg.\, g(xxg))$$

then in fact Yg reduces to $g(Yg)$, which is a stronger statement. This latter version of Y is known as "Turing's combinator."

8.21 Applying the Fixed-Point Theorem

The fixed-point theorem essentially lets us define partial computable functions in terms of their indices. For example, we can find an index e such that for every y,

$$\varphi_e(y) = e + y.$$

As another example, one can use the proof of the fixed-point theorem to design a program in Java or C++ that prints itself out.

Remember that if for each e, we let W_e be the domain of φ_e, then the sequence $W_0, W_1, W_2, \ldots$ enumerates the computably enumerable sets. Some of these sets are computable. One can ask if there is an algorithm which takes as input a value x, and, if W_x happens to be computable, returns an index for its characteristic function. The answer is "no," there is no such algorithm:

Theorem 8.25. *There is no partial computable function f with the following property: whenever W_e is computable, then $f(e)$ is defined and $\varphi_{f(e)}$ is its characteristic function.*

Proof. Let f be any computable function; we will construct an e such that W_e is computable, but $\varphi_{f(e)}$ is not its characteristic function. Using the fixed point theorem, we can find an index e such that

$$\varphi_e(y) \simeq \begin{cases} 0 & \text{if } y = 0 \text{ and } \varphi_{f(e)}(0) \downarrow = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

That is, e is obtained by applying the fixed-point theorem to the function defined by

$$g(x, y) \simeq \begin{cases} 0 & \text{if } y = 0 \text{ and } \varphi_{f(x)}(0) \downarrow = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Informally, we can see that g is partial computable, as follows: on input x and y, the algorithm first checks to see if y is equal to 0. If it is, the algorithm computes $f(x)$, and then uses the universal machine to compute $\varphi_{f(x)}(0)$. If this last computation halts and returns 0, the algorithm returns 0; otherwise, the algorithm doesn't halt.

But now notice that if $\varphi_{f(e)}(0)$ is defined and equal to 0, then $\varphi_e(y)$ is defined exactly when y is equal to 0, so $W_e = \{0\}$. If $\varphi_{f(e)}(0)$ is not defined, or is defined but not equal to 0, then $W_e = \emptyset$. Either way, $\varphi_{f(e)}$ is not the characteristic function of W_e, since it gives the wrong answer on input 0. □

8.22 Defining Functions using Self-Reference

It is generally useful to be able to define functions in terms of themselves. For example, given computable functions k, l, and m, the fixed-point lemma tells us that there is a partial computable function f satisfying the following equation for every y:

$$f(y) \simeq \begin{cases} k(y) & \text{if } l(y) = 0 \\ f(m(y)) & \text{otherwise.} \end{cases}$$

Again, more specifically, f is obtained by letting

$$g(x, y) \simeq \begin{cases} k(y) & \text{if } l(y) = 0 \\ \varphi_x(m(y)) & \text{otherwise} \end{cases}$$

and then using the fixed-point lemma to find an index e such that $\varphi_e(y) = g(e, y)$.

For a concrete example, the "greatest common divisor" function $\gcd(u, v)$ can be defined by

$$\gcd(u, v) \simeq \begin{cases} v & \text{if } 0 = 0 \\ \gcd(\text{mod}(v, u), u) & \text{otherwise} \end{cases}$$

where $\text{mod}(v, u)$ denotes the remainder of dividing v by u. An appeal to the fixed-point lemma shows that gcd is partial computable. (In fact, this can be put in the format above, letting y code the pair $\langle u, v \rangle$.) A subsequent induction on u then shows that, in fact, gcd is total.

Of course, one can cook up self-referential definitions that are much fancier than the examples just discussed. Most programming languages support definitions of functions in terms of themselves, one way or another. Note that this is a little bit less dramatic than being able to define a function in terms of an *index* for an algorithm computing the functions, which is what, in full generality, the fixed-point theorem lets you do.

8.23 Minimization with Lambda Terms

When it comes to the lambda calculus, we've shown the following:

1. Every primitive recursive function is represented by a lambda term.
2. There is a lambda term Y such that for any lambda term G, $YG \triangleright G(YG)$.

To show that every partial computable function is represented by some lambda term, we only need to show the following.

Lemma 8.26. *Suppose $f(x, y)$ is primitive recursive. Let g be defined by*

$$g(x) \simeq \mu y\ f(x, y) = 0.$$

Then g is represented by a lambda term.

Proof. The idea is roughly as follows. Given x, we will use the fixed-point lambda term Y to define a function $h_x(n)$ which searches for a y starting at n; then $g(x)$ is just $h_x(0)$. The function h_x can be expressed as the solution of a fixed-point equation:

$$h_x(n) \simeq \begin{cases} n & \text{if } f(x, n) = 0 \\ h_x(n+1) & \text{otherwise.} \end{cases}$$

Here are the details. Since f is primitive recursive, it is represented by some term F. Remember that we also have a lambda term D such that $D(M, N, \overline{0}) \triangleright M$ and $D(M, N, \overline{1}) \triangleright N$. Fixing x for the moment, to represent h_x we want to find a term H (depending on x) satisfying

$$H(\overline{n}) \equiv D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})).$$

We can do this using the fixed-point term Y. First, let U be the term

$$\lambda h.\, \lambda z.\, D(z, (h(Sz)), F(x, z)),$$

and then let H be the term YU. Notice that the only free variable in H is x. Let us show that H satisfies the equation above.

By the definition of Y, we have

$$H = YU \equiv U(YU) = U(H).$$

In particular, for each natural number n, we have

$$\begin{aligned} H(\overline{n}) &\equiv U(H, \overline{n}) \\ &\triangleright D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})), \end{aligned}$$

as required. Notice that if you substitute a numeral $\overline{m}$ for x in the last line, the expression reduces to $\overline{n}$ if $F(\overline{m}, \overline{n})$ reduces to $\overline{0}$, and it reduces to $H(S(\overline{n}))$ if $F(\overline{m}, \overline{n})$ reduces to any other numeral.

To finish off the proof, let G be $\lambda x.\, H(\overline{0})$. Then G represents g; in other words, for every m, $G(\overline{m})$ reduces to reduces to $\overline{g(m)}$, if $g(m)$ is defined, and has no normal form otherwise. □

Part III

Turing Machines

Chapter 9

Turing Machine Computations

9.1 Introduction

Even though the term "Turing machine" evokes the image of a physical machine with moving parts, strictly speaking a Turing machine is a purely mathematical construct. It is perhaps best to think of a Turing machine as a program for a special kind of imaginary mechanism. This mechanism consists of a *tape* and a *read-write head*. In our version of Turing machines, the tape is infinite in one direction (to the right), and it is divided into *squares*, each of which may contain a symbol from a finite *alphabet*. Such alphabets can contain any number of different symbols, but we will mainly make do with three: $\triangleright$, $\sqcup$, and $|$. When he mechanism is started, the tape is empty (i.e., each square contains the symbol $\sqcup$) except for the leftmost square, which contains $\triangleright$, and a finite number of squares which contain the *input*. At any time, the mechanism is in one of a finite number of *states*. At the outset, the head scans the leftmost square and in a specified *initial state*. At each step of the mechanism's run, the content of the square currently scanned together with the state the mechanism is in and the Turing machine program determine what happens next. The Turing machine program consists of a list of 5-tuples $\langle q_i, \sigma, q_j, \sigma', D\rangle$. Whenever the mechanism is in state q_i and reads symbol σ, it replaces the symbol on the current square with σ', the head moves left, right, or stays put according to whether D is L, R, or N, and the mechanism goes into state q_j. When the mechanism enters state h we say it *halts*, and the contents of the tape at that point is its *output*.

9.2 Turing Machines

The formal definition of what constitutes a Turing machine looks abstract, but is actually simple: it merely packs into one mathematical structure all the information needed to specify the workings of a Turing machine. This includes (1) which states the machine can be in, (2) which symbols are allowed

to be on the tape, (3) which state the machine should start in, and (4) what the instruction set of the machine is.

Definition 9.1. A *Turing machine* $T = \langle Q, \Sigma, s, I \rangle$ consists of

1. a finite set of states Q which includes the *halting state* h,
2. a finite alphabet Σ which includes $\triangleright$ and $\llcorner$,
3. an initial state $s \in Q$,
4. a finite instruction set $I \subseteq Q \times \Sigma \times Q \times \Sigma \times \{L, R, N\}$.

We assume that the tape is infinite in one direction only. For this reason it is useful to designate a special symbol $\triangleright$ as a marker for the left end of the tape. This makes it easier for Turing machne programs to tell when they're "in danger" of running off the tape. Other definitions of Turing machines are possible, including one where the tape is infinite in both directions. In that case, marker for the left end of the tape is not necessary.

9.3 Configurations and Computations

The imaginary mechanism consisting of tape, read/write head, and Turing machine program is really just in intuitive way of visualizing what a Turing machine computation is. Formally, we can define the computation of a Turing machine on a given input as a sequence of *configurations*—and a configuration in turn is a sequence of symbols (corresponding to the contents of the tape at a given point in the computation), a number indicating the position of the read/write head, and a state. Using these, we can define what the Turing machine M computes on a given input.

Definition 9.2. A *configuration* of Turing machine $M = \langle Q, \Sigma, s, I \rangle$ is a triple $\langle C, n, q \rangle$ where

1. $C \in \Sigma^*$ is a finite sequence of symbols from Σ,
2. $n \in \mathbb{N}$ is a number $\leq \text{len}(C)$, and
3. $q \in Q$

The potential input for a Turing machine is a sequence of symbols, usually a sequence that encodes a number in some form. The initial configuration of the Turing machine is that configuration in which we start the Turing machine to work on that input: the tape contains the tape end marker immediately followed by the input written on the squares to the right, the read/write head is scanning the leftmost square of the tape (i.e., the left end marker), and the mechanism is in the designated start state s.

Definition 9.3. The *initial configuration* of M for input $I \in \Sigma^*$ is

$$\langle \triangleright \frown I, 0, s \rangle$$

Definition 9.4. We say that a configuration $\langle C, n, q \rangle$ *yields* $\langle C', n', q' \rangle$ *in one step* (according to M), iff

1. the n-th symbol of C is σ,
2. the instruction set of M contains a tuple $\langle q, \sigma, q', \sigma', D \rangle$,
3. the n-th symbol of C' is $sigma'$,
4. a) $D = L$ and $n' = n - 1$, or
   b) $D = R$ and $n' = n$, or
   c) $D = N$ and $n' = n$,
5. for all $i \neq n$, $C'(i) = C(i)$,
6. if $n' > \text{len}(C)$, then $\text{len}(C') = \text{len}(C) + 1$ and the n'-th symbol of C' is $\sqcup$.

Definition 9.5. A *run of M on input I* is a sequence C_i of configurations of M, where C_0 is the initial configuration of M for input I, and each C_i yields C_{i+1} in one step.

We say that *M halts on input I after k steps* if $C_k = \langle \triangleright \frown O, n, h \rangle$. In that case the *output* of M for input I is O.

9.4 Unary Representation of Numbers

Turing machines work on sequences of symbols written on their tape. Depending on the alphabet a Turing machine uses, these sequences of symbols can represent various inputs and outputs. Of particular interest, of course, are Turing machines which compute *arithmetical* functions, i.e., functions of natural numbers. A simple way to represent positive integers is by coding them as sequences of a single symbol $|$.

Definition 9.6. A Turing machine M *computes* the function $f\colon \mathbb{N}^n \to \mathbb{N}$ iff M halts on input

$$|^{k_1} \sqcup |^{k_2} \sqcup \ldots \sqcup |^{k_n}$$

with output $|^{f(k_1,\ldots,k_n)}$.

Chapter 10

Undecidability

10.1 Decision Problems

We have a number of important logical notions, such as those of satisfiability, validity, and consequence, which are properties of (sets) formulas or relations between them. The definitions of these logical notions provide *criteria* for when, say, a sentence is valid. (In this case, it is valid iff it is satisfied in every first-order structure.) These criteria, however, do not in general provide a *method for deciding* if a sentence or set of sentences has the property in question. The question of whether there is such a method for a given notion and class of formulas or sentences is called a *decision problem*.

Example 10.1. The *decision problem for validity of sentences in a first-order language* is the question of whether there is a procedure for deciding, given a sentence in a first-order language, if it is valid or not.

Example 10.2. The *decision problem for satisfiability of sentences in a first-order language* is the question of whether there is a procedure for deciding, given a sentence in a first-order language, if it is satisfiable or not.

We say that a decision problem is *solvable* if there is such a procedure, and *unsolvable* otherwise.

To show that a decision problem is solvable, you typically simply write down the procedure that solves it (and prove that it in fact solves it correctly in every case). Proving that a decision problem is unsolvable is a lot harder: you have to show that there can *in principle* be no procedure whatsoever that solves it. This can only be done rigorously if there is a precise definition of what a "procedure" is.

One such precise definition is provided by the notion of a *Turing machine*. A decision problem is solvable by a Turing machine if there is a Turing machine which, when started on the description of an instance of the decision problem as input, eventually halts with simply "1" or "0" on the tape, representing

"yes" or "no", respectively. To show that a decision problem is not solvable it suffices to show that no such Turing machine exists.

There are decision problems that can relatively easily be shown to be unsolvable by Turing machines, such as the Halting problem. To show logical decision problems undecidable, you show that a known unsolvable problem can be "reduced" to it. A decision problem A can be reduced to a decision problem B iff the answer to an instance of A can be obtained by an answer to an instance of B. For instance, the problem of validity of a sentence can be reduced to that of satisfiability: Given a sentence φ, form the sentence $\neg\varphi$: the former is valid iff the latter is not satisfiable. So the decision problem for validity can be solved by taking an instance, transforming that instance into its negation by putting "$\neg$" in front of it, obtaining the answer to "Is $\neg\varphi$ satisfiable?," and switching the answer. This shows that the problem of validity can be reduced to the problem of satisfiability.

A reduction of a decision problem A to a decision problem B shows two things:

1. If B is solvable, then A is solvable.

2. If A is unsolvable, then B is unsolvable.

A decision problem can therefore be shown to be unsolvable by reducing a known unsolvable problem, such as the Halting Problem, to it.

10.2 Representing Turing Machines

In order to represent Turing machines and their behavior by a sentence of first-order logic, we have to define a suitable language. The language consists of two parts: predicates for describing configurations of the machine, and expressions for counting execution steps ("moments") and positions on the tape. The latter require an initial moment, o, a "successor" function which is traditionally written as a postfix $'$, and an ordering $x<y$ of "before."

Definition 10.3. Given a Turing machine $M = \langle Q, \Sigma, \delta, s\rangle$, the language $\mathcal{L}_M$ consists of:

1. A two-place predicate $Q_q(x, y)$ for every state $q \in Q$.

2. A two-place predicate $S_\sigma(x, y)$ for every symbol $\sigma \in \Sigma$

3. A constant o

4. A one-place function $'$

5. A two-place predicate $<$

For each number n there is a canonical term $\overline{n}$, the *numeral* for n, which represents it in $\mathcal{L}_M$. $\overline{0}$ is o, $\overline{1}$ is o', $\overline{2}$ is o'', and so on. More formally:

$$\overline{0} = \mathsf{o}$$
$$\overline{n+1} = \overline{n}'$$

The sentences describing the operation of the Turing machine M on input w are the following:

I Axioms describing numbers:

a) A sentence that says that the successor function is injective:

$$\forall x \forall y (x' = y' \to x = y)$$

b) A sentence that says that every number is less than its successor:

$$\forall x (x < x')$$

c) A sentence that ensures that $<$ is transitive:

$$\forall x \forall y \forall z ((x < y \wedge y < z) \to x < z)$$

II. Axioms describing the input configuration:

a) M is in the inital state s at time 0, scanning square 0:

$$Q_s(\mathsf{o}, \mathsf{o})$$

b) The first n squares contain the symbols $\triangleright, \sigma_{i_1}, \ldots, \sigma_{i_n}$:

$$S_{\triangleright}(\mathsf{o}, \mathsf{o}) \wedge S_{i_1}(\overline{1}, \mathsf{o}) \wedge \cdots \wedge S_{i_n}(\overline{n}, \mathsf{o})$$

c) Otherwise, the tape is empty:

$$\forall x (\overline{n} < x \to S_{\sqcup}(x, \mathsf{o})$$

III. Axioms describing the transition from one configuration to the next:

For the following, let $A(x, y)$ be the conjunction of all sentences of the form

$$\forall z ((z < x \vee x < z \wedge S_\sigma(z, y)) \to S_\sigma(z, y'))$$

where $\sigma \in \Sigma$.

1. For every instruction $\langle q_i, \sigma, q_j, \sigma', L \rangle$, the sentence:

$$\forall x \forall y ((Q_{q_i}(x', y) \wedge S_\sigma(x, y)) \to (Q_{q_j}(x, y) \wedge S_{\sigma'}(x, y) \wedge A(x, y)))$$

2. For every instruction $\langle q_i, \sigma, q_j, \sigma', R\rangle$, the sentence:

$$\forall x \forall y((Q_{q_i}(x,y) \wedge S_\sigma(x,y)) \rightarrow (Q_{q_j}(x',y) \wedge S_{\sigma'}(x,y) \wedge A(x,y)))$$

(Probably also need axioms saying every square as exactly one symbol on it at all times, machine always in exactly one state.)

Let $T(M,w)$ be the conjunction of all the above sentences for Turing machine M and input w

The sentence $H(M,w)$:

$$\exists x \exists y\, Q_h(x,y)$$

expresses that the Turing machine M halts on input w.

10.3 Verifying the Representation

In order to verify that our representation works, we first have to make sure that if M halts on input w, then $T(m,w) \rightarrow H(M,w)$ is valid. We can do this simly by proving that $T(m,w)$ implies a descriptin of the configuration of M for each step of the execution of M on input w. If M halts on input w, then for some n, M will be in a halting configuration at step n (and be scanning square m, for some m). Hence, $T(M,w)$ implies $Q_h(\overline{m},\overline{n})$.

Definition 10.4. Let $C(M,w,n)$ be the sentence

$$Q_q(\overline{m},\overline{n}) \wedge S_{\sigma_0}(\overline{0},\overline{n}) \wedge \cdots \wedge S_{\sigma_k}(\overline{k},\overline{n})$$

where q is the state of M at time n, M is scanning sqare m at time n, square i contains symbol σ_i at time n for $0 \leq i \leq k$ and k is the right-most non-blank square of the tape at time m.

Lemma 10.5. *For each n, $T(M,w)$ implies $C(M,w,n)$.*

Proof. By induction on n.

If $n = 0$, then $C(M,w,n)$ is a conjunct of $T(M,w)$, so implied by it.

Suppose $n > 0$ and at time n, M started on w is in state q scanning square m, and the content of the tape is $\sigma_0, \ldots, \sigma_k$.

... to be completed

□

To complete the verification of our clam, we also have to establish the reverse direction: if $T(M,w) \rightarrow H(M,w)$ is valid, then M does in fact halt when started on input m.

Lemma 10.6. *If $T(M,w)$ entails $H(M,w)$, then M halts on input w.*

Proof. Consider the $\mathcal{L}_M$-structure $\mathfrak{M}$ with domain $\mathbb{N}$ which inerprets o as 0, $'$ as the successor function, and $<$ as the lest-than relation, and the predicates Q_q and S_σ as follows:

$$Q_q^{\mathfrak{M}} = \{(m,n) : \text{after } n \text{ steps, } M \text{ started on } w \text{ is in state } q \text{ scanning square } m\}$$
$$S_\sigma^{\mathfrak{M}} = \{(m,n) : \text{after } n \text{ steps, } M \text{ started on } w \text{ has symbol } \sigma \text{ on square } m\}$$

Clearly, $\mathfrak{M} \models T(M,w)$. If $T(M,w)$ implies $H(M,w)$, then $\mathfrak{M} \models H(M,w)$, i.e.,

$$\mathfrak{M} \models \exists x \exists y\, Q_h(x,y).$$

As $|\mathfrak{M}| = \mathbb{N}$, there must be $m, n \in \mathbb{N}$ so that $\mathfrak{M} \models Q_h(\overline{n}, \overline{m})$. By the definition of $\mathfrak{M}$, this means that M started on input w is in state h after m steps, i.e., has halted. □

Part IV

Incompleteness

Chapter 11

Arithmetization of Syntax

11.1 Introduction

In order to connect computability and logic, we need a way to talk about the objects of logic (symbols, terms, formulas, derivations), operations on them, and their properties and relations, in a way amenable to computational treatment. We can do this directly, by considering computable functions and relations on symbols, sequences of symbols, and other objects built from them. Since the objects of logical syntax are all finite and built from an enumerable sets of symbols, this is possible for some models of computation. But other models of computation are restricted to numbers, their relations and functions. Moreover, ultimately we also want to deal with syntax in certain theories, specifically, in theories formulated in the language of arithmetic. In these cases it is necessary to *arithmetize* syntax, i.e., to represent syntactic objects, operations, and relations as numbers, arithmetical functions, and arithmetical relations, respectively. This is done by assigning numbers to symbols as their "codes." Since we can deal with sequences of numbers purely arithmetically by the powers-of-primes coding, we can extend this coding of individual symbols to coding of sequences of symbols (such as terms and formulas) and also arrangements of such sequences (such as derivations). This extended coding is called "Gödel numbering." Because the sequences of interest (terms, formulas, derivations) are inductively defined, and the operations and relations on them are computable, the corresponding sets, operations, and relations are in fact all computable, and almost all of them are in fact primitive recursive.

11.2 Coding Symbols

The basic language $\mathcal{L}$ of first order logic makes use of the symbols

$$\neg, \vee, \wedge, \rightarrow, \forall, \exists, =, (,)$$

together with enumerable sets of variables and constant symbols, and enumerable sets of function symbols and predicate symbols of arbitrary arity. We can assign *codes* to each of these symbols in such a way that every symbol is assigned a unique number as its code, and no two different symbols are assigned the same number. We know that this is possible since the set of all symbols is enumerable and so there is a bijection between it and the set of natural numbers. But we want to make sure that we can recover the symbol (as well as some information about it, e.g., the arity of a function symbol) from its code in a computable way. There are many possible ways of doing this, of course. Here is one such way, which uses primitive recursive functions. (Recall that $\langle n_0, \ldots, n_k \rangle$ is the number coding the sequence of numbers $n_0, \ldots, n_k$.)

Definition 11.1. If s is a symbol of $\mathcal{L}$, let the *symbol code* $\mathrm{c}(s)$ be defined as follows:

1. If s is among the logical symbols, $\mathrm{c}(s)$ is given by the following table:

$\neg$	$\vee$	$\wedge$	$\rightarrow$	$\forall$	$\exists$	$=$	$($	$)$	,
$\langle 0,0 \rangle$	$\langle 0,1 \rangle$	$\langle 0,2 \rangle$	$\langle 0,3 \rangle$	$\langle 0,4 \rangle$	$\langle 0,5 \rangle$	$\langle 0,6 \rangle$	$\langle 0,7 \rangle$	$\langle 0,8 \rangle$	$\langle 0,9 \rangle$

2. If s is the i-th variable x_i, then $\mathrm{c}(s) = \langle 1, i \rangle$.
3. If s is the i-th constant symbol c_i^n, then $\mathrm{c}(s) = \langle 2, i \rangle$.
4. If s is the i-th n-ary function symbol f_i^n, then $\mathrm{c}(s) = \langle 3, n, i \rangle$.
5. If s is the i-th n-ary predicate symbol P_i^n, then $\mathrm{c}(s) = \langle 4, n, i \rangle$.

Proposition 11.2. *The following relations are primitive recursive:*

1. $\mathrm{Fn}(x, n)$ *iff* x *is the code of* f_i^n *for some* i*, i.e.,* x *is the code of an* n*-ary function symbol.*
2. $\mathrm{Pred}(x, n)$ *iff* x *is the code of* P_i^n *for some* i *or* x *is the code of* $=$ *and* $n = 2$*, i.e.,* x *is the code of an* n*-ary predicate symbol.*

Definition 11.3. If $s_0, \ldots, s_n$ is a sequence of symbols, its *Gödel number* is $\langle \mathrm{c}(s_0), \ldots, \mathrm{c}(s_n) \rangle$.

11.3 Coding Terms

A term is simply a certain kind of sequence of symbols: it is built up inductively from constants and variables according to the formation rules for terms. Since sequences of symbols can be coded as numbers—using a coding scheme for the symbols plus a way to code sequences of numbers—assigning Gödel numbers to terms is not difficult. The challenge is rather to show that the property a number has if it is the Gödel number of a correctly formed term is computable, or in fact primitive recursive.

Proposition 11.4. *The relation* $\mathrm{Term}(x)$ *which holds iff* x *is the Gödel number of a term, is primitive recursive.*

Proof. A sequence of symbols s is a term iff there is a sequence $s_0, \ldots, s_{k-1} = s$ of terms which records how the term s was formed from constant symbols and variables according to the formation rules for terms. To express that such a putative formation sequence follows the formation rules it has to be the case that, for each $i < k$, either

1. s_i is a variable v_j, or
2. s_i is a constant symbol c_j, or
3. s_i is built from n terms $t_1, \ldots, t_n$ occurring prior to place i using an n-place function symbol f^n_j.

To show that the corresponding relation on Gödel numbers is primitive recursive, we have to express this condition primitive recursively, i.e., using primitive recursive functions, relations, and bounded quantification.

Suppose y is the number that codes the sequence $s_0, \ldots, s_{k-1}$, i.e., $y = \langle \#(s_0), \ldots, \#(s_k) \rangle$. It codes a formation sequence for the term with Gödel number x iff for all $i < k$:

1. there is a j such that $(y)_i = \#(v_j)$, or
2. there is a j such that $(y)_i = \#(c_j)$, or
3. there is an n and a number $z = \langle z_1, \ldots, z_n \rangle$ such that each z_l is equal to some $(y)_{i'}$ for $i' < i$ and

$$(y)_i = \#(f^n_j() \frown \mathrm{flatten}(z) \frown \#()),$$

and moreover $(y)_{k-1} = x$. The function $\mathrm{flatten}(z)$ turns the sequence $\langle \#(t_1), \ldots, \#(t_n) \rangle$ into $\#(t_1, \ldots, t_n)$ and is primitive recursive.

The indices j, n, the Gödel numbers z_l of the terms t_l, and the code z of the sequence $\langle z_1, \ldots, z_n \rangle$, in (3) are all less than y. We can replace k above with $\mathrm{len}(y)$. Hence we can express "y is the code of a formation sequence of the term with Gödel number x" in a way that shows that this relation is primitive recursive.

We now just have to convince ourselves that there is a primitive recursive bound on y. But if x is the Gödel number of a term, it must have a formation sequence with at most $\mathrm{len}(x)$ terms (since every term in the formation sequence of s must start at some place in s, and no two subterms can start at the same place). The Gödel number of each subterm of s is of course $\leq x$. Hence, there always is a formation sequence with code $\leq x^{\mathrm{len}(x)}$. □

11.4 Coding Formulas

Proposition 11.5. *The relation* $\mathrm{Atom}(x)$ *which holds iff* x *is the Gödel number of an atomic formula, is primitive recursive.*

Proof. The number x is the Gödel number of an atomic formula iff one of the following holds:

1. There are $n, j < x$, and $z < x$ such that for each $i < n$, $\mathrm{Term}((z)_i)$ and
$$x = \#(P^n_j() \frown \mathrm{flatten}(z) \frown \#()).$$
2. There are $z_1, z_2 < x$ such that $\mathrm{Term}(z_1)$, $\mathrm{Term}(z_2)$, and
$$x = z_1 \frown \#(=) \frown z_2.$$
3. $x = \#(\bot)$.
4. $x = \#(\top)$.

□

Proposition 11.6. *The relation* $\mathrm{Frm}(x)$ *which holds iff* x *is the Gödel number of a formula is primitive recursive.*

Proof. A sequence of symbols s is a formula iff there is formation sequence s_0, $\ldots$, $s_{k-1} = s$ of formula which records how s was formed from atomic formulas according to the formation rules. The code for each s_i (and indeed of the code of the sequence $\langle s_0, \ldots, s_{k-1} \rangle$ is less than the code x of s. □

11.5 Substitution

Proposition 11.7. *There is a primitive recursive function* $\mathrm{Subst}(x, y, z)$ *with the property that*
$$\mathrm{Subst}(\#(\varphi), \#(t), \#(x)) = \#(\varphi[t/x])$$

Proof. Let us suppose that the predicate $\mathrm{FreeOcc}(x, z, i)$, which holds if the i-th symbols of the formula with Gödel number x is a free occurrence of the variable with Gödel number z, is primitive recursive. We can then define a function Subst' by primitive recursion as follows:

$$\mathrm{Subst}'(0, x, y, z) = \emptyset$$
$$\mathrm{Subst}'(i+1, x, y, z) = \begin{cases} \mathrm{Subst}'(i, x, y, z) \frown y & \text{if } \mathrm{FreeOcc}(x, z, i+1) \\ \mathrm{append}(\mathrm{Subst}'(i, x, y, z), (x)_{i+1}) & \text{otherwise.} \end{cases}$$

$\mathrm{Subst}(x, y, z)$ can now be defined as $\mathrm{Subst}'(\mathrm{len}(x), x, y, z)$. □

11.6 Proofs in LK

Definition 11.8. If Γ is a finite set of sentences, $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$, then $\#(\Gamma) = \langle \#(\varphi_1), \ldots, \#(\varphi_n) \rangle$.

If $\Gamma \Rightarrow \Delta$ is a sequent, then a Gödel number of $\Gamma \Rightarrow \Delta$ is

$$\#(\Gamma \Rightarrow \Delta) = \langle \#(\Gamma), \#(\Delta) \rangle$$

If Π is a derivation in **LK**, then $\#(\Pi)$ is

1. $\langle 0, \#(\Gamma \Rightarrow \Delta) \rangle$ if Π consists only of the initial sequent $\Gamma \Rightarrow \Delta$.

2. $\langle 1, \#(\Gamma \Rightarrow \Delta), k, \#(\Pi') \rangle$ if Π ends in an inference with one premise, k is given by the following table according to which rule was used in the last inference, and Π' is the immediate subproof ending in the premise of the last inference.

Rule:	Contr	$\neg$ left	$\neg$ right	$\wedge$ left	$\vee$ right	$\rightarrow$ right
k:	1	2	3	4	5	6

Rule:	$\forall$ left	$\forall$ right	$\exists$ left	$\exists$ right	=
k:	7	8	9	10	11

3. $\langle 2, \#(\Gamma \Rightarrow \Delta), k, \#(\Pi'), \#(\Pi'') \rangle$ if Π ends in an inference with two premises, k is given by the following table according to which rule was used in the last inference, and Π', Π'' are the immediate subproof ending in the left and right premise of the last inference, respectively.

Rule:	Cut	$\wedge$ right	$\vee$ left	$\rightarrow$ left
k:	1	2	3	4

Proposition 11.9. *The following relations are primitive recursive:*

1. $\varphi \in \Gamma$.
2. $\Gamma \subseteq \Delta$.
3. *$\Gamma \Rightarrow \Delta$ is an initial sequent.*
4. *$\Gamma \Rightarrow \Delta$ follows from $\Gamma' \Rightarrow \Delta'$ (and $\Gamma'' \Rightarrow \Delta''$) by a rule of* **LK**.
5. *Π is a correct* **LK***-derivation.*

Proof. We have to show that the corresponding relations between Gödel numbers of formulas, sequences of Gödel numbers of formulas (which code sets of formulas), and Gödel numbers of sequents, are primitive recursive.

1. $\varphi \in \Gamma$ iff $\#(\varphi)$ occurs in the sequence $\#(\Gamma)$, i.e, $\mathrm{IsIn}(x, g) \Leftrightarrow \exists i < \mathrm{len}(g)\, (g)_i = x$. We'll abbreviate this as $x \in g$.

2. $\Gamma \subseteq \Delta$ iff every element of $\#(\Gamma)$ is also an an element of $\#(\Delta)$, i.e., $\mathrm{SubSet}(g, d) \Leftrightarrow \forall i < \mathrm{len}(g)\, (g)_i \in d$. We'll abbreviate this as $g \subseteq d$.

3. $\Gamma \Rightarrow \Delta$ is an initial sequent if either there is a sentence φ such that $\Gamma \Rightarrow \Delta$ is $\varphi \Rightarrow \varphi$, or there is a term t such that $\Gamma \Rightarrow \Delta$ is $\emptyset \Rightarrow t = t$. In terms of Gödel numbers,

$$\begin{aligned}\mathrm{InitSeq}(s) \Leftrightarrow &\exists x < s\, (\mathrm{Sent}(x) \wedge s = \langle\langle x\rangle, \langle x\rangle\rangle) \vee \\ &\exists t < s\, (\mathrm{Term}(t) \wedge s = \langle 0, t \frown \#(=) \frown t\rangle).\end{aligned}$$

4. Here we have to show that for each rule of inference R the relation $\mathrm{FollowsBy}_R(s, s')$ which holds if s and s' are the Gödel numbers of conclusion and premise of a correct application of R is primitive recursive. If R has two premises, $\mathrm{FollowsBy}_R$ of course has three arguments.

   For instance, $\Gamma \Rightarrow \Delta$ follows correctly from $\Gamma' \Rightarrow \Delta'$ by $\exists$right iff $\Gamma = \Gamma'$ and there is a formula φ, a variable x and a closed term t such that $\varphi[t/x] \in \Delta'$ and $\exists x\, \varphi \in \Delta$, for every $\psi \in \Delta$, either $\psi = \exists x\, \varphi$ or $\psi \in \Delta'$, and for every $\psi \in \Delta'$, $\psi = \varphi[t/x]$ or $\psi \in \Delta$. We just have to translate this into Gödel numbers. If $s = \#(\Gamma \Rightarrow \Delta)$ then $(s)_0 = \#(\Gamma)$ and $(s)_1 = \#(\Delta)$. So:

$$\begin{aligned}\mathrm{FollowsBy}_{\exists\mathrm{right}}(s, s') \Leftrightarrow\ & (s)_0 \subseteq (s')_0 \wedge (s')_0 \subseteq (s)_0 \wedge \\ & \exists f < s\, \exists x < s\, \exists t < s'\, (\mathrm{Frm}(f) \wedge \mathrm{Var}(x) \wedge \mathrm{Term}(t) \wedge \\ & \mathrm{Subst}(f, t, x) \in (s')_1 \wedge \#(\exists) \frown x \frown f \in (s)_1 \wedge \\ & \forall i < \mathrm{len}((s)_1)\, (((s)_1)_i = \#(\exists) \frown x \frown f \vee ((s)_1)_i \in (s')_1) \wedge \\ & \forall i < \mathrm{len}((s')_1)\, (((s)'_1)_i = \mathrm{Subst}(f, t, x) \vee ((s')_1)_i \in (s)_1))\end{aligned}$$

   The individual lines express, respectively, "$\Gamma \subseteq \Gamma' \wedge \Gamma' \subseteq \Gamma$," "there is a formula with Gödel number f, a variable with Gödel number x, and a term with Gödel number t," "$\varphi[t/x] \in \Delta' \wedge \exists x\, \varphi \in \Delta$," "for all $\psi \in \Delta$, either $\psi = \exists x\, \varphi$ or $\psi \in \Delta'$," "for all $\psi \in \Delta'$, either $\psi = \varphi[t/x]$ or $\psi \in \Delta$. Note that in the last two lines, we quantify over the elements ψ of Δ and Δ' not directly, but via their place i in the Gödel numbers of Δ and Δ'. (Remember that $\#(\Delta)$ is the number of a sequence of Gödel numbers of formulas in Δ.)

5. We first define a helper relation $\mathrm{hDeriv}(s, n)$ which holds if s codes a correct derivation at least to n inferences up from the end sequent. If $n = 0$ we let the relation be satisfied by default. Otherwise, $\mathrm{hDeriv}(s, n+1)$ iff either s consists just of an initial sequent, or it ends in a correct inference

and the codes of the immediate subderivations satisfy $\text{nDeriv}(s, n)$.

$$
\begin{aligned}
\text{nDeriv}(s, 0) &\Leftrightarrow 1 \\
\text{nDeriv}(s, n+1) &\Leftrightarrow ((s)_0 = 0 \wedge \text{InitialSeq}((s)_1)) \vee \\
&\quad ((s)_0 = 1 \wedge \\
&\quad\quad ((s)_2 = 1 \wedge \text{FollowsBy}_{\text{Contr}}((s)_1, ((s)_3)_1)) \vee \\
&\quad\quad\quad \vdots \\
&\quad\quad ((s)_2 = 11 \wedge \text{FollowsBy}_{=}((s)_1, ((s)_3)_1)) \wedge \\
&\quad\quad \text{nDeriv}((s)_3, n)) \vee \\
&\quad ((s)_0 = 2 \wedge \\
&\quad\quad ((s)_2 = 1 \wedge \text{FollowsBy}_{\text{Cut}}((s)_1, ((s)_3)_1), ((s)_4)_1)) \vee \\
&\quad\quad\quad \vdots \\
&\quad\quad ((s)_2 = 4 \wedge \text{FollowsBy}_{\to\text{left}}((s)_1, ((s)_3)_1), ((s)_4)_1)) \wedge \\
&\quad\quad \text{nDeriv}((s)_3, n) \wedge \text{nDeriv}((s)_4, n))
\end{aligned}
$$

This is a primitive recursive definition. If the number n is large enough, e.g., larger than the maximum number of inferences between an initial sequent and the end sequent in s, it holds of s iff s is the Gödel number of a correct derivation. The number s itself is larger than that maximum number of inferences. So we can now define $\text{Deriv}(s)$ by $\text{nDeriv}(s, s)$.

□

Proposition 11.10. *Suppose Γ is a primitive recursive set of sentences. Then the relation $\text{Pr}_\Gamma(x, y)$ expressing "x is the code of a derivation Π of $\Gamma_0 \Rightarrow \varphi$ for some finite $\Gamma_0 \subseteq \Gamma$ and x is the Gödel number of φ" is primitive recursive.*

Proof. Suppose "$y \in \Gamma$" is given by the primitive recursive predicate $R_\Gamma(y)$. We have to show that $\text{Pr}_\Gamma(x, y)$ which holds iff y is the Gödel number of a sentence φ and x is the code of an **LK**-derivation with end sequent $\Gamma_0 \Rightarrow \varphi$ is primitive recursive.

By the previous proposition, the property $\text{Deriv}()$ which holds iff x is the code of a correct derivation Π in **LK** is primitive recursive. If x is such a code, then $(x)_1$ is the code of the end sequent of Π, and so $((x)_1)_0$ is the code of the left side of the end sequent and $((x)_1)_1$ the right side. So we can express "the right side of the end sequent of Π is φ" as $\text{len}(((x)_1)_1) = 1 \wedge (((x)_1)_1)_0 = x$. The left side of the end sequent of Π is of course automatically finite, we just have to express that every sentence in it is in Γ. Thus we can define $\text{Pr}_\Gamma(x, y)$

by

$$\begin{aligned}\mathrm{Pr}_\Gamma(x,y) \Leftrightarrow\ & \mathrm{Sent}(y) \wedge \mathrm{Deriv}(x) \wedge \\ & \forall i < \mathrm{len}(((x)_1)_0)\ (((x)_1)_0)_i \in \Gamma \wedge \\ & \mathrm{len}(((x)_1)_1) = 1 \wedge (((x)_1)_1)_0 = x\end{aligned}$$

□

Chapter 12

Representability in Q

12.1 Introduction

We will describe a very minimal such theory called "**Q**" (or, sometimes, "Robinson's Q," after Raphael Robinson). We will say what it means for a function to be *representable* in **Q**, and then we will prove the following:

> A function is representable in **Q** if and only if it is computable.

For one thing, this provides us with another model of computability. But we will also use it to show that the set $\{\varphi : \mathbf{Q} \vdash \varphi\}$ is not decidable, by reducing the halting problem to it. By the time we are done, we will have proved much stronger things than this.

The language of **Q** is the language of arithmetic; **Q** consists of the following axioms (to be used in conjunction with the other axioms and rules of first-order logic with identity predicate):

1. $\forall x \, \forall y \, x' = y' \rightarrow x = y$
2. $\forall x \, 0 \neq x'$
3. $\forall x \, x \neq 0 \rightarrow \exists y \, x = y'$
4. $\forall x \, (x + 0) = x$
5. $\forall x \, \forall y \, (x + y') = (x + y)'$
6. $\forall x \, (x \times 0) = 0$
7. $\forall x \, \forall y \, (x \times y') = ((x \times y) + x)$
8. $\forall x \, \forall y \, x < y \leftrightarrow \exists z \, (z' + x) = y$

For each natural number n, define the numeral $\bar{n}$ to be the term $0^{\prime\cdots\prime}$ where there are n tick marks in all.

As a theory of arithmetic, Q is *extremely* weak; for example, you can't even prove very simple facts like $\forall x\, x \neq x'$ or $\forall x \forall y\, (x + y) = (y + x)$. But we will see that much of the reason that Q is so interesting is *because* it is so weak, in fact, just barely strong enough for the incompleteness theorem to hold; and also because it has a *finite* set of axioms.

A stronger theory than **Q** called *Peano arithmetic* **PA**, is obtained by adding a schema of induction to **Q**:

$$(\varphi(0) \wedge \forall x\, (\varphi(x) \to \varphi(x'))) \to \forall x\, \varphi(x)$$

where $\varphi(x)$ is any formula, possibly with free variables other than x. Using induction, one can do much better; in fact, it takes a good deal of work to find "natural" statements about the natural numbers that can't be proved in Peano arithmetic!

Definition 12.1. A function $f(x_0, \ldots, x_k)$ from the natural numbers to the natural numbers is said to be *representable in* **Q** if there is a formula $\varphi_f(x_0, \ldots, x_k, y)$ such that whenever $f(n_0, \ldots, n_k) = m$, **Q** proves

1. $\varphi_f(\bar{n}_0, \ldots, \bar{n}_k, \bar{m})$
2. $\forall y\, (\varphi_f(\bar{n}_0, \ldots, \bar{n}_k, y) \to \bar{m} = y)$.

There are other ways of stating the definition; for example, we could equivalently require that **Q** proves $\forall y\, (\varphi_f(\bar{n}_0, \ldots, \bar{n}_k, y) \leftrightarrow \bar{m} = y)$.

Theorem 12.2. *A function is representable in* **Q** *if and only if it is computable.*

There are two directions to proving the theorem. One of them is fairly straightforward once arithmetization of syntax is in place. The other direction requires more work.

12.2 Functions Representable in Q are Computable

Lemma 12.3. *Every function that is representable in* **Q** *is computable.*

Proof. All we need to know is that we can code terms, formulas, and proofs in such a way that the relation "d is a proof of φ in the theory **Q**" is computable, as well as the function SubNumeral(φ, n, v) which returns (a numerical code of) the result of substituting the numeral corresponding to n for the variable (coded by) v in the formula (coded by) φ. Assuming this, suppose the function f is represented by $\varphi_f(x_0, \ldots, x_k, y)$. Then the algorithm for computing f is as follows: on input $n_0, \ldots, n_k$, search for a number m and a proof of the formula $\varphi_f(\bar{n}_0, \ldots, \bar{n}_k, \bar{m})$; when you find one, output m. In other words,

$$f(n_0, \ldots, n_k) = (\mu s(\text{"}(s)_0 \text{ is a proof of } \varphi(\bar{n}_0, \ldots, \bar{n}_k, \overline{(s)_1}) \text{ in } \mathbf{Q}\text{"}))_1.$$

This completes the proof, modulo the (involved but routine) details of coding and defining the function and relation above. □

12.3 Computable Functions are Representable in Q

Lemma 12.4. *Every computable function is representable in* **Q**.

1. We will define a set of (total) functions, C.
2. We will show that C is the set of computable functions, i.e. our definition provides another characterization of computability.
3. Then we will show that every function in C can be represented in **Q**.

12.4 The Functions C

Let C be the smallest set of functions containing

1. 0,
2. successor,
3. addition,
4. multiplication,
5. projections, and
6. the characteristic function for equality, $\chi_=$;

and closed under

1. composition, and
2. unbounded search, applied to regular functions.

Remember this last restriction means simply that you can only use the μ operation when the result is total. Compare this to the definition of the *general recursive* functions: here we have added plus, times, and $\chi_=$, but we have dropped primitive recursion.

Clearly everything in C is recursive, since plus, times, and $\chi_=$ are. We will show that the converse is also true; this amounts to saying that with the other stuff in C we can carry out primitive recursion.

12.5 The Beta Function Lemma

In order to show that C can carry out primitive recursion, we need to develop functions that handle sequences. (If we had exponentiation as well, our task would be easier.) When we had primitive recursion, we could define things like the "nth prime," and pick a fairly straightforward coding. But here we do not have primitive recursion, so we need to be more clever.

Lemma 12.5. *There is a function $\beta(d, i)$ in C such that for every sequence $a_0, \ldots, a_n$ there is a number d, such that for every i less than or equal to n, $\beta(d, i) = a_i$.*

Think of d as coding the sequence $\langle a_0, \ldots, a_n \rangle$, and $\beta(d, i)$ returning the ith element. The lemma is fairly minimal; it doesn't say we can concatenate sequences or append elements with functions in C, or even that we can *compute* d from $a_0, \ldots, a_n$ using functions in C. All it says is that there is a "decoding" function such that every sequence is "coded."

The use of the notation β is Gödel's. To repeat, the hard part of proving the lemma is defining a suitable β using the seemingly restricted resources in the definition of C. There are various ways to prove this lemma, but one of the cleanest is still Gödel's original method, which used a number-theoretic fact called the Chinese Remainder theorem.

Definition 12.6. Two natural numbers a and b are *relatively prime* if their greatest common divisor is 1; in other words, they have no other divisors in common.

Definition 12.7. $a \equiv b \mod c$ means $c \mid (a - b)$, i.e. a and b have the same remainder when divided by c.

Here is the *Chinese Remainder theorem*:

Theorem 12.8. *Suppose $x_0, \ldots, x_n$ are (pairwise) relatively prime. Let $y_0, \ldots, y_n$ be any numbers. Then there is a number z such that*

$$\begin{aligned} z &\equiv y_0 \mod x_0 \\ z &\equiv y_1 \mod x_1 \\ &\vdots \\ z &\equiv y_n \mod x_n. \end{aligned}$$

Here is how we will use the Chinese Remainder theorem: if $x_0, \ldots, x_n$ are bigger than $y_0, \ldots, y_n$ respectively, then we can take z to code the sequence $\langle y_0, \ldots, y_n \rangle$. To recover y_i, we need only divide z by x_i and take the remainder. To use this coding, we will need to find suitable values for $x_0, \ldots, x_n$.

A couple of observations will help us in this regard. Given $y_0, \ldots, y_n$, let

$$j = \max(n, y_0, \ldots, y_n) + 1,$$

and let

$$\begin{aligned} x_0 &= 1 + j! \\ x_1 &= 1 + 2 \cdot j! \\ x_2 &= 1 + 3 \cdot j! \\ &\vdots \\ x_n &= 1 + (n+1) \cdot j! \end{aligned}$$

Then two things are true:

1. $x_0, \ldots, x_n$ are relatively prime.
2. For each i, $y_i < x_i$.

To see that clause 1 is true, note that if p is a prime number and $p \mid x_i$ and $p \mid x_k$, then $p \mid 1 + (i+1)j!$ and $p \mid 1 + (k+1)j!$. But then p divides their difference,

$$(1 + (i+1)j!) - (1 + (k+1)j!) = (i-k)j!.$$

Since p divides $1 + (1+1)j!$, it can't divide $j!$ as well (otherwise, the first division would leave a remainder of 1). So p divides $i - k$. But $|i - k|$ is at most n, and we have chosen $j > n$, so this implies that $p \mid j!$, again a contradiction. So there is no prime number dividing both x_i and x_k. Clause 2 is easy: we have $y_i < j < j! < x_i$.

Now let us prove the β function lemma. Remember that C is the smallest set containing 0, successor, plus, times, $\chi_=$, projections, and closed under composition and μ applied to regular functions. As usual, say a relation is in C if its characteristic function is. As before we can show that the relations in C are closed under boolean combinations and bounded quantification; for example:

1. $\text{not}(x) = \chi_=(x, 0)$
2. $\mu x \leq z\ R(x, y) = \mu x\ (R(x, y) \vee x = z)$
3. $\exists x \leq z\, R(x, y) \Leftrightarrow R(\mu x \leq z\ R(x, y), y)$

We can then show that all of the following are in C:

1. The pairing function, $J(x, y) = \frac{1}{2}[(x + y)(x + y + 1)] + x$
2. Projections
   $$K(z) = \mu x \leq q\ (\exists y \leq z\, [z = J(x, y)])$$
   and
   $$L(z) = \mu y \leq q\ (\exists x \leq z\, [z = J(x, y)]).$$
3. $x < y$

4. $x \mid y$

5. The function $\text{rem}(x, y)$ which returns the remainder when y is divided by x

Now define

$$\beta^*(d_0, d_1, i) = \text{rem}(1 + (i+1)d_1, d_0)$$

and

$$\beta(d, i) = \beta^*(K(d), L(d), i).$$

This is the function we need. Given $a_0, \ldots, a_n$, as above, let

$$j = \max(n, a_0, \ldots, a_n) + 1,$$

and let $d_1 = j!$. By the observations above, we know that $1 + d_1, 1 + 2d_1, \ldots, 1 + (n+1)d_1$ are relatively prime and all are bigger than $a_0, \ldots, a_n$. By the Chinese Remainder theorem there is a value d_0 such that for each i,

$$d_0 \equiv a_i \mod (1 + (i+1)d_1)$$

and so (because d_1 is greater than a_i),

$$a_i = \text{rem}(1 + (i+1)d_1, d_0).$$

Let $d = J(d_0, d_1)$. Then for each i from 0 to n, we have

$$\begin{aligned} \beta(d, i) &= \beta^*(d_0, d_1, i) \\ &= \text{rem}(1 + (i+1)d_1, d_0) \\ &= a_i \end{aligned}$$

which is what we need. This completes the proof of the β-function lemma.

12.6 Primitive Recursion in C

Now we can show that C is closed under primitive recursion. Suppose $f(\vec{z})$ and $g(u, v, \vec{z})$ are both in C. Let $h(x, \vec{z})$ be the function defined by

$$\begin{aligned} h(0, \vec{z}) &= f(\vec{z}) \\ h(x+1, \vec{z}) &= g(x, h(x, \vec{z}), \vec{z}). \end{aligned}$$

We need to show that h is in C.

First, define an auxiliary function $\hat{h}(x, \vec{z})$ which returns the least number d such that d codes a sequence satisfying

1. $(d)_0 = f(\vec{z})$, and

2. for each $i < x$, $(d)_{i+1} = g(i, (d)_i, \vec{z})$,

where now $(d)_i$ is short for $\beta(d, i)$. In other words, $\hat{h}$ returns a sequence that begins $\langle h(0, \vec{z}), h(1, \vec{z}), \ldots, h(x, \vec{z})\rangle$. $\hat{h}$ is in C, because we can write it as

$$\hat{h}(x, z) = \mu d\ (\beta(d, 0) = f(\vec{z}) \wedge \forall i < x\, \beta(d, i+1) = g(i, \beta(d, i), \vec{z})).$$

But then we have

$$h(x, \vec{z}) = \beta(\hat{h}(x, \vec{z}), x),$$

so h is in C as well.

12.7 Functions in C are Representable in Q

We have to show that every function in C is representable in **Q**. In the end, we need to show how to assign to each k-ary function $f(x_0, \ldots, x_{k-1})$ in C a formula $\varphi_f(x_0, \ldots, x_{k-1}, y)$ that represents it.

Lemma 12.9. *Given natural numbers n and m, if $n \neq m$, then $Q \vdash \overline{n} \neq \overline{m}$.*

Proof. Use induction on n to show that for every m, if $n \neq m$, then $Q \vdash \overline{n} \neq \overline{m}$.

In the base case, $n = 0$. If m is not equal to 0, then $m = k + 1$ for some natural number k. We have an axiom that says $\forall x\, 0 \neq x'$. By a quantifier axiom, replacing x by $\overline{k}$, we can conclude $0 \neq \overline{k}'$. But $\overline{k}'$ is just $\overline{m}$.

In the induction step, we can assume the claim is true for n, and consider $n + 1$. Let m be any natural number. There are two possibilities: either $m = 0$ or for some k we have $m = k + 1$. The first case is handled as above. In the second case, suppose $n + 1 \neq k + 1$. Then $n \neq k$. By the induction hypothesis for n we have $\mathbf{Q} \vdash \overline{n} \neq \overline{k}$. We have an axiom that says $\forall x\, \forall y\, x' = y' \rightarrow x = y$. Using a quantifier axiom, we have $\overline{n}' = \overline{k}' \rightarrow \overline{n} = \overline{k}$. Using propositional logic, we can conclude, in **Q**, $\overline{n} \neq \overline{k} \rightarrow \overline{n}' \neq \overline{k}'$. Using modus ponens, we can conclude $\overline{n}' \neq \overline{k}'$, which is what we want, since $\overline{k}'$ is $\overline{m}$. □

Note that the lemma does not say much: in essence it says that **Q** can prove that different numerals denote different objects. For example, **Q** proves $0'' \neq 0'''$. But showing that this holds in general requires some care. Note also that although we are using induction, it is induction *outside* of **Q**.

We will be able to represent zero, successor, plus, times, the characteristic function for equality, and projections. In each case, the appropriate representing function is entirely straightforward; for example, zero is represented by the formula

$$y = 0,$$

successor is represented by the formula

$$x_0' = y,$$

and plus is represented by the formula

$$x_0 + x_1 = y.$$

The work involves showing that **Q** can prove the relevant statements; for example, saying that plus is represented by the formula above involves showing that for every pair of natural numbers m and n, **Q** proves

$$\overline{n} + \overline{m} = \overline{n+m}$$

and

$$\forall y\, ((\overline{n} + \overline{m}) = y \to y = \overline{n+m}).$$

What about composition? Suppose h is defined by

$$h(x_0, \ldots, x_{l-1}) = f(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{k-1}(x_0, \ldots, x_{l-1})).$$

where we have already found formulas $\varphi_f, \varphi_{g_0}, \ldots, \varphi_{g_{k-1}}$ representing the functions $f, g_0, \ldots, g_{k-1}$, respectively. Then we can define a formula φ_h representing h, by defining $\varphi_h(x_0, \ldots, x_{l-1}, y)$ to be

$$\exists z_0, \ldots\, \exists z_{k-1}\, (\varphi_{g_0}(x_0, \ldots, x_{l-1}, z_0) \wedge \cdots \wedge \varphi_{g_{k-1}}(x_0, \ldots, x_{l-1}, z_{k-1}) \wedge \varphi_f(z_0, \ldots, z_{k-1}, y)).$$

Finally, let us consider unbounded search. Suppose $g(x, \vec{z})$ is regular and representable in **Q**, say by the formula $\varphi_g(x, \vec{z}, y)$. Let f be defined by $f(\vec{z}) = \mu x\ g(x, \vec{z})$. We would like to find a formula $\varphi_f(\vec{z}, y)$ representing f. Here is a natural choice:

$$\varphi_f(\vec{z}, y) \equiv \varphi_g(y, \vec{z}, 0) \wedge \forall w\, (w < y \to \neg \varphi_g(w, \vec{z}, 0)).$$

It can be shown that this works using some additional lemmas, e.g.,

Lemma 12.10. *For every variable x and every natural number n, **Q** proves $(x' + \overline{n}) = (x + \overline{n})'$.*

It is again worth mentioning that this is weaker than saying that **Q** proves $\forall x\, \forall y\, (x' + y) = (x + y)'$ (which is false).

Proof. The proof is, as usual, by induction on n. In the base case, $n = 0$, we need to show that **Q** proves $(x' + 0) = (x + 0)'$. But we have:

$(x' + 0) = x'$ from axiom 4
$(x + 0) = x$ from axiom 4
$(x + 0)' = x'$ by line 2
$(x' + 0) = (x + 0)'$ lines 1 and 3

In the induction step, we can assume that we have derived $(x' + \overline{n}) = (x + \overline{n})'$ in **Q**. Since $\overline{n+1}$ is $\overline{n}'$, we need to show that **Q** proves $(x' + \overline{n}') = (x + \overline{n}')'$. The following chain of equalities can be derived in **Q**:

$$\begin{aligned}(x' + \overline{n}') &= (x' + \overline{n})' && \text{axiom 5}\\ &= (x + \overline{n}')' && \text{from the inductive hypothesis}\end{aligned}$$

□

Lemma 12.11. *1.* **Q** *proves* $\neg(x < \overline{0})$.

2. For every natural number n, **Q** *proves*

$$x < \overline{n+1} \to (x = \overline{0} \vee \cdots \vee x = \overline{n}).$$

Proof. Let us do 1 and part of 2, informally (i.e., only giving hints as to how to construct the formal derivation).

For part 1, by the definition of $<$, we need to prove $\neg\exists y\,(y' + x) = 0$ in **Q**, which is equivalent (using the axioms and rules of first-order logic) to $\forall y\,(y' + x) \neq 0$. Here is the idea: suppose $(y' + x) = 0$. If x is 0, we have $(y' + 0) = 0$. But by axiom 4 of **Q**, we have $(y' + 0) = y'$, and by axiom 2 we have $y' \neq 0$, a contradiction. So $\forall y\,(y' + x) \neq 0$. If x is not 0, by axiom 3 there is a z such that $x = z'$. But then we have $(y' + z') = 0$. By axiom 5, we have $(y' + z)' = 0$, again contradicting axiom 2.

For part 2, use induction on n. Let us consider the base case, when $n = 0$. In that case, we need to show $x < \overline{1} \to x = \overline{0}$. Suppose $x < \overline{1}$. Then by the defining axiom for $<$, we have $\exists y\,(y' + x) = 0'$. Suppose y has that property; so we have $y' + x = 0'$.

We need to show $x = 0$. By axiom 3, if x is not 0, it is equal to z' for some z. Then we have $(y' + z') = 0'$. By axiom 5 of **Q**, we have $(y' + z)' = 0'$. By axiom 1, we have $(y' + z) = 0$. But this means, by definition, $z < 0$, contradicting part 1. □

We have shown that the set of computable functions can be characterized as the set of functions representable in **Q**. In fact, the proof is more general. From the definition of representability, it is not hard to see that any theory extending **Q** (or in which one can interpret **Q**) can represent the computable functions; but, conversely, in any proof system in which the notion of proof is computable, every representable function is computable. So, for example, the set of computable functions can be characterized as the set of functions represented in Peano arithmetic, or even Zermelo Fraenkel set theory. As Gödel noted, this is somewhat surprising. We will see that when it comes to provability, questions are very sensitive to which theory you consider; roughly, the stronger the axioms, the more you can prove. But across a wide range of axiomatic theories, the representable functions are exactly the computable ones.

12.8 Representing Relations

Let us say what it means for a *relation* to be representable.

Definition 12.12. A relation $R(x_0, \ldots, x_k)$ on the natural numbers is *representable in* **Q** if there is a formula $\varphi_R(x_0, \ldots, x_k)$ such that whenever $R(n_0, \ldots, n_k)$ is true, **Q** proves $\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$, and whenever $R(n_0, \ldots, n_k)$ is false, **Q** proves $\neg\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$.

Theorem 12.13. *A relation is representable in* **Q** *if and only if it is computable.*

Proof. For the forwards direction, suppose $R(x_0, \ldots, x_k)$ is represented by the formula $\varphi_R(x_0, \ldots, x_k)$. Here is an algorithm for computing R: on input $n_0, \ldots, n_k$, simultaneously search for a proof of $\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$ and a proof of $\neg\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$. By our hypothesis, the search is bound to find one of the other; if it is the first, report "yes," and otherwise, report "no."

In the other direction, suppose $R(x_0, \ldots, x_k)$ is computable. By definition, this means that the function $\chi_R(x_0, \ldots, x_k)$ is computable. By Theorem 12.2, χ_R is represented by a formula, say $\varphi_{\chi_R}(x_0, \ldots, x_k, y)$. Let $\varphi_R(x_0, \ldots, x_k)$ be the formula $\varphi_{\chi_R}(x_0, \ldots, x_k, \overline{1})$. Then for any $n_0, \ldots, n_k$, if $R(n_0, \ldots, n_k)$ is true, then $\chi_R(n_0, \ldots, n_k) = 1$, in which case **Q** proves $\varphi_{\chi_R}(\overline{n}_0, \ldots, \overline{n}_k, \overline{1})$, and so **Q** proves $\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$. On the other hand if $R(n_0, \ldots, n_k)$ is false, then $\chi_R(n_0, \ldots, n_k) = 0$. This means that **Q** proves $\varphi_{\chi_R}(\overline{n}_0, \ldots, \overline{n}_k, y) \to y = \overline{0}$. Since **Q** proves $\neg(\overline{0} = \overline{1})$, **Q** proves $\neg\varphi_{\chi_R}(\overline{n}_0, \ldots, \overline{n}_k, \overline{1})$, and so it proves $\neg\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$. □

12.9 Undecidability

We call a theory **T** *undecidable* if there is no computational procedure which, after finitely many steps and unfailingly, provides a correct answer to the question "does **T** prove φ?" for any sentence φ in the language of **T**. So **Q** would be decidable iff there were a computational procedure which decides, given a sentence φ in the language of arithmetic, whether $\mathbf{Q} \vdash \varphi$ or not. We can make this more precise by asking: Is the relation $\mathrm{Prov}_{\mathbf{Q}}(y)$, which holds of y iff y is the Gödel number of a sentence provable in **Q**, recursive? The answer is: no.

Theorem 12.14. **Q** *is undecidable, i.e., the relation*

$$\mathrm{Prov}_{\mathbf{Q}}(y) \Leftrightarrow \mathrm{Sent}(y) \wedge \exists x\, \mathrm{Pr}_{\mathbf{Q}}(x, y)$$

is not recursive.

Proof. Suppose it were. Then we could solve the halting problem as follows: Given e and n, we know that $\varphi_e(n) \downarrow$ iff there is an s such that $T(e, n, s)$, where T is Kleene's predicate from Theorem 6.8. Since T is primitive recursive it is representable in **Q** by a formula ψ_T, that is, $\mathbf{Q} \vdash \psi_T(\overline{e}, \overline{n}, \overline{s})$ iff $T(e, n, s)$. If

$\mathbf{Q} \vdash \psi_T(\overline{e}, \overline{n}, \overline{s})$ then also $\mathbf{Q} \vdash \exists y\, \psi_T(\overline{e}, \overline{n}, y)$. If no such s exists, then $\mathbf{Q} \vdash \neg\psi_T(\overline{e}, \overline{n}, \overline{s})$ for every s. But $\mathbf{Q}$ is ω-consistent, i.e., if $\mathbf{Q} \vdash \neg\varphi(\overline{n})$ for every $n \in \mathbb{N}$, then $\mathbf{Q} \nvdash \exists y\, \varphi(y)$. We know this because the axioms of $\mathbf{Q}$ are true in the standard model $\mathfrak{N}$. So, $\mathbf{Q} \nvdash \exists y\, \psi_T(\overline{e}, \overline{n}, y)$. In other words, $\mathbf{Q} \vdash \exists y\, \psi_T(\overline{e}, \overline{n}, y)$ iff there is an s such that $T(e, n, s)$, i.e., iff $\varphi_e(n) \downarrow$. From e and n we can compute $\#(\exists y\, \psi_T(\overline{e}, \overline{n}, y))$, let $g(e, n)$ be the primitive recursive function which does that. So

$$h(e, n) = \begin{cases} 1 & \text{if } \mathrm{Pr}_{\mathbf{Q}}(g(e, n)) \\ 0 & \text{otherwise.} \end{cases}$$

This would show that h is recursive if $\mathrm{Pr}_{\mathbf{Q}}$ is. But h is not recursive, by Theorem 6.9, so $\mathrm{Pr}_{\mathbf{Q}}$ cannot be either. □

Corollary 12.15. *First-order logic is undecidable.*

Proof. If first-order logic were decidable, provability in $\mathbf{Q}$ would be as well, since $\mathbf{Q} \vdash \varphi$ iff $\vdash \omega \to \varphi$, where ω is the conjunction of the axioms of $\mathbf{Q}$. □

Chapter 13

Theories and Computability

13.1 Introduction

We have the following:

1. A definition of what it means for a function to be representable in **Q** (Definition 12.1)
2. a definition of what it means for a relation to be representable in **Q** (Definition 12.12)
3. a theorem asserting that the representable functions of **Q** are exactly the computable ones (Theorem 12.2)
4. a theorem asserting that the representable relations of **Q** are exactly the computable ones Theorem 12.13)

A *theory* is a set of sentences that is deductively closed, that is, with the property that whenever T proves φ then φ is in T. It is probably best to think of a theory as being a collection of sentences, together with all the things that these sentences imply. From now on, I will use **Q** to refer to the *theory* consisting of the set of sentences derivable from the eight axioms in section 12.1. Remember that we can code formula of **Q** as numbers; if φ is such a formula, let $\#(\varphi)$ denote the number coding φ. Modulo this coding, we can now ask whether various sets of formulas are computable or not.

13.2 Q is c.e.-complete

Theorem 13.1. **Q** *is c.e. but not decidable. In fact, it is a complete c.e. set.*

Proof. It is not hard to see that **Q** is c.e., since it is the set of (codes for) sentences y such that there is a proof x of y in **Q**:

$$Q = \{y : \exists x \, \mathrm{Pr}_{\mathbf{Q}}(x, y)\}.$$

But we know that $\mathrm{Pr}_{\mathbf{Q}}(x, y)$ is computable (in fact, primitive recursive), and any set that can be written in the above form is c.e.

Saying that it is a complete c.e. set is equivalent to saying that $K \leq_m Q$, where $K = \{x : \varphi_x(x) \downarrow\}$. So let us show that K is reducible to $\mathbf{Q}$. Since Kleene's predicate $T(e, x, s)$ is primitive recursive, it is representable in $\mathbf{Q}$, say, by φ_T. Then for every x, we have

$$\begin{aligned} x \in K &\to \exists s\, T(x, x, s) \\ &\to \exists s\, (\mathbf{Q} \vdash \varphi_T(\overline{x}, \overline{x}, \overline{s})) \\ &\to \mathbf{Q} \vdash \exists s\, \varphi_T(\overline{x}, \overline{x}, s). \end{aligned}$$

Conversely, if $\mathbf{Q} \vdash \exists s\, \varphi_T(\overline{x}, \overline{x}, s)$, then, in fact, for some natural number n the formula $\varphi_T(\overline{x}, \overline{x}, \overline{n})$ must be true. Now, if $T(x, x, n)$ were false, $\mathbf{Q}$ would prove $\neg\varphi_T(\overline{x}, \overline{x}, \overline{n})$, since φ_T represents T. But then $\mathbf{Q}$ proves a false formula, which is a contradiction. So $T(x, x, n)$ must be true, which implies $\varphi_x(x) \downarrow$.

In short, we have that for every x, x is in K if and only if $\mathbf{Q}$ proves $\exists s\, T(\overline{x}, \overline{x}, s)$. So the function f which takes x to (a code for) the sentence $\exists s\, T(\overline{x}, \overline{x}, s)$ is a reduction of K to $\mathbf{Q}$. □

13.3 ω-Consistent Extensions of Q are Undecidable

The proof that $\mathbf{Q}$ is c.e.-complete relied on the fact that any sentence provable in $\mathbf{Q}$ is "true" of the natural numbers. The next definition and theorem strengthen this theorem, by pinpointing just those aspects of "truth" that were needed in the proof above. Don't dwell on this theorem too long, though, because we will soon strengthen it even further. We include it mainly for historical purposes: Gödel's original paper used the notion of ω-consistency, but his result was strengthened by replacing ω-consistency with ordinary consistency soon after.

Definition 13.2. A theory $\mathbf{T}$ is ω-consistent if the following holds: if $\exists x\, \varphi(x)$ is any sentence and $\mathbf{T}$ proves $\neg\varphi(\overline{0})$, $\neg\varphi(\overline{1})$, $\neg\varphi(\overline{2})$, ... then $\mathbf{T}$ does not prove $\exists x\, \varphi(x)$.

Theorem 13.3. *Let $\mathbf{T}$ be any ω-consistent theory that includes $\mathbf{Q}$. Then $\mathbf{T}$ is not decidable.*

Proof. If $\mathbf{T}$ includes $\mathbf{Q}$, then $\mathbf{T}$ represents the computable functions and relations. We need only modify the previous proof. As above, if $x \in K$, then $\mathbf{T}$ proves $\exists s\, \varphi_T(\overline{x}, \overline{x}, s)$. Conversely, suppose $\mathbf{T}$ proves $\exists s\, \varphi_T(\overline{x}, \overline{x}, s)$. Then x must be in K: otherwise, there is no halting computation of machine x on input x; since φ_T represents Kleene's T relation, $\mathbf{T}$ proves $\neg\varphi_T(\overline{x}, \overline{x}, \overline{0})$, $\neg\varphi_T(\overline{x}, \overline{x}, \overline{1})$, ..., making $\mathbf{T}$ ω-inconsistent. □

13.4 Consistent Extensions of Q are Undecidable

Remember that a theory is *consistent* if it does not prove φ and $\neg\varphi$ for any formula φ. Since anything follows from a contradiction, an inconsistent theory is trivial: every sentence is provable. Clearly, if a theory if ω-consistent, then it is consistent. But being consistent is a weaker requirement (i.e., there are theories that are consistent but not ω-consistent — we will see an example soon). We can weaken the assumption in Definition 13.2 to simple consistency to obatin a stronger theorem.

Lemma 13.4. *There is no "universal computable relation." That is, there is no binary computable relation $R(x, y)$, with the following property: whenever $S(y)$ is a unary computable relation, there is some k such that for every y, $S(y)$ is true if and only if $R(k, y)$ is true.*

Proof. Suppose $R(x, y)$ is a universal computable relation. Let $S(y)$ be the relation $\neg R(y, y)$. Since $S(y)$ is computable, for some k, $S(y)$ is equivalent to $R(k, y)$. But then we have that $S(k)$ is equivalent to both $R(k, k)$ and $\neg R(k, k)$, which is a contradiction. □

Theorem 13.5. *Let* **T** *be any consistent theory that includes* **Q**. *Then* **T** *is not decidable.*

Proof. Suppose **T** is a consistent, decidable extension of **Q**. We will obtain a contradiction by using **T** to define a universal computable relation.

Let $R(x, y)$ hold if and only if

x codes a formula $\theta(u)$, and **T** proves $\theta(\overline{y})$.

Since we are assuming that **T** is decidable, R is computable. Let us show that R is universal. If $S(y)$ is any computable relation, then it is representable in **Q** (and hence **T**) by a formula $\theta_S(u)$. Then for every n, we have

$$\begin{aligned} S(\overline{n}) &\rightarrow T \vdash \theta_S(\overline{n}) \\ &\rightarrow R(\#(\theta_S(u)), n) \end{aligned}$$

and

$$\begin{aligned} \neg S(\overline{n}) &\rightarrow T \vdash \neg\theta_S(\overline{n}) \\ &\rightarrow T \nvdash \theta_S(\overline{n}) \quad \text{(since } \mathbf{T} \text{ is consistent)} \\ &\rightarrow \neg R(\#(\theta_S(u)), n). \end{aligned}$$

That is, for every y, $S(y)$ is true if and only if $R(\#(\theta_S(u)), y)$ is. So R is universal, and we have the contradiction we were looking for. □

Let "true arithmetic" be the theory $\{\varphi : \mathbb{N} \models \varphi\}$, that is, the set of sentences in the language of arithmetic that are true in the standard interpretation.

Corollary 13.6. *True arithmetic is not decidable.*

13.5 Computably Axiomatizable Theories

A theory **T** is said to be *computably axiomatizable* if it has a computable set of axioms A. (Saying that A is a set of axioms for **T** means $T = \{\varphi : A \vdash \varphi\}$.) Any "reasonable" axiomatization of the natural numbers will have this property. In particular, any theory with a finite set of axioms is computably axiomatizable. The phrase "effectively axiomatizable" is also commonly used.

Lemma 13.7. *Suppose* **T** *is computably axiomatizable. Then* **T** *is computably enumerable.*

Proof. Suppose A is a computable set of axioms for **T**. To determine if $\varphi \in T$, just search for a proof of φ from the axioms.

Put slightly differently, φ is in **T** if and only if there is a finite list of axioms $\psi_1, \ldots, \psi_k$ in A and a proof of $(\psi_1 \wedge \cdots \wedge \psi_k) \rightarrow \varphi$ in first-order logic. But we already know that any set with a definition of the form "there exists ... such that ..." is c.e., provided the second "..." is computable. □

13.6 Computably Axiomatizable Complete Theories are Decidable

A theory is said to be *complete* if for every sentence φ, either φ or $\neg\varphi$ is provable.

Lemma 13.8. *Suppose a theory* **T** *is complete and computably axiomatizable. Then* **T** *is decidable.*

Proof. Suppose **T** is complete and A is a computable set of axioms. If **T** is inconsistent, it is clearly computable. (Algorithm: "just say yes.") So we can assume that **T** is also consistent.

To decide whether or not a sentence φ is in **T**, simultaneously search for a proof of φ from A and a proof of $\neg\varphi$. Since **T** is complete, you are bound to find one or another; and since **T** is consistent, if you find a proof of $\neg\varphi$, there is no proof of φ.

Put in different terms, we already know that **T** is c.e.; so by a theorem we proved before, it suffices to show that the complement of **T** is c.e. But a formula φ is in $\overline{T}$ if and only if $\neg\varphi$ is in **T**; so $\overline{T} \leq_m T$. □

13.7 Q has no Complete, Consistent, Computably Axiomatized Extensions

Theorem 13.9. *There is no complete, consistent, computably axiomatized extension of* **Q**.

Proof. We already know that there is no consistent, decidable extension of **Q**. But if **T** is complete and computably axiomatized, then it is decidable. □

This theorems is not that far from Gödel's original 1931 formulation of the First Incompleteness Theorem. Aside from the more modern terminology, the key differences are this: Gödel has "ω-consistent" instead of "consistent"; and he could not say "computably axiomatized" in full generality, since the formal notion of computability was not in place yet. (The formal models of computability were developed over the following decade, in large part by Gödel, and in large part to be able to characterize the kinds of theories that are susceptible to the Gödel phenomenon.)

The theorem says you can't have it all, namely, completeness, consistency, and computable axiomatizability. If you give up any one of these, though, you can have the other two: **Q** is consistent and computably axiomatized, but not complete; the inconsistent theory is complete, and computably axiomatized (say, by $\{0 \neq 0\}$), but not consistent; and the set of true sentence of arithmetic is complete and consistent, but it is not computably axiomatized.

13.8 Sentences Provable and Refutable in Q are Computably Inseparable

Let $\bar{\mathbf{Q}}$ be the set of sentences whose *negations* are provable in **Q**, i.e., $\bar{\mathbf{Q}} = \{\varphi : \mathbf{Q} \vdash \neg\varphi\}$. Remember that disjoint sets A and B are said to be computably inseparable if there is no computable set C such that $A \subseteq C$ and $B \subseteq \overline{C}$.

Lemma 13.10. **Q** *and* $\bar{\mathbf{Q}}$ *are computably inseparable.*

Proof. Suppose C is a computable set such that $\mathbf{Q} \subseteq C$ and $\bar{\mathbf{Q}} \subseteq \overline{C}$. Let $R(x, y)$ be the relation

> x codes a formula $\theta(u)$ and $\theta(\overline{y})$ is in C.

We will show that $R(x, y)$ is a universal computable relation, yielding a contradiction.

Suppose $S(y)$ is computable, represented by $\theta_S(u)$ in **Q**. Then

$$\begin{aligned} S(\overline{n}) &\rightarrow \mathbf{Q} \vdash \theta_S(\overline{n}) \\ &\rightarrow \theta_S(\overline{n}) \in C \end{aligned}$$

and

$$\begin{aligned} \neg S(\overline{n}) &\rightarrow \mathbf{Q} \vdash \neg\theta_S(\overline{n}) \\ &\rightarrow \theta_S(\overline{n}) \in \bar{\mathbf{Q}} \\ &\rightarrow \theta_S(\overline{n}) \notin C \end{aligned}$$

So $S(y)$ is equivalent to $R(\#(\theta_S(\overline{u})), y)$. □

13.9 Theories Consistent with Q are Undecidable

The following theorem says that not only is **Q** undecidable, but, in fact, any theory that does not disagree with **Q** is undecidable.

Theorem 13.11. *Let* **T** *be any theory in the language of arithmetic that is consistent with* **Q** *(i.e.,* $\mathbf{T} \cup \mathbf{Q}$ *is consistent). Then* **T** *is undecidable.*

Proof. Remember that **Q** has a finite set of axioms, $\varphi_1, \ldots, \varphi_8$. We can even replace these by a single axiom, $\alpha = \varphi_1 \wedge \cdots \wedge \varphi_8$.

Suppose **T** is a decidable theory consistent with **Q**. Let

$$C = \{\varphi : \mathbf{T} \vdash \alpha \to \varphi\}.$$

We show that C would be a computable separation of **Q** and $\bar{\mathbf{Q}}$, a contradiction. First, if φ is in **Q**, then φ is provable from the axioms of **Q**; by the deduction theorem, there is a proof of $\alpha \to \varphi$ in first-order logic. So φ is in C.

On the other hand, if φ is in $\bar{\mathbf{Q}}$, then there is a proof of $\alpha \to \neg\varphi$ in first-order logic. If **T** also proves $\alpha \to \varphi$, then **T** proves $\neg\alpha$, in which case $\mathbf{T} \cup \mathbf{Q}$ is inconsistent. But we are assuming $\mathbf{T} \cup \mathbf{Q}$ is consistent, so **T** does not prove $\alpha \to \varphi$, and so φ is not in C.

We've shown that if φ is in **Q**, then it is in C, and if φ is in Q', then it is in $\overline{C}$. So C is a computable separation, which is the contradiction we were looking for. □

This theorem is very powerful. For example, it implies:

Corollary 13.12. *First-order logic for the language of arithmetic (that is, the set* $\{\varphi : \varphi$ *is provable in first-order logic*$\}$*) is undecidable.*

Proof. First-order logic is the set of consequences of $\emptyset$, which is consistent with **Q**. □

13.10 Theories In Which Q is Intepretable are Undecidable

We can strengthen these results even more. Informally, an interpretation of a language $\mathcal{L}_1$ in another language $\mathcal{L}_2$ involves defining the universe, relation symbols, and function symbols of $\mathcal{L}_1$ with formulas in $\mathcal{L}_2$. Though we won't take the time to do this, one can make this definition precise.

Theorem 13.13. *Suppose* **T** *is a theory in a language in which one can interpret the language of arithmetic, in such a way that* **T** *is consistent with the interpretation of* **Q**. *Then* **T** *is undecidable. If* **T** *proves the interpretation of the axioms of* **Q**, *then no consistent extension of* **T** *is decidable.*

The proof is just a small modification of the proof of the last theorem; one could use a counterexample to get a separation of **Q** and $\bar{\mathbf{Q}}$. One can take **ZFC**, Zermelo Fraenkel set theory with the axiom of choice, to be an axiomatic foundation that is powerful enough to carry out a good deal of ordinary mathematics. In **ZFC** one can define the natural numbers, and via this interpretation, the axioms of **Q** are true. So we have

Corollary 13.14. *There is no decidable extension of* **ZFC**.

Corollary 13.15. *There is no complete, consistent, computably axiomatized extension of* **ZFC**.

The language of **ZFC** has only a single binary relation, $\in$. (In fact, you don't even need equality.) So we have

Corollary 13.16. *First-order logic for any language with a binary relation symbol is undecidable.*

This result extends to any language with two unary function symbols, since one can use these to simulate a binary relation symbol. The results just cited are tight: it turns out that first-order logic for a language with only *unary* relation symbols and at most one *unary* function symbol is decidable.

One more bit of trivia. We know that the set of sentences in the language $0, S, +, \times, <$ true in the standard model is undecidable. In fact, one can define $<$ in terms of the other symbols, and then one can define $+$ in terms of $\times$ and S. So the set of true sentences in the language $0, S, \times$ is undecidable. On the other hand, Presburger has shown that the set of sentences in the language $0, S, +$ true in the language of arithmetic is decidable. The procedure is computationally infeasible, however.

Chapter 14

Incompleteness and Provability

14.1 Introduction

Hilbert thought that a system of axioms for a mathematical structure, such as the natural numbers, is inadequate unless it allows one to derive all true statements about the structure. Combined with his later interest in formal systems of deduction, this suggests that one should try to guarantee that, say, the formal system one is using to reason about the natural numbers is not only consistent, but also *complete*, i.e., every statement is either provable or refutable. Gödel's first incompleteness theorem shows that no such system of axioms exists: there is no complete, consistent, effectively axiomatized formal system for arithmetic. In fact, no "sufficiently strong," consistent, effectively axiomatized mathematical theory is complete.

A more important goal of Hilbert's, the centerpiece of his program for the justification of modern ("classical") mathematics, was to find finitary consistency proofs for formal systems representing classical reasoning. With regard to Hilbert's program, then, Gödel's second incompleteness theorem was a much bigger blow.

The second incompleteness theorem can be stated in vague terms, like the first incompleteness theorem. Roughly speaking, then, it says that no sufficiently strong theory of arithmetic can prove its own consistency. We will have to take "sufficiently strong" to include a little bit more than **Q**.

The idea behind Gödel's original proof of the incompleteness theorem can be found in the Epimenides paradox. Epimenides, a Cretin, asserted that all Cretans are liars; a more direct form of the paradox is the assertion "this sentence is false." Essentially, by replacing truth with provability, Gödel was able to formalize a sentence which, in essence, asserts "this sentence is not provable." Assuming ω-consistency—a property stronger than consistency—Gödel was able to show that this sentence is neither provable nor refutable from the system of axioms he was considering.

The first challenge is to understand how one can construct a sentence that

refers to itself. For every formula φ in the language of **Q**, let $\ulcorner\varphi\urcorner$ denote the numeral corresponding to $\#(\varphi)$. Think about what this means: φ is a formula in the language of **Q**, $\#(\varphi)$ is a natural number, and $\ulcorner\varphi\urcorner$ is a *term* in the language of **Q**. So every formula φ in the language of **Q** has a *name*, $\ulcorner\varphi\urcorner$, which is a term in the language of **Q**; this provides us with a conceptual framework in which formulas in the language of **Q** can "say" things about other formulas. The following lemma is known as Gödel's fixed-point lemma.

Lemma 14.1. *Let* **T** *be any theory extending* **Q**, *and let* $\psi(x)$ *be any formula with free variable* x. *Then there is a sentence* φ *such that* **T** *proves* $\varphi \leftrightarrow \psi(\ulcorner\varphi\urcorner)$.

The lemma asserts that given any property $\psi(x)$, there is a sentence φ that asserts "$\psi(x)$ is true of me."

How can we construct such a sentence? Consider the following version of the Epimenides paradox, due to Quine:

> "Yields falsehood when preceded by its quotation" yields falsehood when preceded by its quotation.

This sentence is not directly self-referential. It simply makes an assertion about the syntactic objects between quotes, and, in doing so, it is on par with sentences like

1. "Robert" is a nice name.
2. "I ran." is a short sentence.
3. "Has three words" has three words.

But what happens when one takes the phrase "yields falsehood when preceded by its quotation," and precedes it with a quoted version of itself? Then one has the original sentence! In short, the sentence asserts that it is false.

14.2 The Fixed-Point Lemma

Let $\mathrm{diag}(y)$ be the computable (in fact, primitive recursive) function that does the following: if y is the Gödel number of a formula $\psi(x)$, $\mathrm{diag}(y)$ returns the Gödel number of $\psi(\ulcorner\underline{\psi(x)}\urcorner)$. ($\ulcorner\psi(x)\urcorner$ is the standard numeral of the Gödel number of $\psi(x)$, i.e., $\#(\psi(x))$). If $diag$ were a function symbol in **T** representing the function diag, we could take φ to be the formula $\psi(diag(\ulcorner\psi(diag(x))\urcorner))$. Notice that

$$\begin{aligned}\mathrm{diag}(\#(\psi(diag(x)))) &= \#(\psi(diag(\ulcorner\psi(diag(x))\urcorner))\\ &= \#(\varphi).\end{aligned}$$

Assuming **T** can prove

$$diag(\ulcorner\psi(diag(x))\urcorner) = \ulcorner\varphi\urcorner,$$

it can prove $\psi(diag(\ulcorner\psi(diag(x))\urcorner)) \leftrightarrow \psi(\ulcorner\varphi\urcorner)$. But the left hand side is, by definition, φ.

In general, $diag$ will not be a function symbol of **T**. But since **T** extends **Q**, the function diag will be *represented* in **T** by some formula $\theta_{\text{diag}}(x, y)$. So instead of writing $\psi(diag(x))$ we will have to write $\exists y\,(\theta_{\text{diag}}(x, y) \wedge \psi(y))$. Otherwise, the proof sketched above goes through.

Lemma 14.2. *Let* **T** *be any theory extending* **Q**, *and let* $\psi(x)$ *be any formula with free variable* x. *Then there is a sentence* φ *such that* **T** *proves* $\varphi \leftrightarrow \psi(\ulcorner\varphi\urcorner)$.

Proof. Given $\psi(x)$, let $\alpha(x)$ be the formula $\exists y\,(\theta_{\text{diag}}(x, y) \wedge \psi(y))$ and let φ be the formula $\alpha(\ulcorner\alpha(x)\urcorner)$.

Since θ_{diag} represents diag, **T** can prove

$$\forall y\,(\theta_{\text{diag}}(\ulcorner\alpha(x)\urcorner, y) \leftrightarrow y = \overline{\text{diag}(\ulcorner\alpha(x)\urcorner)}).$$

But by definition, $\text{diag}(\#(\alpha(x))) = \#(\alpha(\ulcorner\alpha(x)\urcorner)) = \#(\varphi)$, so **T** can prove

$$\forall y\,(\theta_{\text{diag}}(\ulcorner\alpha(x)\urcorner, y) \leftrightarrow y = \ulcorner\varphi\urcorner).$$

Going back to the definition of $\alpha(x)$, we see $\alpha(\ulcorner\alpha(x)\urcorner)$ is just the formula

$$\exists y\,(\theta_{\text{diag}}(\ulcorner\alpha(x)\urcorner, y) \wedge \psi(y)).$$

Using the last two sentences and ordinary first-order logic, one can then prove

$$\alpha(\ulcorner\alpha(x)\urcorner) \leftrightarrow \psi(\ulcorner\varphi\urcorner).$$

But the left-hand side is just φ. □

You should compare this to the proof of the fixed-point lemma in computability theory. The difference is that here we want to define a *statement* in terms of itself, whereas there we wanted to define a *function* in terms of itself; this difference aside, it is really the same idea.

14.3 The First Incompleteness Theorem

We can now describe Gödel's original proof of the first incompleteness theorem. Let **T** be any computably axiomatized theory in a language extending the language of arithmetic, such that **T** includes the axioms of **Q**. This means that, in particular, **T** represents computable functions and relations.

We have argued that, given a reasonable coding of formulas and proofs as numbers, the relation $\text{Pr}_{\mathbf{T}}(x, y)$ is computable, where $\text{Pr}_{\mathbf{T}}(x, y)$ holds if and only if x is a proof of formula y in **T**. In fact, for the particular theory that Gödel had in mind, Gödel was able to show that this relation is primitive recursive, using the list of 45 functions and relations in his paper. The 45th

relation, xBy, is just $\mathrm{Pr}_{\mathbf{T}}(x, y)$ for his particular choice of $\mathbf{T}$. Remember that where Gödel uses the word "recursive" in his paper, we would now use the phrase "primitive recursive."

Since $\mathrm{Pr}_{\mathbf{T}}(x, y)$ is computable, it is representable in $\mathbf{T}$. We will use $\mathsf{Pr}_{\mathbf{T}}(x, y)$ to refer to the formula that represents it. Let $\mathsf{Prov}_{\mathbf{T}}(y)$ be the formula $\exists x\, \mathsf{Pr}_{\mathbf{T}}(x, y)$. This describes the 46th relation, $\mathrm{Bew}(y)$, on Gödel's list. As Gödel notes, this is the only relation that "cannot be asserted to be recursive." What he probably meant is this: from the definition, it is not clear that it is computable; and later developments, in fact, show that it isn't.

Definition 14.3. A theory $\mathbf{T}$ is ω-consistent if the following holds: if $\exists x\, \varphi(x)$ is any sentence and $\mathbf{T}$ proves $\neg\varphi(\overline{0})$, $\neg\varphi(\overline{1})$, $\neg\varphi(\overline{2})$, ... then $\mathbf{T}$ does not prove $\exists x\, \varphi(x)$.

We can now prove the following.

Theorem 14.4. *Let $\mathbf{T}$ be any ω-consistent, computably axiomatized theory extending $\mathbf{Q}$. Then $\mathbf{T}$ is not complete.*

Proof. Let $\mathbf{T}$ be any computably axiomatized theory containing $\mathbf{Q}$, and let $\mathsf{Prov}_{\mathbf{T}}(y)$ be the formula we described above. By the fixed-point lemma, there is a formula $\gamma_{\mathbf{T}}$ such that $\mathbf{T}$ proves

$$\gamma_{\mathbf{T}} \leftrightarrow \neg\mathsf{Prov}_{\mathbf{T}}(\ulcorner\gamma_{\mathbf{T}}\urcorner). \tag{14.1}$$

Note that φ says, in essence, "I am not provable."

We claim that

1. If $\mathbf{T}$ is consistent, $\mathbf{T}$ doesn't prove $\gamma_{\mathbf{T}}$
2. If $\mathbf{T}$ is ω-consistent, $\mathbf{T}$ doesn't prove $\neg\gamma_{\mathbf{T}}$.

This means that if $\mathbf{T}$ is ω-consistent, it is incomplete, since it proves neither $\gamma_{\mathbf{T}}$ nor $\neg\gamma_{\mathbf{T}}$. Let us take each claim in turn.

Suppose $\mathbf{T}$ proves $\gamma_{\mathbf{T}}$. Then there *is* a proof, and so, for some number m, the relation $\mathrm{Pr}_{\mathbf{T}}(m, \#(\gamma_{\mathbf{T}}))$ holds. But then $\mathbf{T}$ proves the sentence $\mathsf{Pr}_{\mathbf{T}}(\overline{m}, \ulcorner\gamma_{\mathbf{T}}\urcorner)$. So $\mathbf{T}$ proves $\exists x\, \mathsf{Pr}_{\mathbf{T}}(x, \ulcorner\gamma_{\mathbf{T}}\urcorner)$, which is, by definition, $\mathsf{Prov}_{\mathbf{T}}(\ulcorner\gamma_{\mathbf{T}}\urcorner)$. By eq. (14.1), $\mathbf{T}$ proves $\neg\gamma_{\mathbf{T}}$. We have shown that if $\mathbf{T}$ proves $\gamma_{\mathbf{T}}$, then it also proves $\neg\gamma_{\mathbf{T}}$, and hence it is inconsistent.

For the second claim, let us show that if $\mathbf{T}$ proves $\neg\gamma_{\mathbf{T}}$, then it is ω-inconsistent. Suppose $\mathbf{T}$ proves $\neg\gamma_{\mathbf{T}}$. If $\mathbf{T}$ is inconsistent, it is ω-inconsistent, and we are done. Otherwise, $\mathbf{T}$ is consistent, so it does not prove $\gamma_{\mathbf{T}}$. Since there is no proof of $\gamma_{\mathbf{T}}$ in $\mathbf{T}$, $\mathbf{T}$ proves

$$\neg\mathsf{Pr}_{\mathbf{T}}(\overline{0}, \ulcorner\gamma_{\mathbf{T}}\urcorner), \neg\mathsf{Pr}_{\mathbf{T}}(\overline{1}, \ulcorner\gamma_{\mathbf{T}}\urcorner), \neg\mathsf{Pr}_{\mathbf{T}}(\overline{2}, \ulcorner\gamma_{\mathbf{T}}\urcorner), \ldots$$

On the other hand, by eq. (14.1), $\neg\gamma_{\mathbf{T}}$ is equivalent to $\exists x\, \mathsf{Pr}_{\mathbf{T}}(x, \ulcorner\gamma_{\mathbf{T}}\urcorner)$. So $\mathbf{T}$ is ω-inconsistent. □

14.4 Rosser's Theorem

Can we modify Gödel's proof to get a stronger result, replacing "ω-consistent" with simply "consistent"? The answer is "yes," using a trick discovered by Rosser. Let $\mathrm{not}(x)$ be the primitive recursive function which does the following: if x is the code of a formula φ, $\mathrm{not}(x)$ is a code of $\neg\varphi$. To simplify matters, assume $\mathbf{T}$ has a function symbol not such that for any formula φ, $\mathbf{T}$ proves $not(\ulcorner\varphi\urcorner) = \ulcorner\neg\varphi\urcorner$. This is not a major assumption; since $\mathrm{not}(x)$ is computable, it is represented in $\mathbf{T}$ by some formula $\theta_{\mathrm{not}}(x, y)$, and we could eliminate the reference to the function symbol in the same way that we avoided using a function symbol $diag$ in the proof of the fixed-point lemma.

Rosser's trick is to use a "modified" provability predicate $\mathsf{RProv}_T(y)$, defined to be

$$\exists x\,(\mathsf{Pr}_{\mathbf{T}}(x, y) \wedge \forall z\,(z < x \rightarrow \neg\mathsf{Pr}_{\mathbf{T}}(z, not(y)))).$$

Roughly, $\mathsf{RProv}_{\mathbf{T}}(y)$ says "there is a proof of y in $\mathbf{T}$, and there is no shorter proof of the negation of y." (You might find it convenient to read $\mathsf{RProv}_{\mathbf{T}}(y)$ as "y is shmovable.") Assuming $\mathbf{T}$ is consistent, $\mathsf{RProv}_{\mathbf{T}}(y)$ is true of the same numbers as $\mathsf{Prov}_{\mathbf{T}}(y)$; but from the point of view of *provability* in $\mathbf{T}$ (and we now know that there is a difference between truth and provability!) the two have different properties.

By the fixed-point lemma, there is a formula $\rho_{\mathbf{T}}$ such that $\mathbf{T}$ proves

$$\rho_{\mathbf{T}} \leftrightarrow \neg\mathsf{RProv}_{\mathbf{T}}(\ulcorner\rho_{\mathbf{T}}\urcorner).$$

In contrast to the proof above, here we claim that if $\mathbf{T}$ is consistent, $\mathbf{T}$ doesn't prove $\rho_{\mathbf{T}}$, and $\mathbf{T}$ also doesn't prove $\neg\rho_{\mathbf{T}}$. (In other words, we don't need the assumption of ω-consistency.)

By comparison to the proof of Theorem 13.9, the proofs of Theorem 14.4 and its improvement by Rosser explicitly exhibit a statement φ that is independent of $\mathbf{T}$. In the former, you have to dig to extract it from the argument. The Gödel-Rosser methods therefore have the advantage of making the independent statement perfectly clear.

14.5 Comparison with Gödel's Original Paper

It is worthwhile to spend some time with Gödel's 1931 paper. The introduction sketches the ideas we have just discussed. Even if you just skim through the paper, it is easy to see what is going on at each stage: first Gödel describes the formal system P (syntax, axioms, proof rules); then he defines the primitive recursive functions and relations; then he shows that xBy is primitive recursive, and argues that the primitive recursive functions and relations are represented in $\mathbf{P}$. He then goes on to prove the incompleteness theorem, as above. In section 3, he shows that one can take the unprovable assertion to be a sentence in the language of arithmetic. This is the origin of the β-lemma,

which is what we also used to handle sequences in showing that the recursive functions are representable in **Q**. Gödel doesn't go so far to isolate a minimal set of axioms that suffice, but we now know that **Q** will do the trick. Finally, in Section 4, he sketches a proof of the second incompleteness theorem.

14.6 The Provability Conditions for PA

Peano arithmetic, or **PA**, is the theory extending **Q** with induction axioms for all formulas. In other words, one adds to **Q** axioms of the form

$$(\varphi(0) \wedge \forall x\,(\varphi(x) \to \varphi(x'))) \to \forall x\,\varphi(x)$$

for every formula φ. Notice that this is really a *schema*, which is to say, infinitely many axioms (and it turns out that **PA** is *not* finitely axiomatizable). But since one can effectively determine whether or not a string of symbols is an instance of an induction axiom, the set of axioms for **PA** is computable. **PA** is a much more robust theory than **Q**. For example, one can easily prove that addition and multiplication are commutative, using induction in the usual way. In fact, most finitary number-theoretic and combinatorial arguments can be carried out in **PA**.

Since **PA** is computably axiomatized, the provability predicate $\mathrm{Pr}_{\mathbf{PA}}(x, y)$ is computable and hence represented in **Q** (and so, in **PA**). As before, I will take $\mathrm{Pr}_{\mathbf{PA}}(x, y)$ to denote the formula representing the relation. Let $\mathrm{Prov}_{\mathbf{PA}}(y)$ be the formula $\exists x\,\mathrm{Pr}_{\mathbf{PA}}(x, y)$, which, intuitively says, "y is provable from the axioms of **PA**." The reason we need a little bit more than the axioms of **Q** is we need to know that the theory we are using is strong enough to prove a few basic facts about this provability predicate. In fact, what we need are the following facts:

1. If $\mathbf{PA} \vdash \varphi$, then $\mathbf{PA} \vdash \mathrm{Prov}_{\mathbf{PA}}(\ulcorner\varphi\urcorner)$
2. For every formula φ and ψ, $\mathbf{PA} \vdash \mathrm{Prov}_{\mathbf{PA}}(\ulcorner\varphi \to \psi\urcorner) \to (\mathrm{Prov}_{\mathbf{PA}}(\ulcorner\varphi\urcorner) \to \mathrm{Prov}_{\mathbf{PA}}(\ulcorner\psi\urcorner))$
3. For every formula φ, $\mathbf{PA} \vdash \mathrm{Prov}_{\mathbf{PA}}(\ulcorner\varphi\urcorner) \to \mathrm{Prov}_{\mathbf{PA}}(\ulcorner\mathrm{Prov}_{\mathbf{PA}}(\ulcorner\varphi\urcorner)\urcorner)$.

The only way to verify that these three properties hold is to describe the formula $\mathrm{Prov}_{\mathbf{PA}}(y)$ carefully and use the axioms of **PA** to describe the relevant formal proofs. Clauses 1 and 2 are easy; it is really clause 3 that requires work. (Think about what kind of work it entails…) Carrying out the details would be tedious and uninteresting, so here we will ask you to take it on faith that **PA** has the three properties listed above. A reasonable choice of $\mathrm{Prov}_{\mathbf{PA}}(y)$ will also satisfy

4. If **PA** proves $\mathrm{Prov}_{\mathbf{PA}}(\ulcorner\varphi\urcorner)$, then **PA** proves φ.

But we will not need this fact.

Incidentally, Gödel was lazy in the same way we are being now. At the end of the 1931 paper, he sketches the proof of the second incompleteness theorem, and promises the details in a later paper. He never got around to it; since everyone who understood the argument believed that it could be carried out (he did not need to fill in the details.)

14.7 The Second Incompleteness Theorem

How can we express the assertion that **PA** doesn't prove its own consistency? Saying **PA** is inconsistent amounts to saying that **PA** proves $0 = 1$. So we can take $\mathsf{Con}_{\mathbf{PA}}$ to be the formula $\neg\mathsf{Prov}_{\mathbf{PA}}(\ulcorner 0 = 1 \urcorner)$, and then the following theorem does the job:

Theorem 14.5. *Assuming* **PA** *is consistent, then* **PA** *does not prove* $\mathsf{Con}_{\mathbf{PA}}$.

It is important to note that the theorem depends on the particular representation of $\mathsf{Con}_{\mathbf{PA}}$ (i.e., the particular representation of $\mathsf{Prov}_{\mathbf{PA}}(y)$). All we will use is that the representation of $\mathsf{Prov}_{\mathbf{PA}}(y)$ has the three properties above, so the theorem generalizes to any theory with a provability predicate having these properties.

It is informative to read Gödel's sketch of an argument, since the theorem follows like a good punch line. It goes like this. Let $\gamma_{\mathbf{PA}}$ be the Gödel sentence that we constructed in the proof of Theorem 14.4. We have shown "If **PA** is consistent, then **PA** does not prove $\gamma_{\mathbf{PA}}$." If we formalize this *in* **PA**, we have a proof of

$$\mathsf{Con}_{\mathbf{PA}} \to \neg\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner).$$

Now suppose **PA** proves $\mathsf{Con}_{\mathbf{PA}}$. Then it proves $\neg\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner)$. But since $\gamma_{\mathbf{PA}}$ is a Gödel sentence, this is equivalent to $\gamma_{\mathbf{PA}}$. So **PA** proves $\gamma_{\mathbf{PA}}$.

But: we know that if **PA** is consistent, it doesn't prove $\gamma_{\mathbf{PA}}$! So if **PA** is consistent, it can't prove $\mathsf{Con}_{\mathbf{PA}}$.

To make the argument more precise, we will let $\gamma_{\mathbf{PA}}$ be the Gödel sentence for **PA** and use properties 1–3 above to show that **PA** proves $\mathsf{Con}_{\mathbf{PA}} \to \gamma_{\mathbf{PA}}$. This will show that **PA** doesn't prove $\mathsf{Con}_{\mathbf{PA}}$. Here is a sketch of the proof,

in **PA**:

$\gamma_{\mathbf{PA}} \to \neg \mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner)$	since $\gamma_{\mathbf{PA}}$ is a Gödel sentence
$\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \to \neg \mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \urcorner)$	by 1
$\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \to$ $\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \neg \mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \urcorner)$	by 2
$\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \to$ $\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \to 0 = 1 \urcorner)$	by 1 and 2
$\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \to$ $\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \urcorner)$	by 3
$\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \to \mathsf{Prov}_{\mathbf{PA}}(\ulcorner 0 = 1 \urcorner)$	using 1 and 2
$\mathsf{Con}_{\mathbf{PA}} \to \neg \mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner)$	by contraposition
$\mathsf{Con}_{\mathbf{PA}} \to \gamma_{\mathbf{PA}}$	since $\gamma_{\mathbf{PA}}$ is a Gödel sentence

The move from the third to the fourth line uses the fact that $\neg \mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner)$ is equivalent to $\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \gamma_{\mathbf{PA}} \urcorner) \to 0 = 1$ in **PA**. The more abstract version of the incompleteness theorem is as follows:

Theorem 14.6. *Let* $\mathbf{T}$ *be any theory extending* $\mathbf{Q}$ *and let* $\mathsf{Prov}_{\mathbf{T}}(y)$ *be any formula satisfying 1–3 for* $\mathbf{T}$*. Then if* $\mathbf{T}$ *is consistent, then* $\mathbf{T}$ *does not prove* $\mathsf{Con}_{\mathbf{T}}$*.*

The moral of the story is that no "reasonable" consistent theory for mathematics can prove its own consistency. Suppose $\mathbf{T}$ is a theory of mathematics that includes $\mathbf{Q}$ and Hilbert's "finitary" reasoning (whatever that may be). Then, the whole of $\mathbf{T}$ cannot prove the consistency of $\mathbf{T}$, and so, a fortiori, the finitary fragment can't prove the consistency of $\mathbf{T}$ either. In that sense, there cannot be a finitary consistency proof for "all of mathematics."

There is some leeway in interpreting the term finitary, and Gödel, in the 1931 paper, grants the possibility that something we may consider "finitary" may lie outside the kinds of mathematics Hilbert wanted to formalize. But Gödel was being charitable; today, it is hard to see how we might find something that can reasonably be called finitary but is not formalizable in, say, **ZFC**.

14.8 Löb's Theorem

In this section, we will consider a fun application of the fixed-point lemma. We now know that any "reasonable" theory of arithmetic is incomplete, which is to say, there are sentences φ that are neither provable nor refutable in the theory. One can ask whether, in general, a theory can prove "If I can prove φ, then it must be true." The answer is that, in general, it can't. More precisely, we have:

Theorem 14.7. *Let* $\mathbf{T}$ *be any theory extending* $\mathbf{Q}$*, and suppose* $\mathsf{Prov}_{\mathbf{T}}(y)$ *is a formula satisfying conditions 1–3 from section 14.7. If* $\mathbf{T}$ *proves* $\mathsf{Prov}_{\mathbf{T}}(\ulcorner\varphi\urcorner) \to \varphi$*, then in fact* $\mathbf{T}$ *proves* φ*.*

Put differently, if φ is not provable in $\mathbf{T}$, $\mathbf{T}$ can't prove $\mathsf{Prov}_{\mathbf{T}}(\ulcorner\varphi\urcorner) \to \varphi$. This is known as Löb's theorem.

The heuristic for the proof of Löb's theorem is a clever proof that Santa Claus exists. (If you don't like that conclusion, you are free to substitute any other conclusion you would like.) Here it is:

1. Let X be the sentence, "If X is true, then Santa Claus exists."
2. Suppose X is true.
3. Then what it says is true; i.e., if X is true, then Santa Claus exists.
4. Since we are assuming X is true, we can conclude that Santa Claus exists.
5. So, we have shown: "If X is true, then Santa Claus exists."
6. But this is just the statement X. So we have shown that X is true.
7. But then, by the argument above, Santa Claus exists.

A formalization of this idea, replacing "is true" with "is provable," yields the proof of Löb's theorem.

Proof. Suppose φ is a sentence such that $\mathbf{T}$ proves $\mathsf{Prov}_{\mathbf{T}}(\ulcorner\varphi\urcorner) \to \varphi$. Let $\psi(y)$ be the formula $\mathsf{Prov}_{\mathbf{T}}(y) \to \varphi$, and use the fixed-point lemma to find a sentence θ such that $\mathbf{T}$ proves $\theta \leftrightarrow \psi(\ulcorner\theta\urcorner)$. Then each of the following is provable in $\mathbf{T}$:

$\theta \to (\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner) \to \varphi)$	
$\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta \to (\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner) \to \varphi)\urcorner)$	by 1
$\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner) \to \mathsf{Prov}_{\mathbf{T}}(\ulcorner\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner) \to \varphi\urcorner)$	using 2
$\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner) \to$ $(\mathsf{Prov}_{\mathbf{T}}(\ulcorner\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner)\urcorner) \to \mathsf{Prov}_{\mathbf{T}}(\ulcorner\varphi\urcorner))$	using 2
$\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner) \to \mathsf{Prov}_{\mathbf{T}}(\ulcorner\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner)\urcorner)$	by 3
$\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner) \to \mathsf{Prov}_{\mathbf{T}}(\ulcorner\varphi\urcorner)$	
$\mathsf{Prov}_{\mathbf{T}}(\ulcorner\varphi\urcorner) \to \varphi$	by assumption
$\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner) \to \varphi$	
θ	def of θ
$\mathsf{Prov}_{\mathbf{T}}(\ulcorner\theta\urcorner)$	by 1
φ	

□

With Löb's theorem in hand, there is a short proof of the first incompleteness theorem (for theories having a provability predicate satisfying 1–3): if a theory proves $\mathsf{Prov}_{\mathbf{T}}(\ulcorner 0 = 1 \urcorner) \to 0 = 1$, it proves $0 = 1$.

14.9 The Undefinability of Truth

The notion of *definability* depends on having a formal semantics for the language of arithmetic. We have described a set of formulas and sentences in the language of arithmetic. The "intended interpretation" is to read such sentences as making assertions about the natural numbers, and such an assertion can be true or false. Let $\mathfrak{N}$ be the structure with domain $\mathbb{N}$ and the standard interpretation for the symbols in the language of arithmetic. Then $\mathfrak{N} \models \varphi$ means "φ is true in the standard interpretation."

Definition 14.8. A relation $R(x_1, \ldots, x_k)$ of natural numbers is *definable* in $\mathfrak{N}$ if and only if there is a formula $\varphi(x_1, \ldots, x_k)$ in the language of arithmetic such that for every $n_1, \ldots, n_k$, $R(n_1, \ldots, n_k)$ if and only if $\mathfrak{N} \models \varphi(\overline{n}_1, \ldots, \overline{n}_k)$.

Put differently, a relation is definable in in $\mathfrak{N}$ if and only if it is representable in the theory **TA**, where $\mathbf{TA} = \{\varphi : \mathfrak{N} \models \varphi\}$ is the set of true sentences of arithmetic. (If this is not immediately clear to you, you should go back and check the definitions and convince yourself that this is the case.)

Lemma 14.9. *Every computable relation is definable in* $\mathfrak{N}$.

Proof. It is easy to check that the formula representing a relation in **Q** defines the same relation in $\mathfrak{N}$. □

Now one can ask, is the converse also true? That is, is every replation definable in $\mathfrak{N}$ computable? The answer is no. For example:

Lemma 14.10. *The halting relation is definable in* $\mathfrak{N}$.

Proof. Let H be the halting relation, i.e.,

$$H = \{\langle e, x\rangle : \exists s\, T(e, x, s)\}.$$

Let θ_T define T in $\mathfrak{N}$. Then

$$H = \{\langle e, x\rangle : \mathfrak{N} \models \exists s\, \theta_T(\overline{e}, \overline{x}, s)\},$$

so $\exists s\, \theta_T(z, x, s)$ defines H in $\mathfrak{N}$. □

What about **TA** itself? Is it definable in arithmetic? That is: is the set $\{\#(\varphi) : \mathfrak{N} \models \varphi\}$ definable in arithmetic? Tarski's theorem answers this in the negative.

Theorem 14.11. *The set of true statements of arithmetic is not definable in arithmetic.*

Proof. Suppose $\theta(x)$ defined it. By the fixed-point lemma, there is a formula φ such that $\mathbf{Q}$ proves $\varphi \leftrightarrow \neg\theta(\ulcorner\varphi\urcorner)$, and hence $\mathfrak{N} \models \varphi \leftrightarrow \neg\theta(\ulcorner\varphi\urcorner)$. But then $\mathfrak{N} \models \varphi$ if and only if $\mathfrak{N} \models \neg\theta(\ulcorner\varphi\urcorner)$, which contradicts the fact that $\theta(y)$ is supposed to define the set of true statements of arithmetic. □

Tarski applied this analysis to a more general philosophical notion of truth. Given any language L, Tarski argued that an adequate notion of truth for L would have to satisfy, for each sentence X,

> 'X' is true if and only if X.

Tarski's oft-quoted example, for English, is the sentence

> 'Snow is white' is true if and only if snow is white.

However, for any language strong enough to represent the diagonal function, and any linguistic predicate $T(x)$, we can construct a sentence X satisfying "X if and only if not $T(\text{'}X\text{'})$." Given that we do not want a truth predicate to declare some sentences to be both true and false, Tarski concluded that one cannot specify a truth predicate for all sentences in a language without, somehow, stepping outside the bounds of the language. In other words, a the truth predicate for a language cannot be defined in the language itself.

Part V

Sets, Relations, Functions

Chapter 15

Sets

15.1 Basics

Sets are the most fundamental building blocks of mathematical objects. In fact, almost every mathematical object can be seen as a set of some kind. In logic, as in other parts of mathematics, sets and set theoretical talk is ubiquitous. So it will be important to discuss what sets are, and introduce the notations necessary to talk about sets and operations on sets in a standard way.

Definition 15.1. A *set* is a collection of objects, considered independently of the way it is specified, of the order of its elements, or of their multiplicity. The objects making up the set are called *elements* or *members* of the set. If a is an element of a set X, we write $a \in X$ (otherwise, $a \notin X$). The set which has no elements is called the empty set and denoted $\emptyset$.

Example 15.2. Whenever you have a bunch of objects, you can collect them together in a set. The set of Richard's siblings, for instance, is a set that contains one person, and we could write it as $S = \{\text{Ruth}\}$. In general, when we have some objects $a_1, \ldots, a_n$, then the set consisting of exactly those objects is written $\{a_1, \ldots, a_n\}$. Frequently we'll specify a set by some property that its elements share—as we just did, for instance, by specifying S as the set of Richard's siblings. We'll use the following shorthand notation for that: $\{x : \ldots x \ldots\}$, where the $\ldots x \ldots$ stands for the property that x has to have in order to be counted among the elements of the set. In our example, we could have specified S also as $S = \{x : x \text{ is a sibling of Richard}\}$.

When we say that sets are independent of the way they are specified, we mean that the elements of a set are all that matters. For instance, it so happens that $\{\text{Nicole, Jacob}\}$, $\{x : \text{is a niece or nephew of Richard}\}$ and $\{x : \text{is a child of Ruth}\}$ are three ways of specifying one and the same set.

Saying that sets are considered independently of the order of their elements and their multiplicity is a fancy way of saying that $\{\text{Nicole, Jacob}\}$ and

{Jacob, Nicole} are two ways of specifying the same set; and that {Nicole, Jacob} and {Jacob, Nicole, Nicole} are two ways of specifying the same set.

15.2 Some Important Sets

Example 15.3. Mostly we'll be dealing with sets that have mathematical objects as members. You will remember the various sets of numbers: $\mathbb{N}$ is the set of *natural* numbers $\{0, 1, 2, 3, \dots\}$; $\mathbb{Z}$ the set of *integers* $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$; $\mathbb{Q}$ the set of *rationals* ($\mathbb{Q} = \{z/n : z \in \mathbb{Z}, n \in \mathbb{N}, n \neq 0\}$); and $\mathbb{R}$ the set of *real* numbers. These are all *infinite* sets, that is, they each have infinitely many elements. As it turns out, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ have the same number of elements, while $\mathbb{R}$ has a whole bunch more—$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ are "enumerable and infinite" whereas $\mathbb{R}$ is "non-enumerable".

We'll sometimes also use the set of positive integers $\mathbb{Z}^+ = \{1, 2, 3, \dots\}$ and the set containing just the first two natural numbers $\mathbb{B} = \{0, 1\}$.

Example 15.4 (Strings)**.** Another interesting example is the set A^* of *finite strings* over an alphabet A: any finite sequence of elements of A is a string over A. We include the *empty string* Λ among the strings over A, for every alphabet A. For instance,

$$\mathbb{B}^* = \{\Lambda, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \dots\}.$$

If $x = x_1 \dots x_n \in A^*$is a string consisting of n "letters" from A, then we say *length* of the string is n and write $\mathrm{len}(x) = n$.

Example 15.5 (Infinite sequences)**.** For any set A we may also consider the set A^ω of infinite sequences of elements of A. An infinite sequence $a_1 a_2 a_3 a_4 \dots$ consists of a one-way infinite list of objects, each one of which is an element of A.

15.3 Subsets

Sets are made up of their elements, and every element of a set is a part of that set. But there is also a sense that some of the elements of a set *taken together* are a "part of" that set. For instance, the number 2 is part of the set of integers, but the set of even numbers is also a part of the set of integers. It's important to keep those two senses of being part of a set separate.

Definition 15.6. If every element of a set X is also an element of Y, then we say that X is a *subset* of Y, and write $X \subseteq Y$.

Example 15.7. First of all, every set is a subset of itself, and $\emptyset$ is a subset of every set. The set of even numbers is a subset of the set of natural numbers. Also, $\{a, b\} \subseteq \{a, b, c\}$.

But $\{a, b, e\}$ is not a subset of $\{a, b, c\}$.

Note that a set may contain other sets! In particular, a set may happen to *both* be an element and a subset of another, e.g., $\{0\} \in \{0, \{0\}\}$ and also $\{0\} \subseteq \{0, \{0\}\}$.

Definition 15.8. The set consisting of all subsets of a set X is called the *power set of* X, written $\wp(X)$.

$$\wp(X) = \{x : x \subseteq X\}$$

Example 15.9. What are all the possible subsets of $\{a, b, c\}$? They are: $\emptyset$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, $\{a, b, c\}$. The set of all these subsets is $\wp(\{a, b, c\})$:

$$\wp(\{a, b, c\}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$$

15.4 Unions and Intersections

Definition 15.10. The *union* of two sets X and Y, written $X \cup Y$, is the set of all things which are members of X, Y, or both.

$$X \cup Y = \{x : x \in X \vee x \in Y\}$$

Example 15.11. Since the multiplicity of elements doesn't matter, the union of two sets which have an element in common contains that element only once, e.g., $\{a, b, c\} \cup \{a, 0, 1\} = \{a, b, c, 0, 1\}$.

The union of a set and one of its subsets is just the bigger set: $\{a, b, c\} \cup \{a\} = \{a, b, c\}$.

The union of a set with the empty set is identical to the set: $\{a, b, c\} \cup \emptyset = \{a, b, c\}$.

Definition 15.12. The *intersection* of two sets X and Y, written $X \cap Y$, is the set of all things which are elements of both X and Y.

$$X \cap Y = \{x : x \in X \wedge x \in Y\}$$

Two sets are called *disjoint* if their intersection is empty. This means they have no elements in common.

Example 15.13. If two sets have no elements in common, their intersection is empty: $\{a, b, c\} \cap \{0, 1\} = \emptyset$.

If two sets do have elements in common, their intersection is the set of all those: $\{a, b, c\} \cap \{a, b, d\} = \{a, b\}$.

The intersection of a set with one of its subsets is just the smaller set: $\{a, b, c\} \cap \{a, b\} = \{a, b\}$.

The intersection of any set with the empty set is empty: $\{a, b, c\} \cap \emptyset = \emptyset$.

We can also form the union or intersection of more than two sets. An elegant way of dealing with this in general is the following: suppose you collect all the sets you want to form the union (or intersection) of into a single set. Then we can define the union of all our original sets as the set of all objects which belong to at least one element of the set, and the intersection as the set of all objects which belong to every element of the set.

Definition 15.14. If C is a set of sets, then $\bigcup C$ is the set of elements of elements of C:

$$\bigcup C = \{x : x \text{ belongs to an element of } C\}, \text{ i.e.,}$$
$$\bigcup C = \{x : \text{there is a } y \in C \text{ so that } x \in y\}$$

Definition 15.15. If C is a set of sets, then $\bigcap C$ is the set of objects which all elements of C have in common:

$$\bigcap C = \{x : x \text{ belongs to every element of } C\}, \text{i.e.,} \bigcap C = \{x : \text{for all } y \in C, x \in y\}$$

Example 15.16. Suppose $C = \{\{a, b\}, \{a, d, e\}, \{a, d\}\}$. Then $\bigcup C = \{a, b, d, e\}$ and $\bigcap C = \{a\}$.

We could also do the same for a sequence of sets $A_1, A_2, \ldots$

$$\bigcup_i A_i = \{x : x \text{ belongs to one of the } A_i\}$$
$$\bigcap_i A_i = \{x : x \text{ belongs to every } A_i\}.$$

Definition 15.17. The *difference* $X \setminus Y$ is the set of all elements of X which are not also elements of Y, i.e.,

$$X \setminus Y = \{x : x \in X \text{ and } x \notin Y\}.$$

15.5 Proofs about Sets

Sets and the notations we've introduced so far provide us with convenient shorthands for specifying sets and expressing relationships between them. Often it will also be necessary to prove claims about such relationships. If you're not familiar with mathematical proofs, this may be new to you. So we'll walk through a simple example. We'll prove that for any sets X and Y, it's always the case that $X \cap (X \cup Y) = X$. How do you prove an identity between sets like this? Recall that sets are determined solely by their elements, i.e., sets are identical iff they have the same elements. So in this case we have to prove that (a) every element of $X \cap (X \cup Y)$ is also an element of X and, conversely, that (b) every element of X is also an element of $X \cap (X \cup Y)$. In other words, we show that both (a) $X \cap (X \cup Y) \subseteq X$ and (b) $X \subseteq X \cap (X \cup Y)$.

A proof of a general claim like "every element z of $X \cap (X \cup Y)$ is also an element of X" is proved by first assuming that an arbitrary $z \in X \cap (X \cup Y)$ is given, and proving from this assumtion that $z \in X$. You may know this pattern as "general conditional proof." In this proof we'll also have to make use of the definitions involved in the assumption and conclusion, e.g., in this case of "$\cap$" and "$\cup$." So case (a) would be argued as follows:

> (a) We first want to show that $X \cap (X \cup Y) \subseteq X$, i.e., by definition of $\subseteq$, that if $z \in X \cap (X \cup Y)$ then $z \in X$, for any z. So assume that $z \in X \cap (X \cup Y)$. Since z is an element of the intersection of two sets iff it is an element of both sets, we can conclude that $z \in X$ and also $z \in X \cup Y$. In particular, $z \in X$. But this is what we wanted to show.

This completes the first half of the proof. Note that in the last step we used the fact that if a conjunction ($z \in X$ and $z \in X \cup Y$) follows from an assumption, each conjunct follows from that same assumption. You may know this rule as "conjunction elimination," or $\wedge$Elim. Now let's prove (b):

> (b) We now prove that $X \subseteq X \cap (X \cup Y)$, i.e., by definition of $\subseteq$, that if $z \in X$ then also $z \in X \cap (X \cup Y)$, for any z. Assume $z \in X$. To show that $z \in X \cap (X \cup Y)$, we have to show (by definition of "$\cap$") that (i) $z \in X$ and also (ii) $z \in X \cup Y$. Here (i) is just our assumption, so there is nothing further to prove. For (ii), recall that z is an element of a union of sets iff it is an element of at least one of those sets. Since $z \in X$, and $X \cup Y$ is the union of X and Y, this is the case here. So $z \in X \cup Y$. We've shown both (i) $z \in X$ and (ii) $z \in X \cup Y$, hence, by definition of "$\cap$," $z \in X \cap (X \cup Y)$.

This was somewhat long-winded, but it illustrates how we reason about sets and their relationships. We usually aren't this explicit; in particular, we might not repeat all the definitions. A "textbook" proof of our result would look something like this.

Proposition 15.18 (Absorption)**.** *For all sets X, Y,*

$$X \cap (X \cup Y) = X$$

Proof. (a) Suppose $z \in X \cap (X \cup Y)$. Then $z \in X$, so $X \cap (X \cup Y) \subseteq X$.

(b) Now suppose $z \in X$. Then also $z \in X \cup Y$, and therefore also $z \in X \cap (X \cup Y)$. Thus, $X \subseteq X \cap (X \cup Y)$. □

15.6 Pairs, Tuples, Cartesian Products

Sets have no order to their elements. We just think of them as an unordered collection. So if we want to represent order, we use *ordered pairs* $\langle x, y \rangle$, or more generally, *ordered n-tuples* $\langle x_1, \ldots, x_n \rangle$.

Definition 15.19. Given sets X and Y, their *Cartesian product* $X \times Y$ is $\{\langle x, y \rangle : x \in X \text{ and } y \in Y\}$.

Example 15.20. If $X = \{0, 1\}$, and $Y = \{1, a, b\}$, then their product is

$$X \times Y = \{\langle 0, 1 \rangle, \langle 0, a \rangle, \langle 0, b \rangle, \langle 1, 1 \rangle, \langle 1, a \rangle, \langle 1, b \rangle\}.$$

Example 15.21. If X is a set, the product of X with itself, $X \times X$, is also written X^2. It is the set of *all* pairs $\langle x, y \rangle$ with $x, y \in X$.

Chapter 16

Relations

16.1 Relations as Sets

You will no doubt remember some interesting relations between objects of some of the sets we've mentioned. For instance, numbers come with an *order relation* $<$ and from the theory of whole numbers the relation of *divisibility without remainder* (usually written $n \mid m$) may be familar. There is also the relation *is identical with* that every object bears to itself and to no other thing. But there are many more interesting relations that we'll encounter, and even more possible relations. Before we review them, we'll just point out that we can look at relations as a special sort of set. For this, first recall what a *pair* is: if a and b are two objects, we can combine them into the *ordered pair* $\langle a, b\rangle$. Note that for ordered pairs the order *does* matter, e.g, $\langle a, b\rangle \neq \langle b, a\rangle$, in contrast to unordered pairs, i.e., 2-element sets, where $\{a, b\} = \{b, a\}$.

If X and Y are sets, then the *Cartesian product* $X \times Y$ of X and Y is the set of all pairs $\langle a, b\rangle$ with $a \in X$ and $b \in Y$. In particular, $X^2 = X \times X$ is the set of all pairs from X.

Now consider a relation on a set, e.g., the $<$-relation on the set $\mathbb{N}$ of natural numbers, and consider the set of all pairs of numbers $\langle n, m\rangle$ where $n < m$, i.e.,

$$R = \{\langle n, m\rangle : n, m \in \mathbb{N} \text{ and } n < m\}.$$

Then there is a close connection between the number n being less than a number m and the corresponding pair $\langle n, m\rangle$ being a member of R, namely, $n < m$ if and only if $\langle n, m\rangle \in R$. In a sense we can consider the set R to *be* the $<$-relation on the set $\mathbb{N}$. In the same way we can construct a subset of $\mathbb{N}^2$ for any relation between numbers. Conversely, given any set of pairs of numbers $S \subseteq \mathbb{N}^2$, there is a corresponding relation between numbers, namely, the relationship n bears to m if and only if $\langle n, m\rangle \in S$. This justifies the following definition:

Definition 16.1. A *binary relation* on a set X is a subset of X^2. If $R \subseteq X^2$ is a binary relation on X and $x, y \in X$, we write Rxy (or xRy) for $\langle x, y \rangle \in R$.

Example 16.2. The set $\mathbb{N}^2$ of pairs of natural numbers can be listed in a 2-dimensional matrix like this:

$$\begin{array}{ccccc}
\langle \mathbf{0},\mathbf{0} \rangle & \langle 0,1 \rangle & \langle 0,2 \rangle & \langle 0,3 \rangle & \dots \\
\langle 1,0 \rangle & \langle \mathbf{1},\mathbf{1} \rangle & \langle 1,2 \rangle & \langle 1,3 \rangle & \dots \\
\langle 2,0 \rangle & \langle 2,1 \rangle & \langle \mathbf{2},\mathbf{2} \rangle & \langle 2,3 \rangle & \dots \\
\langle 3,0 \rangle & \langle 3,1 \rangle & \langle 3,2 \rangle & \langle \mathbf{3},\mathbf{3} \rangle & \dots \\
\vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

The subset consisting of the pairs lying on the diagonal, $\{\langle 0,0 \rangle, \langle 1,1 \rangle, \langle 2,2 \rangle, \dots\}$, is the *identity relation on* $\mathbb{N}$. (Since the identity relation is popular, let's define $\mathrm{Id}_X = \{\langle x, x \rangle : x \in X\}$ for any set X.) The subset of all pairs lying above the diagonal, $L = \{\langle 0,1 \rangle, \langle 0,2 \rangle, \dots, \langle 1,2 \rangle, \langle 1,3 \rangle, \dots, \langle 2,3 \rangle, \langle 2,4 \rangle, \dots\}$ is the *less than* relation, i.e., Lnm iff $n < m$. The subset of pairs below the diagonal, $G = \{\langle 1,0 \rangle, \langle 2,0 \rangle, \langle 2,1 \rangle, \langle 3,0 \rangle, \langle 3,1 \rangle, \langle 3,2 \rangle, \dots\}$ is the *greater than* relation, i.e., Gnm iff $n > m$. The union of L with I, $K = L \cup I$, is the *less than or equal to* relation: Knm iff $n \leq m$. Similarly, $H = G \cup I$ is the *greater than or equal to relation*. L, G, K, and H are special kinds of relations called *orders*. L and G have the property that no number bears L or G to itself (i.e., for all n, neither Lnn nor Gnn). Relations with this property are called *antireflexive*, and, if they also happen to be orders, they are called *strict orders*.

Although orders and identity are important and natural relations, it should be emphasized that according to our definition *any* subset of X^2 is a relation on X, regardless of how unnatural or contrived it seems. In particular, $\emptyset$ is a relation on any set (the *empty relation*, which no pair of elements bears), and X^2 itself is a relation on X as well (one which every pair bears). But also something like $E = \{\langle n, m \rangle : n > 5 \text{ or } m \times n \geq 34\}$ counts as a relation.

16.2 Special Properties of Relations

Some kinds of relations turn out to be so common that they have been given special names. For instance, $\leq$ and $\subseteq$ both relate their respective domains (say, $\mathbb{N}$ in the case of $\leq$ and $\wp(X)$ in the case of $\subseteq$) in similar ways. To get at exactly how these relations are similar, and how they differ, we categorize them according to some special properties that relations can have. It turns out that (combinations of) some of these special properties are especially important: orders and equivalence relations.

Definition 16.3. A relation $R \subseteq X^2$ is *reflexive* iff, for every $x \in X$, Rxx.

Definition 16.4. A relation $R \subseteq X^2$ is *transitive* iff, whenever Rxy and Ryz, then also Rxz.

Definition 16.5. A relation $R \subseteq X^2$ is *symmetric* iff, whenever Rxy, then also Ryx.

Definition 16.6. A relation $R \subseteq X^2$ is *anti-symmetric* iff, whenever both Rxy and Ryx, then $x = y$ (or, in other words: if $x \neq y$ then either $\neg Rxy$ or $\neg Ryx$).

In a symmetric relation, Rxy and Ryx always hold together, or neither holds. In an anti-symmetric relation, the only way for Rxy and Ryx to hold together is if $x = y$. Note that this does not *require* that Rxy and Ryx holds when $x = y$, only that it isn't ruled out. So an anti-symmetric relation can be reflexive, but it is not the case that every anti-symmetric relation is reflexive. Also note that being anti-symmetric and merely not being symmetric are different conditions. In fact, a relation can be both symmetric and anti-symmetric at the same time (e.g., the identity relation is).

Definition 16.7. A relation $R \subseteq X^2$ is *connected* if for all $x, y \in X$, if $x \neq y$, then either Rxy or Ryx.

Definition 16.8. A relation $R \subseteq X^2$ that is reflexive, transitive, and anti-symmetric is called a *partial order*. A partial order that is also connected is called a *linear order*.

Definition 16.9. A relation $R \subseteq X^2$ that is reflexive, symmetric, and transitive is called an *equivalence relation*.

16.3 Orders

Definition 16.10. A relation which is both reflexive and transitive is called a *preorder.* A preorder which is also anti-symmetric is called a *partial order*. A partial order which is also connected is called a *total order* or *linear order.* (If we want to emphasize that the order is reflexive, we add the adjective "weak"—see below).

Example 16.11. Every linear order is also a partial order, and every partial order is also a preorder, but the converses don't hold. For instance, the identity relation and the full relation on X are preorders, but they are not partial orders, because they are not anti-symmetric (if X has more than one element). For a somewhat less silly example, consider the *no longer than* relation $\preccurlyeq$ on $\mathbb{B}^*$: $x \preccurlyeq y$ iff $\text{len}(x) \leq \text{len}(y)$. This is a preorder, even a linear preorder, but not a partial order.

The relation of *divisibility without remainder* gives us an example of a partial order which isn't a linear order: for integers n, m, we say n (evenly) divides m, in symbols: $n \mid m$, if there is some k so that $m = kn$. On $\mathbb{N}$, this is a partial order, but not a linear order: for instance, $2 \nmid 3$ and also $3 \nmid 2$. Considered as a relation on $\mathbb{Z}$, divisibility is only a preorder since anti-symmetry fails: $1 \mid -1$

and $-1 \mid 1$ but $1 \neq -1$. Another important partial order is the relation $\subseteq$ on a set of sets.

Notice that the examples L and G from Example 16.2, although we said there that they were called "strict orders" are not linear orders even though they are connected (they are not reflexive). But there is a close connection, as we will see momentarily.

Definition 16.12. A relation R on X is called *irreflexive* if, for all $x \in X$, $\neg Rxx$. R is called *asymmetric* if for no pair $x, y \in X$ we have Rxy and Ryx. A *strict partial order* is a relation which is irreflexive, asymmetric, and transitive. A strict partial order which is also connected is called a *strict linear order.*

A strict partial order R on X can be turned into a weak partial order $R'$by adding the identity relation on X: $R' = R \cup \mathrm{Id}_X$. Conversely, starting from a weak partial order, one can get a strict partial order by removing Id_X, i.e., $R' = R \setminus \mathrm{Id}_X$.

Proposition 16.13. *R is a strict partial (linear) order on X iff R' is a weak partial (linear) order. Moreover, Rxy iff $R'xy$ for all $x \neq y$.*

Example 16.14. $\leq$ is the weak linear order corresponding to the strict linear order $<$. $\subseteq$ is the weak partial order corresponding to the strict partial order $\subsetneq$.

16.4 Operations on Relations

It is often useful to modify or combine relations. We've already used the union of relations above (which is just the union of two relations considered as sets of pairs). Here are some other ways:

Definition 16.15. Let $R, S \subseteq X^2$ be relations and Y a set.

1. The *inverse* R^{-1} of R is $R^{-1} = \{\langle y, x\rangle : \langle x, y\rangle \in R\}$.
2. The *relative product* $R \mid S$ of R and S is
$$(R \mid S) = \{\langle x, z\rangle : \text{for some } y, Rxy \text{ and } Syz\}$$
3. The *restriction* $R \restriction Y$ of R to Y is $R \cap Y^2$
4. The *application* $R[Y]$ of R to Y is
$$R[Y] = \{y : \text{for some } x \in X, Rxy\}$$

Example 16.16. Let $S \subseteq \mathbb{Z}^2$ be the successor relation on $\mathbb{Z}$, i.e., the set of pairs $\langle x, y\rangle$ where $x + 1 = y$, for $x, y \in \mathbb{Z}$. Sxy holds iff y is the successor of x.

1. The inverse S^{-1} of S is the predecessor relation, i.e., $S^{-1}xy$ iff $x - 1 = y$.
2. The relative product $S \mid S$ is the relation x bears to y if $x + 2 = y$.
3. The restriction of S to $\mathbb{N}$ is the successor relation on $\mathbb{N}$.
4. The application of S to a set, e.g., $S[\{1, 2, 3\}]$ is $\{2, 3, 4\}$.

Definition 16.17. The *transitive closure* R^+ of a relation $R \subseteq X^2$ is $R^+ = \bigcup_{i=1}^{\infty} R^i$ where $R^1 = R$ and $R^{i+1} = R^i \mid R$.

The *reflexive transitive closure* of R is $R^* = R^+ \cup I_X$.

Example 16.18. Take the successor relation $S \subseteq \mathbb{Z}^2$. S^2xy iff $x + 2 = y$, S^3xy iff $x + 3 = y$, etc. So R^*xy iff for some $i \geq 1$, $x + i = y$. In other words, S^+xy iff $x < y$ (and R^*xy iff $x \leq y$).

Chapter 17

Functions

17.1 Basics

A function is a relation in which each object is related to a unique partner. Many functions are familiar to us from basic arithmetic. For instance, addition and multiplication are functions. They take in two numbers and return a third. A function, more generally, is something that takes one or more things as input and returns some kind of output. A function is a *black box*: what matters is only what output is paired with what input, not the method for calculating the output.

Definition 17.1. A *function* $f\colon X \to Y$ is a mapping of each element of X to an element of Y. We call X the *domain* of f and Y the *codomain* of f. The *range* of f is the subset of the codomain that is actually output by f for some input.

Example 17.2. Multiplication goes from $\mathbb{N} \times \mathbb{N}$ (the domain) to $\mathbb{N}$ (the codomain). As it turns out, the range is also $\mathbb{N}$, since every $n \in \mathbb{N}$ is $n \times 1$.

Multiplication is a function because it pairs each input—each pair of natural numbers—with a single output: $\times : \mathbb{N}^2 \to \mathbb{N}$. In contrast, the square root operation applied to the domain $\mathbb{N}$ is not functional, since each positive integer n has two square roots: $\sqrt{n}$ and $-\sqrt{n}$. We can make it functional by only returning the positive square root: $\sqrt{\ } : \mathbb{N} \to \mathbb{R}$. The relation that pairs each student in a class with their final grade is a function—no student can get two different final grades in the same class. The relation that pairs each student in a class with their parents is not a function—generally each student will have at least two parents.

Example 17.3. Let $f\colon \mathbb{N} \to \mathbb{N}$ be defined such that $f(x) = x + 1$. This tells us that f is a function which takes in natural numbers and outputs natural numbers. It then tells us that, given a natural number, f will output its successor.

In this case, the codomain $\mathbb{N}$ is not the range of f, since the natural number 0 is not the successor of any natural number. The range of f is the set of all positive integers, $\mathbb{Z}^+$.

Example 17.4. Let $g\colon \mathbb{N} \to \mathbb{N}$ be defined such that $g(x) = x - 1 + 2$. This tells us that g is a function which takes in natural numbers and outputs natural numbers. It then tells us that, given a natural number, g will output the successor of the successor of its predecessor. Despite their different definitions, g and f are the same function.

Functions f and g defined above are the same because for any natural number x, $x - 1 + 2 = x + 1$. f and g pair each natural number with the same output. Functions, just like relations more generally, can be treated as just sets of pairs. The definitions for f and g specify the same set by means of different equations, and as we know, sets are independent of how they are specified.

Example 17.5. We can also define functions by cases. For instance, we could define $f\colon \mathbb{N} \to \mathbb{N}$ by

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ \frac{x+1}{2} & \text{if } x \text{ is odd.} \end{cases}$$

This is fine, since every natural number is either even or odd, and the output of this function will always be a natural number. Just remember that if you define a function by cases, every possible input must fall into exactly one case.

17.2 Kinds of Functions

Definition 17.6. A function $f\colon X \to Y$ is *surjective* iff Y is also the range of f, i.e., for every $y \in Y$ there is at least one $x \in X$ such that $f(x) = y$.

If you want to show that a function is surjective, then you need to show that every object in the codomain is the output of the function given some input or other.

Definition 17.7. A function $f\colon X \to Y$ is *injective* iff for each $y \in Y$ there is at most one $x \in X$ such that $f(x) = y$.

Any function pairs each input with a unique output. An injective function has a unique input for each possible output. If you want to show that a function f is injective, you need to show that for any element y of the codomain, if $f(x) = y$ and $f(w) = y$, then $x = w$.

A function which is neither injective, nor surjective, is the constant function $f\colon \mathbb{N} \to \mathbb{N}$ where $f(x) = 1$.

A function which is both injective and surjective is the identity function $f\colon \mathbb{N} \to \mathbb{N}$ where $f(x) = x$.

The successor function $f\colon \mathbb{N} \to \mathbb{N}$ where $f(x) = x + 1$ is injective, but not surjective.

The function

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ \frac{x+1}{2} & \text{if } x \text{ is odd.} \end{cases}$$

is surjective, but not injective.

Definition 17.8. A function $f\colon X \to Y$ is *bijective* iff it is both surjective and injective. We call such a function a *bijection* from X to Y (or between X and Y).

17.3 Operations on Functions

We've already seen that the inverse f^{-1} of a bijective function f is itself a function. It is also possible to compose functions.

Definition 17.9. Let $f : X \to Y$ and $g : Y \to Z$. Then we *compose* g and f to form the function $(g \circ f)\colon X \to Z$, where $(g \circ f)(x) = g(f(x))$.

The function $(g \circ f)\colon X \to Z$ pairs each member of X with a member of Z. We specify which member of Z a member of X is paired with as follows—given an input $x \in X$, first apply the function f to x, which will output some $y \in Y$. Then apply the function g to y, which will output some $z \in Z$.

Example 17.10. Consider the functions $f(x) = x + 1$, and $g(x) = 2x$. What function do you get when you compose these two? $(g \circ f)(x) = g(f(x))$. So that means for every natural number you give this function, you first add one, and then you multiply the result by two. So their composition is $(g \circ f)(x) = 2(x + 1)$.

17.4 Isomorphism

An *isomorphism* is a bijection that preserves the structure of the sets it relates, where structure is a matter of the relationships that obtain between the members of the sets. Consider the following two sets $X = \{1, 2, 3\}$ and $Y = \{4, 5, 6\}$. These sets are both structured by the relations successor, less than and greater than. An isomorphism between the two sets is a bijection that preserves those structures. So a function $f\colon X \to Y$ is an isomorphism if, among other things, $i < j$ iff $f(i) < f(j)$, and j is the successor of i iff $f(j)$ is the successor of $f(i)$.

Definition 17.11. Let U be the pair $\langle X, R\rangle$ and V be the pair $\langle Y, S\rangle$ such that X and Y are sets and R and S are relations on X and Y respectively. A bijection

f from X to Y is an *isomorphism* from U to V iff it preserves the relational structure, that is, for any x and u in X, $\langle x, u\rangle \in R$ iff $\langle f(x), f(u)\rangle \in S$.

Example 17.12. Consider the following two sets $X = \{1, 2, 3\}$ and $Y = \{4, 5, 6\}$, and the relations successor, less than, and greater than. The function $f\colon X \to Y$ where $f(x) = x + 3$ is an isomorphism between X and Y.

17.5 Partial Functions

It is sometimes useful to relax the definition of function so that it is not required that the output of the function is defined for all possible inputs. Such mappings are called *partial functions*.

Definition 17.13. A *partial function* $f\colon X \nrightarrow Y$ is a mapping which assigns to every element of X at most one element of Y. If f assigns an element of Y to $x \in X$, we say $f(x)$ is *defined*, and otherwise undefined. If $f(x)$ is defined, we write $f(x)\downarrow$, otherwise $f(x)\uparrow$. The *domain* of a partial function f is the subset of X where it is defined, i.e., $\mathrm{dom}(f) = \{x : f(x)\downarrow\}$.

Example 17.14. Every function $f\colon X \to Y$ is also a partial function. Partial functions that are defined everywhere on X are called *total*.

Example 17.15. The partial function $f\colon \mathbb{R} \nrightarrow \mathbb{R}$ given by $f(x) = 1/x$ is undefined for $x = 0$, and defined everywhere else.

Chapter 18

The Size of Sets

18.1 Introduction

When Georg Cantor developed set theory in the 1870s, his interest was in part to make palatable the idea of an infinite collection—an actual infinity, as the medievals would say. Key to this rehabilitation of the notion of the infinite was a way to assign sizes—"cardinalities"—to sets. The cardinality of a finite set is just a natural number, e.g., $\emptyset$ has cardinality 0, and a set containing five things has cardinality 5. But what about infinite sets? Do they all have the same cardinality, ∞? It turns out, they do not.

The first important idea here is that of an enumeration. We can list every finite set by listing all its elements. For some infinite sets, we can also list all their elements if we allow the list itself to be infinite. Such sets are called enumerable. Cantor's surprising result was that some infinite sets are not enumerable.

18.2 Enumerable Sets

Definition 18.1. Informally, an *enumeration* of a set X is a list (possibly infinite) such that every element of X appears some finite number of places into the list. If X has an enumeration, then X is said to be *enumerable*. If X is enumerable and infinite, we say X is denumerable.

A couple of points about enumerations:

1. The order of elements of X in the enumeration does not matter, as long as every element appears: 4, 1, 25, 16, 9 enumerates the (set of the) first five square numbers just as well as 1, 4, 9, 16, 25 does.

2. Redundant enumerations are still enumerations: 1, 1, 2, 2, 3, 3, ... enumerates the same set as 1, 2, 3, ... does.

3. Order and redundancy *do* matter when we specify an enumeration: we can enumerate the natural numbers beginning with 1, 2, 3, 1, ..., but the pattern is easier to see when enumerated in the standard way as 1, 2, 3, 4, ...

4. Enumerations must have a beginning: ..., 3, 2, 1 is not an enumeration of the natural numbers because it has no first element. To see how this follows from the informal definition, ask yourself, "at what place in the list does the number 76 appear?"

5. The following is not an enumeration of the natural numbers: 1, 3, 5, ..., 2, 4, 6, ... The problem is that the even numbers occur at places $\infty + 1$, $\infty + 2$, $\infty + 3$, rather than at finite positions.

6. Lists may be gappy: 2, −, 4, −, 6, −, ... enumerates the even natural numbers.

7. The empty set is enumerable: it is enumerated by the empty list!

The following provides a more formal definition of an enumeration:

Definition 18.2. An *enumeration* of a set X is any surjective function $f : \mathbb{N} \to X$.

Let's convince ourselves that the formal definition and the informal definition using a possibly gappy, possibly infinite list are equivalent. A surjective function (partial or total) from $\mathbb{N}$ to a set X enumerates X. Such a function determines an enumeration as defined informally above. Then an enumeration for X is the list $f(0)$, $f(1)$, $f(2)$, ... Since f is surjective, every element of X is guaranteed to be the value of $f(n)$ for some $n \in \mathbb{N}$. Hence, every element of X appears at some finite place in the list. Since the function may be partial or not injective, the list may be gappy or redundant, but that is acceptable (as noted above). On the other hand, given a list that enumerates all elements of X, we can define a a surjective function $f\colon \mathbb{N} \to X$ by letting $f(n)$ be the $(n+1)$st member of the list, or undefined if the list has a gap in the $(n+1)$st spot.

Example 18.3. A function enumerating the natural numbers ($\mathbb{N}$) is simply the identity function given by $f(n) = n$.

Example 18.4. The functions $f\colon \mathbb{N} \to \mathbb{N}$ and $g\colon \mathbb{N} \to \mathbb{N}$ given by

$$f(n) = 2n \text{ and} \tag{18.1}$$
$$g(n) = 2n + 1 \tag{18.2}$$

enumerate the even natural numbers and the odd natural numbers, respectively. However, neither function is an enumeration of $\mathbb{N}$, since neither is surjective.

Example 18.5. The function $f(n) = \lceil \frac{(-1)^n n}{2} \rceil$ (where $\lceil x \rceil$ denotes the *ceiling* function, which rounds x up to the nearest integer) enumerates the set of integers $\mathbb{Z}$. Notice how f generates the values of $\mathbb{Z}$ by "hopping" back and forth between positive and negative integers:

$$\begin{array}{cccccccc} f(1) & f(2) & f(3) & f(4) & f(5) & f(6) & \dots \\ \lceil -\frac{1}{2} \rceil & \lceil \frac{2}{2} \rceil & \lceil -\frac{3}{2} \rceil & \lceil \frac{4}{2} \rceil & \lceil -\frac{5}{2} \rceil & \lceil \frac{6}{2} \rceil & \dots \\ 0 & 1 & -1 & 2 & -2 & 3 & \dots \end{array}$$

That is fine for "easy" sets. What about the set of, say, pairs of natural numbers?

$$\mathbb{N}^2 = \mathbb{N} \times \mathbb{N} = \{\langle n, m \rangle : n, m \in \mathbb{N}\}$$

Another method we can use to enumerate sets is to organize them in an *array*, such as the following:

	1	**2**	**3**	**4**	...
1	$\langle 1,1 \rangle$	$\langle 1,2 \rangle$	$\langle 1,3 \rangle$	$\langle 1,4 \rangle$	...
2	$\langle 2,1 \rangle$	$\langle 2,2 \rangle$	$\langle 2,3 \rangle$	$\langle 2,4 \rangle$	...
3	$\langle 3,1 \rangle$	$\langle 3,2 \rangle$	$\langle 3,3 \rangle$	$\langle 3,4 \rangle$	...
4	$\langle 4,1 \rangle$	$\langle 4,2 \rangle$	$\langle 4,3 \rangle$	$\langle 4,4 \rangle$	...
⋮	⋮	⋮	⋮	⋮	⋱

Clearly, every ordered pair in $\mathbb{N}^2$ will appear at least once in the array. In particular, $\langle n, m \rangle$ will appear in the nth column and mth row. But how do we organize the elements of an array into a list? The pattern in the array below demonstrates one way to do this:

	1	2	4	7	...
	3	5	8	...	...
	6	9	...	...	...
	10	...	...	...	...
	⋮	⋮	⋮	⋮	⋱

This pattern is called *Cantor's zig-zag method*. Other patterns are perfectly permissible, as long as they "zig-zag" through every cell of the array. By Cantor's zig-zag method, the enumeration for $\mathbb{N}^2$ according to this scheme would be:

$$\langle 1,1 \rangle, \langle 1,2 \rangle, \langle 2,1 \rangle, \langle 1,3 \rangle, \langle 2,2 \rangle, \langle 3,1 \rangle, \langle 1,4 \rangle, \langle 2,3 \rangle, \langle 3,2 \rangle, \langle 4,1 \rangle, \dots$$

What ought we do about enumerating, say, the set of ordered triples of natural numbers?

$$\mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N} = \{(n, m, k) : n, m, k \in \mathbb{N}\}$$

We can think of $\mathbb{N}^3$ as the Cartesian product of $\mathbb{N}^2$ and $\mathbb{N}$, that is,

$$\mathbb{N}^3 = \mathbb{N}^2 \times \mathbb{N} = \{(\vec{a}, k) : \vec{a} \in \mathbb{N}^2, k \in \mathbb{N}\}$$

and thus we can enumerate $\mathbb{N}^3$ with an array by labelling one axis with the enumeration of $\mathbb{N}$, and the other axis with the enumeration of $\mathbb{N}^2$:

	1	**2**	**3**	**4**	...
$\langle \mathbf{1}, \mathbf{1} \rangle$	$\langle 1,1,1 \rangle$	$\langle 1,1,2 \rangle$	$\langle 1,1,3 \rangle$	$\langle 1,1,4 \rangle$	...
$\langle \mathbf{1}, \mathbf{2} \rangle$	$\langle 1,2,1 \rangle$	$\langle 1,2,2 \rangle$	$\langle 1,2,3 \rangle$	$\langle 1,2,4 \rangle$	...
$\langle \mathbf{2}, \mathbf{1} \rangle$	$\langle 2,1,1 \rangle$	$\langle 2,1,2 \rangle$	$\langle 2,1,3 \rangle$	$\langle 2,1,4 \rangle$	...
$\langle \mathbf{1}, \mathbf{3} \rangle$	$\langle 1,3,1 \rangle$	$\langle 1,3,2 \rangle$	$\langle 1,3,3 \rangle$	$\langle 1,3,4 \rangle$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Thus, by using a method like Cantor's zig-zag method, we may similarly obtain an enumeration of $\mathbb{N}^3$.

18.3 Non-enumerable Sets

Some sets, such as the set $\mathbb{N}$ of natural numbers, are infinite. So far we've seen examples of infinite sets which were all enumerable. However, there are also infinite sets which do not have this property. Such sets are called *non-enumerable*.

Cantor's method of diagonalization shows a set to be non-enumerable via a reductio proof. We start with the assumption that the set is enumerable, and show that a contradiction results from this assumption. Our first example is the set $\mathbb{B}^\omega$ of all infinite, non-gappy sequences of 0's and 1's.

Theorem 18.6. *$\mathbb{B}^\omega$ is non-enumerable.*

Proof. Suppose, for reductio, that $\mathbb{B}^\omega$ is enumerable, so that there is a list s_1, s_2, s_3, s_4, ... of all the elements of $\mathbb{B}^\omega$. We may arrange this list, and the elements of each sequence s_i in it vertically in an array with the positive integers on the horizontal axis, as so:

1	2	3	4	...
$s_1(1)$	$s_1(2)$	$s_1(3)$	$s_1(4)$	...
$s_2(1)$	$s_2(2)$	$s_2(3)$	$s_2(4)$	...
$s_3(1)$	$s_3(2)$	$s_3(3)$	$s_3(4)$	...
$s_4(1)$	$s_4(2)$	$s_4(3)$	$s_4(4)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Here $s_1(1)$ is a name for whatever number, a 0 or a 1, is the first member in the sequence s_1, and so on.

Now define $\overline{s}$ as follows: The nth member $\overline{s}(n)$ of the sequence $\overline{s}$ is set to

$$\overline{s}(n) = \begin{cases} 1 & \text{if } s_n(n) = 0 \\ 0 & \text{if } s_n(n) = 1. \end{cases}$$

In other words, $\overline{s}(n)$ has the opposite value to $s_n(n)$.

Clearly $\overline{s}$ is a non-gappy infinite sequence of 0s and 1s, since it is just the mirror sequence to the sequence of 0s and 1s that appear on the diagonal of our array. So $\overline{s}$ is an element of $\mathbb{B}^\omega$. Since it is an element of $\mathbb{B}^\omega$, it must appear somewhere in the enumeration of $\mathbb{B}^\omega$, that is, $\overline{s} = s_n$ for some n.

If $\overline{s} = s_n$, then for any m, $\overline{s}(m) = s_n(m)$. (This is just the criterion of identity for sequences—sequences are identical when they agree at every place.)

So in particular, $\overline{s}(n) = s_n(n)$. $\overline{s}(n)$ must be either an 0 or a 1. If it is a 0 then (given the definition of $\overline{s}$) $s_n(n)$ must be a 1. But if it is a 1 then $s_n(n)$ must be a 0. In either case $\overline{s}(n) \neq s_n(n)$. □

Diagonalization need not involve the presence of an array, though the array method is where it takes its name.

Theorem 18.7. *$\wp(\mathbb{Z}^+)$ is not enumerable.*

Proof. Suppose, for reductio, that $\wp(\mathbb{Z}^+)$ is enumerable, and so it has an enumeration, i.e., a list of all subsets of $\mathbb{Z}^+$:

$$Z_1, Z_2, Z_3, \ldots$$

We now define a set $\overline{Z}$ such that for any positive integer i, $i \in \overline{Z}$ iff $i \notin Z_i$:

$$\overline{Z} = \{i \in \mathbb{Z}^+ : i \notin Z_i\}$$

$\overline{Z}$ is clearly a set of positive integers, and thus $\overline{Z} \in \wp(\mathbb{Z}^+)$. So $\overline{Z}$ must be $= Z_k$ for some $k \in \mathbb{Z}^+$. And if that is the case, i.e., $\overline{Z} = Z_k$, then $i \in \overline{Z}$ iff $i \in Z_k$ for all $i \in \mathbb{Z}^+$.

In particular, $k \in \overline{Z}$ iff $k \in Z_k$.

Now either $k \in Z_k$ or $k \notin Z_k$. In the first case, by the previous line, $k \in \overline{Z}$. But we've defined $\overline{Z}$ so that it contains exactly those $i \in \mathbb{Z}^+$ which are *not* elements of Z_i. So by that definition, we would have to also have $k \notin Z_k$. In the second case, $k \notin Z_k$. But now k satisfies the condition by which we have defined $\overline{Z}$, and that means that $k \in \overline{Z}$. And as $\overline{Z} = Z_k$, we get that $k \in Z_k$ after all. Either case leads to a contradiction. □

18.4 Reduction

We showed $\wp(\mathbb{Z}^+)$ to be non-enumerable by a diagonalization argument. However, with the proof of the non-enumerability of $\mathbb{B}^\omega$, the set of all infinite sequences of 0s and 1s, in place, we could have instead showed $\wp(\mathbb{Z}^+)$

to be non-enumerable by showing that *if* $\wp(\mathbb{Z}^+)$ *is enumerable then* $\mathbb{B}^\omega$ *is also enumerable*. This called *reducing* one problem to another.

Proof of Theorem 18.7 by reduction. Suppose, for reductio, $\wp(\mathbb{Z}^+)$ is enumerable, and thus that there is an enumeration of it $Z_1, Z_2, Z_3, \ldots$

Define the function $f\colon \wp(\mathbb{Z}^+) \to \mathbb{B}^\omega$ by letting $f(Z)$ be the sequence s_k such that $s_k(j) = 1$ iff $j \in Z$.

Every sequence of 0s and 1s corresponds to some set of positive integers, namely the one which has as its members those integers corresponding to the places where the sequence has 1s. In other words, this is a surjective function.

Now consider the list

$$f(Z_1), f(Z_2), f(Z_3), \ldots$$

Since f is surjective, every member of $\mathbb{B}^\omega$ must appear as a value of f for some argument, and so must appear on the list. So this list must enumerate $\mathbb{B}^\omega$.

So if $\wp(\mathbb{Z}^+)$ were enumerable, $\mathbb{B}^\omega$ would be enumerable. But $\mathbb{B}^\omega$ is non-enumerable (Theorem 18.6). □

18.5 Equinumerous Sets

We have an intuitive notion of "size" of sets, which works fine for finite sets. But what about infinite sets? If we want to come up with a formal way of comparing the sizes of two sets of *any* size, it is a good idea to start with defining when sets are the same size. Let's say sets of the same size are *equinumerous*. We want the formal notion of equinumerosity to correspond with our intuitive notion of "same size," hence the formal notion ought to satisfy the following properties:

Reflexivity: Every set is equinumerous with itself.

Symmetry: For any sets X and Y, if X is equinumerous with Y, then Y is equinumerous with X.

Transitivity: For any sets X, Y, and Z, if X is equinumerous with Y and Y is equinumerous with Z, then X is equinumerous with Z.

In other words, we want equinumerosity to be an *equivalence relation*.

Definition 18.8. A set X is *equinumerous* with a set Y if and only if there is a total bijection f from X to Y (that is, $f\colon X \to Y$).

Proposition 18.9. *Equinumerosity defines an equivalence relation.*

Proof. Let X, Y, and Z be sets.

Reflexivity: Using the identity map $1_X \colon X \to X$, where $1_X(x) = x$ for all $x \in X$, we see that X is equinumerous with itself (clearly, 1_X is bijective).

Symmetry: Suppose that X is equinumerous with Y. Then there is a bijection $f \colon X \to Y$. Since f is bijective, its inverse f^{-1} is also a bijection. Since f is surjective, f^{-1} is total. Hence, $f^{-1} \colon Y \to X$ is a total bijection from Y to X, so Y is also equinumerous with X.

Transitivity: Suppose that X is equinumerous with Y via the total bijection f and that Y is equinumerous with Z via the total bijection g. Then the composition of $g \circ f \colon X \to Z$ is a total bijection, and X is thus equinumerous with Z.

Therefore, equinumerosity is an equivalence relation by the given definition. □

Theorem 18.10. *Suppose X and Y are equinumerous. Then X is enumerable if and only if Y is.*

Proof. Let X and Y be equinumerous. Suppose that X is enumerable. Then there is a possibly partial, surjective function $f \colon \mathbb{N} \to X$. Since X and Y are equinumerous, there is a total bijection $g \colon X \to Y$. Claim: $g \circ f \colon \mathbb{N} \to Y$ is surjective. Clearly, $g \circ f$ is a function (since functions are closed under composition). To see $g \circ f$ is surjective, let $y \in Y$. Since g is surjective, there is an $x \in X$ such that $g(x) = y$. Since f is surjective, there is an $n \in \mathbb{N}$ such that $f(n) = x$. Hence,

$$(g \circ f)(n) = g(f(n)) = g(x) = y$$

and thus $g \circ f$ is surjective. Since $g \circ f \colon \mathbb{N} \to Y$ is surjective, it is an enumeration of Y, and so Y is enumerable. □

18.6 Comparing Sizes of Sets

Just like we were able to make precise when two sets have the same size in a way that also accounts for the size of infinite sets, we can also compare the sizes of sets in a precise way. Our definition of "is smaller than (or equinumerous)" will require, instead of a bijection between the sets, a total injective function from the first set to the second. If such a function exists, the size of the first set is less than or equal to the size of the second. Intuitively, an injective function from one set to another guarantees that the range of the function has at least as many elements as the domain, since no two elements of the domain map to the same element of the range.

Definition 18.11. $|X| \leq |Y|$ if and only if there is an injective function $f \colon X \to Y$.

Theorem 18.12 (Schröder-Bernstein)**.** *Let X and Y be sets. If $|X| \leq |Y|$ and $|Y| \leq |X|$, then $|X| = |Y|$.*

In other words, if there is a total injective function from X to Y, and if there is a total injective function from Y back to X, then there is a total bijection from X to Y. Sometimes, it can be difficult to think of a bijection between two equinumerous sets, so the Schröder-Bernstein theorem allows us to break the comparison down into cases so we only have to think of an injection from the first to the second, and vice-versa. The Schröder-Bernstein theorem, apart from being convenient, justifies the act of discussing the "sizes" of sets, for it tells us that set cardinalities have the familiar anti-symmetric property that numbers have.

Definition 18.13. $|X| < |Y|$ if and only if there is an injective function $f\colon X \to Y$ but no bijective $g\colon X \to Y$.

Theorem 18.14 (Cantor)**.** *For all X, $|X| < |\wp(X)|$.*

Proof. The function $f\colon X \to \wp(X)$ that maps any $x \in X$ to its singleton $\{x\}$ is injective, since if $x \neq y$ then also $f(x) = \{x\} \neq \{y\} = f(y)$.

There cannot be a surjective function $g\colon X \to \wp(X)$, let alone a bijective one. For assume that a surjective $g\colon X \to \wp(X)$ exists. Then let $Y = \{x \in X : x \notin g(x)\}$. If $g(x)$ is defined for all $x \in X$, then Y is clearly a well-defined subset of X. If g is surjective, Y must be the value of g for some $x_0 \in X$, i.e., $Y = g(x_0)$. Now consider x_0: it cannot be an element of Y, since if $x_0 \in Y$ then $x_0 \in g(x_0)$, and the definition of Y then would have $x_0 \notin Y$. On the other hand, it must be an element of Y, since if it were not, then $x_0 \notin Y = g(x_0)$. But then x_0 satisfies the defining condition of Y, and so $x_0 \in Y$. In either case, we have a contradiction. □