Introduction to Logic and Computability

Ichiro Hasuo

National Institute of Informatics, Tokyo, Japan

http://group-mmm.org/~ichiro
i.hasuo[at]acm.org

Version: April 20, 2022
Draft—your comments are appreciated!

Contents

0.4pt 0pt

0.4pt 0.5pt 0pt 0pt 0.4pt 0.0pt

Preface

Notice: The current lecture notes were originally meant for the use in the course Information Logic *at Department of Information Science, Faculty of Science, the University of Tokyo. The author was in charge of the course from 2011 to 2016. There are a few examples and references in Japanese, but they do not play essential roles for the lecture notes.*

Logic and theory of computability are so much fundamental topics in computer science that we are never short of good textbooks (see below for some pointers). The reason I am writing the current lecture notes nevertheless is the limited number of lectures given to these topics in our curriculum. In particular, theory of computability taught in a course can either:

- lack intuitions, especially when conciseness is a priority and the presentation is mathematically oriented; or
- take too much time, when all the details are provided so that you get the feeling of a (computing) "machine" working.

In the current lecture notes I try to pick the best of the two approaches and make the most of the limited number of lectures available for theory of computability.

Another goal of the current notes is to familiarize you with theoretical literature. "Theoretical" here means "mathematical"; and mathematics books and papers require the reader to follow certain *rules*. These rules are something you must learn through training; but once you acquire them, you have much easier access to the contents of the vast body of mathematical knowledge. Well, in short: I'd like you to learn *how to learn mathematics by yourself*.

References

Some are listed in the main text.

Overall I strongly recommend reading [17], before, in the middle of, or after reading this textbook. It is an informal introduction to logic, theory of computation and foundation of mathematics, with a lot of intuitions and histrical remarks, presented in an inspiring and intuitive manner. At times when mathematical details in this textbook overwhelm you, the book [17] can save you.

Logic

- The current notes are loosely based on [15], which I can certainly recommend.
- If you are an aspiring student: [6] is a classic textbook that is used often for reading groups at logic-oriented research groups.

  > *Studying this book and doing the exercises made me a logician.*
  > (L. van den Dries, U. Illinois)

- The one [20] by Hagiya-sensei and Nishizaki-sensei is like a dictionary, covering many related (and advanced) topics. It might take time to read it from cover to cover; recommended as a reference.
- Other recent references include [10, 18, 9].

Computability

- The current notes are somewhere in-between [11] and [13]. The former is concise but dry—it might be hard for you to get intuitions; the latter is detailed, with a lot of examples and informal arguments supporting the technical results.
- [2] is a modern comprehensive introduction in English.
- [20] includes a standard treatment of computability using Turing machines as a basic infrastructure (which we will not do in this course).
- On Gödel's incompleteness theorem (Chap. 9), [12] provides a modern, accessible, yet comprehensive overview.

Acknowledgment

Kentaro Honda has helped (and have been helping) me a great deal as a teaching assistant and as a critical proof reader. The use of equational logic as a showcase was suggested by Yde Venema, which I am grateful for. Kenshi Miyabe suggested the proof of Prop. 8.4.6.

Most of the current lecture notes were written during the author's employment at Department of Computer Science, the University of Tokyo, for the course *Information Logic*. After my departure from the department, the lecture notes were used by Masami Hagiya and Hitomi Yanaka in the same course. Their feedbacks, together with the feedbacks from the brilliant, motivated, and critical students there, are gratefully acknowledged.

Part I

Logic

Chapter 1

Set Theory Primer

The material covered here is what is called *naive set theory*—as opposed to *axiomatic set theory* that is built upon formal logic (which is what we will learn in this course).

How to use this chapter This chapter is mostly for fixing notations; it is far from complete, detailed or reader-friendly. It assumes that you have learned about the material somewhere before. Your first reading can be a quick glance; come back later when necessary. In particular: different names can be given to the same notion; you need *not* remember all these names!

References For more detailed exposition, in fact Wikipedia (in English) will do most of the time; but [16] is a classic textbook (maybe there are more modern ones that are good too). If you are interested in posets, [1] is recommended as an introduction.

1.1 Basic Constructions on Sets

1.1.1 Definition. Two sets X, Y are *equal* (we write $X = Y$) if they contain exactly the same elements.

They are *isomorphic* ($X \cong Y$; it is also said: *in a bijective correspondence*; or to *have the same cardinality*) if there is a *bijective* function $f : X \xrightarrow{\cong} Y$, that is,

f is *injective*, i.e.	$f(x) = f(x') \implies x = x'$	; and
f is *surjective*, i.e.	for any $y \in Y$ there exists $x \in X$ such that $f(x) = y$.	

Note: usually in a definition, 'if and only if' is written as 'if'; thus the 'if's in the above in fact mean 'if and only if.'

1.1.2 Example.

$$\{\text{apple}, \text{apple}, \text{orange}\} = \{\text{apple}, \text{orange}\} \neq \{\text{dog}, \text{cat}\}$$

Here all the sets are isomorphic (their cardinality is 2).

1.1.3 Definition. The *emptyset* is the set that contains no elements (i.e. $\{\}$). It is denoted by $\emptyset$.

Note that two emptysets are necessarily the same, thus we say *the* emptyset. Also note: in LaTeX, $\emptyset$ is `\emptyset` and not `\phi`.

1.1.4 Definition.

The *join* of two sets:	$X \cup Y = \{z \mid z \in X \text{ or } z \in Y\}$
The *meet* of two sets:	$X \cap Y = \{z \mid z \in X \text{ and } z \in Y\}$

More generally, given a family of sets $(X_i)_{i\in I}$ indexed by an index set I,

$$\bigcup_{i\in I} X_i = \{z \mid z \in X_i \text{ for some } i \in I\} \ ; \qquad \bigcap_{i\in I} X_i = \{z \mid z \in X_i \text{ for all } i \in I\}$$

where for the definition of $\bigcap_{i\in I} X_i$, the index set I must be nonempty.

Note that $(X_i)_{i\in I}$ is (subtly) different from the *set* of sets $\{X_i\}_{i\in I}$ (what if $X_i = X_j$ for $i \neq j \in I$?).

1.1.5 Definition (Cartesian product $X\times Y$, $\prod_{i\in I} X_i$)**.** The *(Cartesian) product* of two sets X and Y is the collection of *ordered pairs*

$$X \times Y = \{(x, y) \mid x \in X, y \in Y\} \ .$$

Similarly defined:

$$X_1 \times \cdots \times X_n = \{(x_1, \ldots, x_n) \mid x_i \in X_i\} \ ;$$
$$X^n = \{(x_1, \ldots, x_n) \mid x_i \in X\} \ ;$$

More generally, given a family of sets $(X_i)_{i\in I}$ indexed by an index set I,

$$\prod_{i\in I} X_i = \{(x_i)_{i\in I} \mid x_i \in X_i\}$$

where $(x_i)_{i\in I}$ is a choice of an element $x_i \in X_i$ for each $i \in I$.

1.1.6 Remark. The 0-ary product X^0 contains exactly one element $()$, that is, $X^0 = \{()\}$. (Why? Consider $|X^n| = |X|^n$) Therefore $\emptyset^0$ is not empty.

1.1.7 Definition (Disjoint union $X \amalg Y$, $\coprod_{i\in I} X_i$)**.** The *disjoint union* (also called *direct sum* or *coproduct*) of two sets is

$$X \amalg Y = (\{0\} \times X) \cup (\{1\} \times Y) \ .$$

Notice that $\{0\} \times X$ is isomorphic to X; 0 and 1 here are "markers" that distinguish elements of X and those of Y (which may overlap). The set $X \amalg Y$ is also denoted by $X + Y$.

Similarly defined:

$$X_1 \amalg \cdots \amalg X_n \ = \ (\{1\} \times X_1) \cup \cdots \cup (\{n\} \times X_n) \ .$$

More generally, given a family of sets $(X_i)_{i\in I}$ indexed by an index set I,

$$\coprod_{i\in I} X_i = \bigcup_{i\in I} \left(\{i\} \times X_i\right) \ .$$

1.1.8 Remark. The sets $\prod_{i\in I} X_i$ and $\coprod_{i\in I} X_i$ are defined for any index set I, however big I is (it can be uncountable).

1.1.9 Definition (Subset $X \subseteq Y$, powerset $\mathcal{P}(X)$)**.** A set X is said to be a *subset* of a set Y if

$$x \in X \quad \Longrightarrow \quad x \in Y \ .$$

We denote this fact by $X \subseteq Y$.

The *powerset* $\mathcal{P}(X)$ of a set X is the collection of all subsets of X, that is,

$$\mathcal{P}(X) = \{S \mid S \subseteq X\} \ .$$

Note that $\mathcal{P}(X)$ always contains $\emptyset$, and X itself.

1.2 Function

You know the notion of *function* $f : X \to Y$—it is a correspondence which, given $x \in X$, returns $f(x) \in Y$. We formulate this notion via *binary relations*.

1.2.1 Definition (Binary relation)**.** A *binary relation* R between X and Y is a subset $R \subseteq X \times Y$. We write xRy when $(x, y) \in R$.

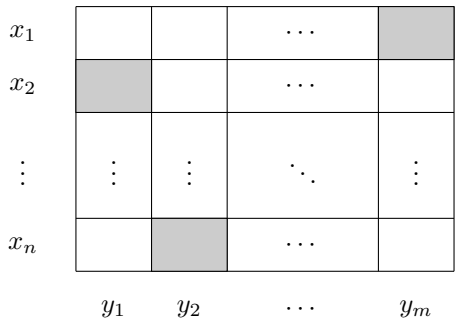


Figure 1.1: Binary relation (in case X, Y are finite)

1.2.2 Definition (Function)**.** A *function* $f : X \to Y$ is a binary relation $f \subseteq X \times Y$ such that: for each $x \in X$, there exists *exactly one* $y \in Y$ such that $(x, y) \in f$. We denote such a unique y by $f(x)$; we also write this fact $f : x \mapsto y$.

Distinguish two arrows ($f : X \to Y$ vs. $f : x \mapsto f(x)$).

From this definition via relations, we see clearly how many functions there are if

- the domain is $\emptyset$, or
- the range is $\emptyset$.

See Exercise 1.2.

1.2.3 Definition (Domain, range)**.** For a function $f : X \to Y$, the set X is called the *domain* of f; and Y is called the *range* (or *codomain*) of f.

1.2.4 Definition (Function space Y^X)**.** The collection of all functions from X to Y is called the *function space* and is denoted by Y^X. That is,

$$Y^X = \{f : X \to Y\} \ .$$

It is also denoted by: $X \to Y$ or $X \Rightarrow Y$.

1.2.5 Definition (Finite set n)**.** Let n be a natural number (note: in computer science or logic, 0 is often considered as a natural number). We denote the n-element set $\{0, 1, 2, \ldots, n-1\}$ also by n. That is,

$$n = \{0, 1, 2, \ldots, n-1\} \ .$$

The following definitions have already appeared.

1.2.6 Definition (Injection, surjection, bijection)**.** A function $f : X \to Y$ is

- *injective* (written $f : X \hookrightarrow Y$ or $X \rightarrowtail Y$) if it maps different elements to different elements, that is,

$$f(x) = f(x') \quad \Longrightarrow \quad x = x' \ ;$$

- *surjective* (written $f : X \twoheadrightarrow Y$) if it "covers its whole range," that is,

$$\text{for any } y \in Y \text{ there exists } x \in X \text{ such that } f(x) = y;$$

- *bijective* (written $f : X \xRightarrow{\cong} Y$) if it is both injective and surjective.

Recall two sets X, Y are said to be *isomorphic* if there is a bijection between them; this means X and Y are "essentially the same."

1.2.7 Definition (Composition $g \circ f$; identity id_X)**.** Given two successive functions $f : X \to Y$ and $g : Y \to Z$ (notice the matching domain and range), the function

$$g \circ f \ : \ X \longrightarrow Z$$

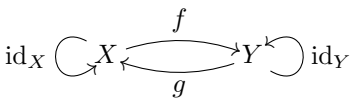
is defined by

$$g \circ f \ : \ x \longmapsto g(f(x)) \ .$$

The *identity function* on a set X is the function

$$\mathrm{id}_X \ : \ X \longrightarrow X \ , \qquad x \longmapsto x \ .$$

1.2.8 Lemma. *A function $f : X \to Y$ is a bijection if and only if there exists a function $g : Y \to X$ such that: $g \circ f = \mathrm{id}_X$ and $f \circ g = \mathrm{id}_Y$.*



□

A *partial function* is like a function but can be undefined in part of its domain.

1.2.9 Definition (Partial function $X \rightharpoonup Y$)**.** A *partial function* $f : X \rightharpoonup Y$ is a function $f : S \to Y$ from some subset $S \subseteq X$ to Y.

$$\begin{array}{ccc} X & & \\ \subseteq\!\uparrow & & \\ S & \xrightarrow{\quad f \quad} & Y \end{array}$$

1.2.10 Lemma. *A partial function $f : X \rightharpoonup Y$ is the same thing as a function*

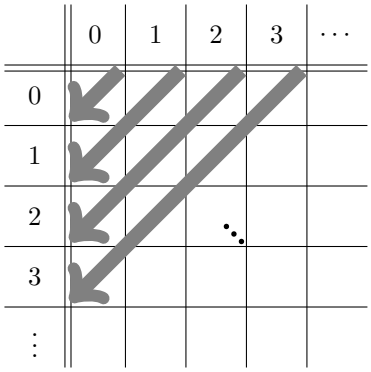
$$f \;:\; X \longrightarrow Y \amalg \{\perp\}$$

where $\perp$ is a fresh symbol meaning "undefined." More precisely: there is a canonical bijection between

$$\{\textit{partial functions } f : X \rightharpoonup Y\} \quad \textit{and} \quad \textit{the function space } \big(Y \amalg \{\perp\}\big)^X \;. \qquad \square$$

1.2.11 Proposition. *The sets $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$ are isomorphic.*

Proof.



$\square$

1.2.12 Proposition. *The sets $Z^{X \times Y}$ and $(Z^Y)^X$ are isomorphic.* $\square$

The last result says: a function $f : X \times Y \to Z$ is "the same thing" as a function $f^\wedge : X \to (Y \Rightarrow Z)$, that is,

$$\frac{f : X \times Y \to Z}{f^\wedge : X \to (Y \Rightarrow Z)} \;.$$

This is the principle of *currying* in functional programming.

1.2.13 Definition (Characteristic function χ_S)**.** Let $S \subseteq X$. The *characteristic function* $\chi_S : X \to 2$ is defined by

$$\chi_S \;:\; x \longmapsto \begin{cases} 0 & \text{if } x \in S \\ 1 & \text{if } x \notin S \end{cases}$$

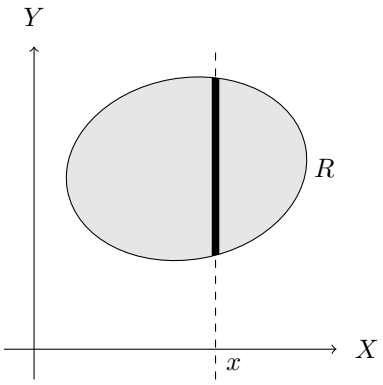
Recall $2 = \{0, 1\}$ (Def. 1.2.5); here 0 means "true" and 1 means "false." Characteristic functions are used in:

1.2.14 Proposition. *The sets $\mathcal{P}(X)$ and 2^X are isomorphic.* $\square$

Moreover:

1.2.15 Proposition. *The set* $\{$*binary relations* $R \subseteq X \times Y\}$ *is isomorphic to the function space* $\big(\mathcal{P}(Y)\big)^X$.

Proof.



Also see Exercise 1.8. □

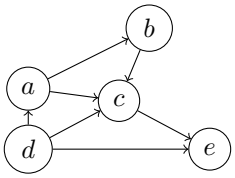
1.2.16 Proposition. *Assume* $X_i = X$ *for each* $i \in \mathbb{N}$. *Then the product* $\prod_{i\in\mathbb{N}} X_i$ *is isomorphic to* $X^{\mathbb{N}}$. □

1.3 Binary Relation

1.3.1 Example. A directed graph can be thought of as a pair

$$(V, E)$$

of the set V of *vertices* and a binary relation $E \subseteq V \times V$ that represents *edges*—xEy if there is an edge from x to y.



1.3.2 Definition (Relational composition $S \circ R$)**.** Let $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ be two binary relations (with matching (co)domains). The *(relational) composition*

$$S \circ R \;\subseteq\; X \times Z$$

is defined by

$$(x, z) \in S \circ R \quad \overset{\text{def}}{\iff} \quad \text{there exists } y \in Y \text{ such that } xRy \text{ and } ySz.$$

Note that the order $S \circ R$ is the same as in $g \circ f$ (Def. 1.2.7).

1.3.3 Definition (R^n, R^0)**.** Let $R \subseteq X \times X$ be a binary relation. The relation R^n is defined in the obvious way:

$$R^n := R \circ R \circ \cdots \circ R \ ,$$

for $n \geq 1$. For $n = 0$, we define R^0 to be the *diagonal relation*:

$$R^0 := \Delta_X = \{(x, x) \mid x \in X\} \ . \tag{1.1}$$

See Exercise 1.11 for the justification of the definition of R^0.

1.3.4 Definition ($*$-closure)**.** Let $R \subseteq X \times X$ be a binary relation (note that R is between the same set X). The $*$-*closure* of R, denoted by R^*, is defined by

$$R^* \ := \bigcup_{n \in \mathbb{N}} R^n \ .$$

That is,

$$xR^*y \iff \text{there are } x_0, x_1, \ldots, x_n \in X \text{ such that } x = x_0 \, R \, x_1 \, R \, \ldots \, R \, x_n = y \ .$$

As a "closure" operator (like $U \mapsto \overline{U}$ in a topological space), the operator $(\_)^*$ is required to satisfy the following properties.

1.3.5 Proposition. *Let $R, S \subseteq X \times X$ be two binary relations.*

1. $R \subseteq R^*$.
2. *$R \subseteq S$ implies $R^* \subseteq S^*$.*
3. $(R^*)^* = R^*$. □

1.3.1 Equivalence Relation

1.3.6 Definition (Reflexivity; symmetry; transitivity)**.** A binary relation $R \subseteq X \times X$ is said to be

reflexive if:	xRx for each $x \in X$;
symmetric if:	for each $x, y \in X$, xRy implies yRx;
transitive if:	for each $x, y, z \in X$, xRy and yRz implies xRz.

One can *never* underestimate the important roles of *notations* and *formalisms* in mathematics: proper *understanding*[1] is so often brought about by getting used to a proper formalism.

In this course (and elsewhere) we fancy *rule-based presentation*. In that style of presentation, Def. 1.3.6 is written as follows.

$$\frac{}{xRx} \text{ (Refl.)} \qquad \frac{yRx}{xRy} \text{ (Sym.)} \qquad \frac{xRy \quad yRz}{xRz} \text{ (Trans.)}$$

[1] It is another important issue what is exactly to *understand* mathematics. In my opinion it is not at all taking notes and memorizing them; it is rather the ability to *use* the mathematical notions and reasoning principles.

These "rules" written like fractions are read from top to bottom: if the "assumptions" on the top all hold, then the "conclusion" on the bottom holds. Two lines = means implication in both directions (if and only if): for example, it is easy to see

$$\frac{yRx}{\overline{xRy}}\ \text{(Sym.)}\ .$$

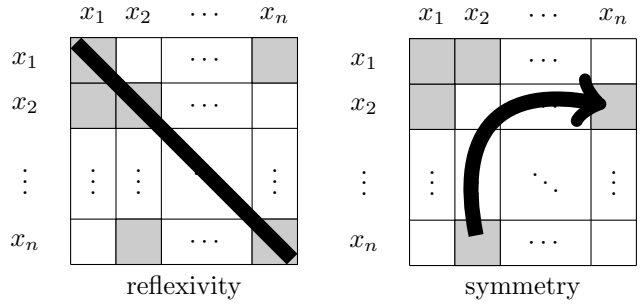


Figure 1.2: Reflexivity, symmetry, transitivity

1.3.7 Definition (Equivalence relation)**.** A relation $R \subseteq X \times X$ is an *equivalence relation* if it is reflexive, symmetric and transitive.

1.3.8 Definition (Equivalence class)**.** Let R be an equivalence relation on X, and $x \in X$. The *R-equivalence class* of x, denoted by $[x]_R$, is the following subset of X.

$$[x]_R := \{x' \in X \mid xRx'\}$$

The collection of R-equivalence classes is called the *R-quotient* of X and is denoted by X/R.

1.3.9 Lemma. *The whole set X is the disjoint union of equivalence classes. That is,*

1. *if two equivalence classes $S, S' \in X/R$ are not disjoint ($S \cap S' \neq \emptyset$), then they are equal ($S = S'$);*
2. *the union $\bigcup X/R := \bigcup_{S \in X/R} S$ coincides with X.* □

1.3.10 Definition (Canonical projection)**.** There is a canonical[2] map

$$\begin{array}{rccc} \pi_R : & X & \longrightarrow & X/R \\ & x & \longmapsto & [x]_R \end{array}$$

which is surjective; this map is called the *R-quotient map* or the *R-projection*.

An equivalence relation can be thought of as an *abstraction*, or an *attribute*.

1.3.11 Example. Each of the following attributes determines an equivalence relation over the set of you guys.

[2] "Canonical" means: there might be some other "coincidental" ones; but this is a "naturally arising" one. The word has to do with "canon" and hence with Christianism (I guess). Cf. two vector spaces V, W of the same dimension are isomorphic, but in general there is no canonical choice among the possible isomorphisms.

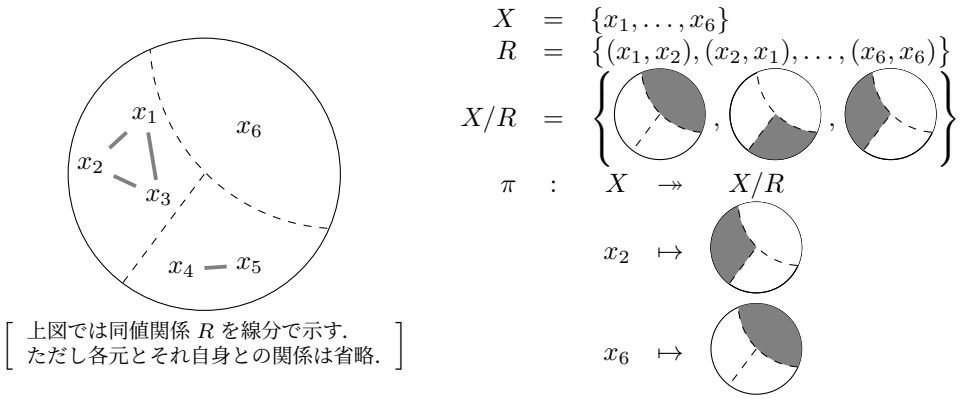


Figure 1.3: Equivalence classes and projection

gender, age, prefecture of birth, city of residence, ...

See also Exercise 1.15.

1.3.12 Definition (Kernel)**.** A function $f : X \to Y$ induces an equivalence relation $\sim_f$ on X, called the *kernel* of f, by:

$$x \sim_f x' \quad \stackrel{\text{def}}{\iff} \quad f(x) = f(x') \ .$$

1.3.13 Definition (Reflexive and transitive closure)**.** Let $R \subseteq X \times X$ be a relation. A relation $S \subseteq X \times X$ is said to be the *reflexive and transitive closure* of R, denoted by R^{rt}, if:

1. $R \subseteq S$;
2. S is reflexive and transitive; and
3. for any reflexive and transitive $T \subseteq X \times X$ such that $R \subseteq T$, we have $S \subseteq T$.

At first sight the previous definition might puzzle you. Another way to put it: S is the "smallest extension of R" that is reflexive and transitive. You add pairs to R (as few as possible) so that you get reflexivity and transitivity.

But is such extension always possible? In this case, yes.

1.3.14 Lemma. *The $*$-closure R^* (Def. 1.3.4) is the reflexive and transitive closure of R.*

Proof. We have to show, for any reflexive and transitive extension T of R: 1) $R \subseteq R^*$ (easy); 2) R^* is reflexive and transitive (easy); 3) if T is a reflexive and transitive extension of R then $R^* \subseteq T$.

3) is proved as follows. Assume $x\, R^*\, y$; then we have a sequence

$$x = x_0\, R\, x_1\, R \ldots R\, x_n = y \ .$$

Since $R \subseteq T$ we have

$$x = x_0 \, T \, x_1 \, T \, \ldots \, T \, x_n = y \ ;$$

by T's transitivity (when $n \geq 2$) or T's reflexivity (when $n = 0$), we have

$$x = x_0 \, T \, x_n = y \ .$$

This concludes the proof. □

In the above lemma we provided an *explicit construction* of R^{rt} and concluded that R^{rt} always exists. Its existence alone can be shown in the following way, too. The proof is a general one and applies to a large number of "closures."

1.3.15 Lemma. *For any relation $R \subseteq X \times X$, its reflexive and transitive closure R^{rt} always exists.*

Proof. Let us denote by $\mathcal{R}$ the family of all reflexive and transitive extensions of R. That is:

$$\mathcal{R} \ := \ \{S \subseteq X \times X \mid R \subseteq S, \text{and } S \text{ is reflexive and transitive.}\}$$

Then it holds that

$$\bigcap \mathcal{R} \ = \ \bigcap_{S \in \mathcal{R}} S$$

also belongs to $\mathcal{R}$. Therefore the intersection $\bigcap \mathcal{R}$ is the smallest reflexive and transitive extension of $\mathcal{R}$, that is, R^{rt}. □

Indeed the following fact can be proved by the same argument.

1.3.16 Lemma. *For any $R \subseteq X \times X$, its* equivalence closure*—the smallest equivalence relation that contains R—always exists.* □

1.3.2 Order

1.3.17 Definition (Antisymmetry)**.** A binary relation $R \subseteq X \times X$ is said to be

$$\textit{antisymmetric} \text{ if:} \qquad\qquad xRy \text{ and } yRx \text{ implies } x = y.$$

That is, as a rule,

$$\frac{xRy \quad yRx}{x = y} \ \text{(AntiSym.)}$$

1.3.18 Definition ((Partial) order; poset; preorder)**.** A binary relation $R \subseteq X \times X$ is a *(partial) order* if it is reflexive, antisymmetric and transitive.

A set X equipped with a partial order $\leq$ is called a *partially ordered set (poset)*. We write $(X, \leq)$.

A relation R is a *preorder* if it is reflexive and transitive.

A preorder can be turned into an order: see Exercise 1.19.

1.3.19 Definition (Total order)**.** An order $\leq$ on X is said to be *total* if

$$x \leq y \quad \text{or} \quad y \leq x \qquad \text{for any } x, y \in X.$$

1.3.20 Example. Let X be a set. Its powerset $\mathcal{P}(X)$ has a natural order $\subseteq$ given by inclusion. It is in general not total. (When is it total?)

1.3.21 Definition (Binary join/meet)**.** Let $(X, \leq)$ be a poset. An element $z \in X$ is said to be the *join* of $x, y \in X$ if

1. $x \leq z$ and $y \leq z$;
2. z is the smallest among such, that is, for any u such that $x \leq u$ and $y \leq u$, we have $z \leq u$.

We write $x \vee y$ for the join of x and y.

The *meet* $x \wedge y$ of $x, y \in X$ is defined in a similar manner.

The previous definitions can be represented as the following rules (recall double lines = means 'if and only if'), which we find very useful.

$$\frac{x \leq u \quad y \leq u}{x \vee y \leq u} \text{ (Join)} \qquad \frac{u \leq x \quad u \leq y}{u \leq x \wedge y} \text{ (Meet)} \tag{1.2}$$

More generally:

1.3.22 Definition (Join, meet)**.** Let $(X, \leq)$ be a poset; and $S \subseteq X$ be a subset of X. The *join* (also called *supremum*) of S, denoted by $\bigvee S$, is the element of X such that

$$\frac{s \leq u \quad \text{for each } s \in S}{\bigvee S \leq u} \text{ (Join)} \ .$$

Similarly, the *meet* $\bigwedge S$ (also called *infimum*) is such that

$$\frac{u \leq s \quad \text{for each } s \in S}{u \leq \bigwedge S} \text{ (Meet)} \ .$$

1.3.23 Lemma. *1. A join $\bigvee S$, if it exists, is unique. The same holds for a meet $\bigwedge S$.*

2. The join $\bigvee \emptyset$ of the emptyset $\emptyset \subseteq X$ is nothing but the smallest element (the minimum*) $\bot_X$ of X. Similarly, $\bigwedge \emptyset$ is the greatest element (the* maximum*) $\top_X$.*

3. The minimum $\bot_X$, if it exists, is the unit for $\vee$. That is, for any $x \in X$

$$\bot_X \vee x = x \ .$$

Similarly, $\top_X \wedge x = x$.

1.3.24 Definition (Lattice; complete lattice)**.** A poset $(X, \leq)$ is said to be a *lattice* if it has

- the minimum $\bot_X$ and the maximum $\top_X$;
- the join $x \vee y$ and the meet $x \wedge y$ for any $x, y \in X$.

A poset $(X, \leq)$ is a *complete lattice* if for any $S \subseteq X$, there are $\bigvee S$ and $\bigwedge S$.

1.3.25 Example. 1. For any set X, its powerset $\mathcal{P}(X)$ is a complete lattice with

$$\bigvee \mathcal{S} = \bigcup \mathcal{S} \ , \quad \bigwedge \mathcal{S} = \bigcap \mathcal{S} \ .$$

2. For any topological space $(X, \mathcal{O}_X)$ (consider $X = \mathbb{R}$ if you are not familiar), the family $\mathcal{O}_X$ of its open sets is a complete lattice, with

$$\bigvee \mathcal{S} = \bigcup \mathcal{S} \ , \quad \bigwedge \mathcal{S} = \left(\bigcap \mathcal{S}\right)^\circ \ ,$$

where $(\_)^\circ$ means taking the interior. Here recall that open sets are closed under arbitrary unions and *finite* intersections.

Exercises

1.1. Describe the elements of the following sets.

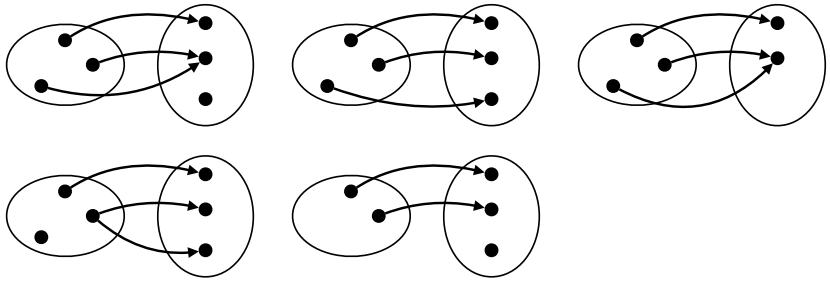
1. $\{1,2\} \times \{3,4\}$
2. $\{1,2\} \times \{\}$
3. $\{1,2\} \times \{\emptyset\}$
4. $\{1,2\} \amalg \{3,4\}$
5. $\{1,2\} \amalg \{2,3\}$
6. $\{1,2\} \times \{0,2\} \times \{0,1\}$

1.2. How many elements do the following sets have?

1. $\{1,2\} \times \{3,4,5\}$
2. $2^{\{3,4,5\}}$
3. $2^{\{3,4,5,6\}}$
4. $\{3,4\}^{\{3,4,5,6\}}$
5. $\{3,4,5\}^{\{1,2\}}$
6. $\{3,4,5\}^2$
7. $\{\}^{\{3,4,5\}}$
8. $\{\emptyset\}^{\{3,4,5\}}$
9. $\{3,4,5\}^{\{\}}$
10. $\{3,4,5\}^{\emptyset}$
11. $\emptyset^{\emptyset}$

1.3. Prove Lem. 1.2.10.

1.4. Are the following correspondences: bijective? injective? surjective? a function? a partial function? a binary relation?



1.5. Fill in the details and prove Prop. 1.2.11.

1.6. Prove Prop. 1.2.12.

1.7. Prove Prop. 1.2.14.

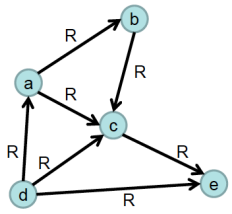
1.8. Prove Prop. 1.2.15 using Prop. 1.2.12 and 1.2.14.

1.9. Prove Prop. 1.2.16.

1.10. For the graph shown in Example 1.3.1, describe V and E.

1.11. Show that the diagonal relation Δ_X (in (1.1)) is the *unit* of relational composition (Def. 1.3.2): $\Delta_Y \circ R = R = R \circ \Delta_X$ for any binary relation $R \subseteq X \times Y$.

1.12. Add edges for the relation $R \circ R$ in the following graph.



What about R^*?

1.13. Prove Prop. 1.3.5.

1.14. Prove Lem. 1.3.9.

1.15. Consider the following relation R over the set of ASCII strings.

$$xRy \quad \stackrel{\text{def}}{\iff} \quad x \text{ and } y, \text{ as C programs, compute the same function.}$$

Is this relation R an equivalence relation?

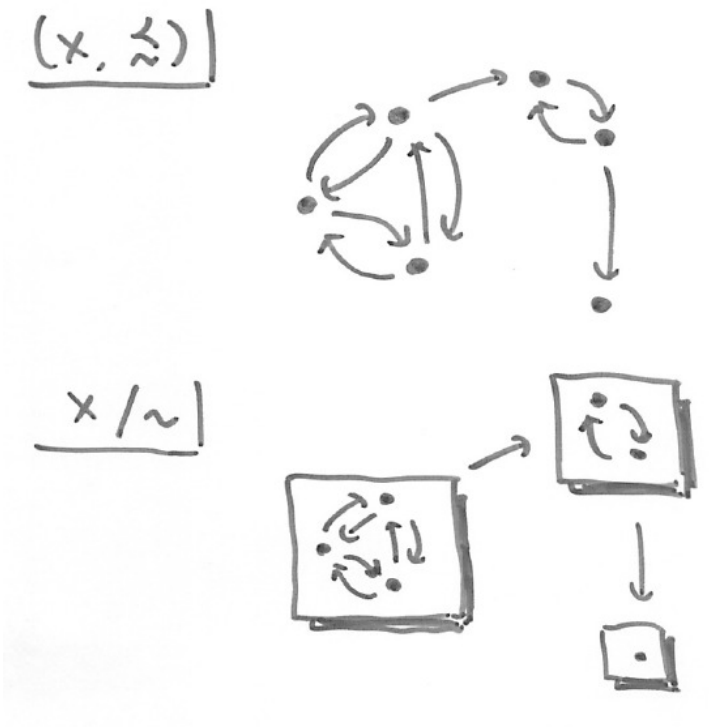
1.16. In Def. 1.3.12, prove that $\sim_f$ is indeed an equivalence relation. Prove also that: $\sim_f$ coincides with the diagonal relation Δ_X (Exercise 1.11) if and only if f is injective.

1.17. In Def. 1.3.13, prove that a reflexive and transitive closure of a given relation R is unique. That is: if S and S' are both a reflexive and transitive closure of R, then $S = S'$.

1.18. Fill in the details and prove Lem. 1.3.15. Caution: can it be that $\mathcal{R} = \emptyset$?

1.19. Let $\lesssim$ be a preorder over a set X (Def. 1.3.18).

1. Show that the relation $\lesssim \cap \gtrsim$ is an equivalence relation, for which we write $\sim$.
2. Show that we have
$$x \sim x' \ , \quad y \sim y' \ , \quad x \lesssim y \quad \Longrightarrow \quad x' \lesssim y' \ .$$
Therefore the relation $\lesssim$ on the set $X/\sim$, defined by
$$[x]_\sim \lesssim [y]_\sim \quad \overset{\text{def}}{\Longleftrightarrow} \quad x \lesssim y$$
is well-defined. We express this fact as: "$\lesssim$ induces a well-defined relation over the quotient set $X/\sim$."
3. Show that the induced relation $\lesssim$ on $X/\sim$ is a partial order.



1.20. The *lexicographic order* is the order of words used in a dictionary. State its precise definition by filling in the blank.

> Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be totally ordered sets. The *lexicographic order* $\leq$ on the set $X \times Y$ is defined by:
> $$(x, y) \leq (x', y') \quad \overset{\text{def}}{\Longleftrightarrow} \quad \boxed{\quad \ldots \quad}$$

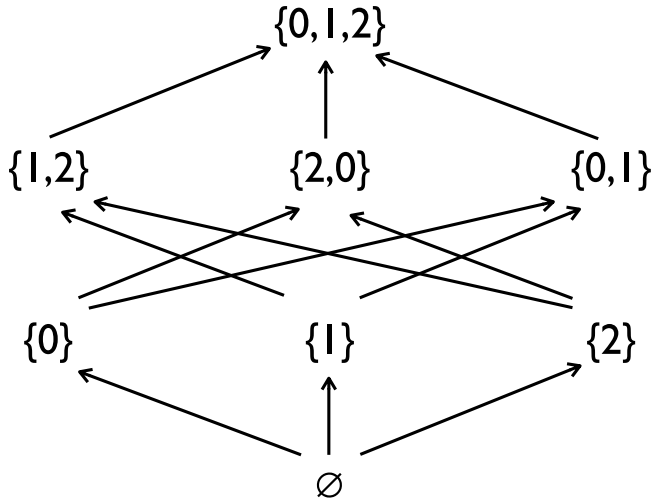
1.21. Provide more examples of a poset which is not total.

1.22. Show that the rules (1.2) are indeed equivalent to the definitions in Def. 1.3.21.

1.23. Prove Lem. 1.3.23.

1.24. Fill in the details of Example 1.3.25.

1.25. Come up with a poset which is *not* a lattice, and express it in a *Hasse diagram*. As a reference, a Hasse diagram for the poset $\mathcal{P}(\{0,1,2\})$ is shown below.



1.26. Let A be a set and T be the the set of all preorders on A. Show that $(T, \subseteq)$ forms a lattice.

1.27. (If you have not taken a course on automata and formal languages, skip this question.)

Let Σ be a nonempty finite alphabet, and $\mathsf{Reg} = \{L \subseteq \Sigma^* \mid L \text{ is regular}\}$ be the class of regular languages over Σ. Show that $(\mathsf{Reg}, \subseteq)$ forms a lattice.

Show also that $(\mathsf{Reg}, \subseteq)$ is *not* a complete lattice.

Chapter 2

Equational Logic as a Showcase

In this course we will learn two basic systems of logic—*propositional logic* and *predicate logic*. Before that we use (even simpler) *equational logic* as the first example and get the feeling of what "(formal) logic" is all about.

2.1 First Examples

2.1.1 Polynomials

You must *know* and *have used* the equality

$$(x+y)^2 = x^2 + 2xy + y^2 \ ; \tag{2.1}$$

you should also be able to *prove* its correctness.

Question: how can a computer decide if (2.1) is true? Can it come up with a proof for that?

One possible way for a computer to do that (and in fact this is probably what you, a human, would do too) consists of the following three steps.

1. **Fix a language.** In the current setting, the following rules determine a language that is sufficient.

$$\frac{\mathbf{x} \text{ is a variable}}{\mathbf{x} \text{ is a term}}\ (\textsc{Var}) \qquad \frac{\mathbf{t} \text{ is a term} \quad \mathbf{t}' \text{ is a term}}{\mathbf{t}\cdot\mathbf{t}' \text{ is a term}}\ (\textsc{Prod})$$
$$\frac{\mathbf{t} \text{ is a term} \quad \mathbf{t}' \text{ is a term}}{\mathbf{t}+\mathbf{t}' \text{ is a term}}\ (\textsc{Sum}) \tag{2.2}$$

   An equivalent definition is given by the following *BNF notation*:

$$\mathbf{t} ::= \mathbf{x} \in \mathbf{Var} \ \mid\ \mathbf{t}\cdot\mathbf{t} \ \mid\ \mathbf{t}+\mathbf{t} \ .$$

2. **Specify axioms.** *Axioms* are equalities that are assumed to hold. In the

current setting, the following set would suffice.

$$\mathbf{t}_1 \cdot (\mathbf{t}_2 + \mathbf{t}_3) = \mathbf{t}_1 \cdot \mathbf{t}_2 + \mathbf{t}_1 \cdot \mathbf{t}_3 \qquad \text{(Distr)}$$
$$\mathbf{t}_1 + \mathbf{t}_2 = \mathbf{t}_2 + \mathbf{t}_1 \qquad \text{(AddComm)}$$
$$\mathbf{t}_1 \cdot \mathbf{t}_2 = \mathbf{t}_2 \cdot \mathbf{t}_1 \qquad \text{(MultComm)}$$
$$\mathbf{t}_1 + (\mathbf{t}_2 + \mathbf{t}_3) = (\mathbf{t}_1 + \mathbf{t}_2) + \mathbf{t}_3 \qquad \text{(AddAssoc)}$$
$$\mathbf{t}_1 \cdot (\mathbf{t}_2 \cdot \mathbf{t}_3) = (\mathbf{t}_1 \cdot \mathbf{t}_2) \cdot \mathbf{t}_3 \qquad \text{(MultAssoc)}$$

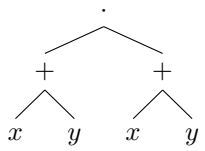
3. **Derive** the equality

$$(x + y) \cdot (x + y) = (((x \cdot x) + (x \cdot y)) + (x \cdot y)) + (y \cdot y) \tag{2.3}$$

(which is the equality (2.1) made precise in the current language), by the following *derivation*.

$$\begin{aligned}
&(x + y) \cdot (x + y) \\
&= ((x + y) \cdot x) + ((x + y) \cdot y) && \text{by (Distr)} \\
&= (x \cdot (x + y)) + (y \cdot (x + y)) && \text{by (MultComm)} \\
&= ((x \cdot x) + (x \cdot y)) + ((y \cdot x) + (y \cdot y)) && \text{by (Distr)} \\
&= ((x \cdot x) + (x \cdot y)) + ((x \cdot y) + (y \cdot y)) && \text{by (MultComm)} \\
&= (((x \cdot x) + (x \cdot y)) + (x \cdot y)) + (y \cdot y) && \text{by (AddAssoc).}
\end{aligned}$$

Note that here (and elsewhere) we identify a sequence of symbols (such as terms) and *abstract syntax trees*: for example, what we write as $(x+y)\cdot(x+y)$ should be interpreted as the following tree.



2.1.2 Group

The equality (2.1) was between polynomials (over a commutative ring). Our second example is the following equality over a group:

$$(xy)^{-1}xy = e \ , \tag{2.4}$$

where e is the unit of a group ($xe = x = ex$).

We shall run the same scenario as in §2.1.1.

1. **Fix a language.**

$$\frac{\mathbf{x} \text{ is a variable}}{\mathbf{x} \text{ is a term}} \ \text{(Var)} \qquad \frac{}{e \text{ is a term}} \ \text{(Unit)}$$
$$\frac{\mathbf{t} \text{ is a term}}{\mathbf{t}^{-1} \text{ is a term}} \ \text{(Inv)} \qquad \frac{\mathbf{t} \text{ is a term} \quad \mathbf{t}' \text{ is a term}}{\mathbf{t} \cdot \mathbf{t}' \text{ is a term}} \ \text{(Prod)}$$

In a BNF notation:

$$\mathbf{t} ::= \mathbf{x} \in \mathbf{Var} \mid e \mid \mathbf{t}^{-1} \mid \mathbf{t} \cdot \mathbf{t} \ .$$

2. **Specify axioms.**

$$\mathbf{t}_1 \cdot (\mathbf{t}_2 \cdot \mathbf{t}_3) = (\mathbf{t}_1 \cdot \mathbf{t}_2) \cdot \mathbf{t}_3 \qquad \text{(Assoc)}$$
$$e \cdot \mathbf{t} = \mathbf{t} = \mathbf{t} \cdot e \qquad \text{(Unit)}$$
$$\mathbf{t}^{-1} \cdot \mathbf{t} = e = \mathbf{t} \cdot \mathbf{t}^{-1} \qquad \text{(Inv)}$$

3. **Derive** the equality

$$((x \cdot y)^{-1} \cdot x) \cdot y = e \qquad (2.5)$$

(which is the equality (2.4) made precise in the current language), by the following *derivation*.

$$\begin{aligned} ((x \cdot y)^{-1} \cdot x) \cdot y &= (x \cdot y)^{-1} \cdot (x \cdot y) && \text{by (Assoc)} \\ &= e && \text{by (Inv)} \end{aligned}$$

2.1.3 *Equational Logic* as a Common Platform

We now extract the essence of the scenario repeated in §2.1.1–2.1.2, and formalize it in a mathematical language. The result is what is called *equational logic*; this is the topic of the current chapter. This chapter provides a rather detailed treatment of equational logic—eliminating ambiguities and describing intuitions—so that it serves as a primer to the following chapters about propositional/predicate logic.

2.1.1 Remark. Equational logic is also called (or rather: is part of the topic called) *universal algebra*. It is "universal" since it is a parametrized theory that instantiates to any kind of algebra: groups, rings, monoids, lattices, etc.[1] These are all instances of the notion of (Σ, E)-*algebra* (defined later), with suitable choices of the parameters Σ and E.

2.1.4 Variables vs. *Meta*-Variables

By the way: have you noticed the distinction between boldface symbols (such as $\mathbf{x}, \mathbf{t}$) and standard-face ones (such as x, y)?

For example, x in (2.3) is a *variable on the object level*; $\mathbf{x}$ in (2.2) is a *metavariable*—variable on the *meta level*—that stands for some variable $x, y, z, x_1, x_2, \ldots$ on the object level. The symbol $\mathbf{t}$ in (2.2) is also a metavariable that stands for some (concrete) term on the object level, such as $(x \cdot y) + x$.

This distinction between *object level symbols* and *meta level ones* is a subtle one, when we try to make it explicit and precise. However you, with programming experience, must have seen it before. Consider the following: it is part of a Java tutorial.

> To read a file (say `foo.txt`) one uses the following code.
>
> ```
> FileInputStream fooFile = new FileInputStream("foo.txt");
> InputStreamReader foo = new InputStreamReader(fooFile);
> ```

[1] Fields are an notable exception since one of its operation $x \mapsto x^{-1}$ is not total (undefined when $x = 0$).

Here the fictitious file name `foo.txt` plays the role of a metavariable[2]: it would not be the actual file you want to read; instead it is a placeholder that is to be replaced with the actual file name (such as `grades.txt`). Another example from a Java tutorial:

> To store some integer value (say `valueToStore`) in the memory one writes:
>
> ```
> int variableName;
> variableName = valueToStore;
> ```

Here `variableName` is a metavariable that is to be replaced with some actual variable like `x`. Thus: `x` is to x (in §2.1.1–2.1.2) what `variableName` is to `x` (in this example).

This distinction between variables and metavariables—or, more generally, distinction between the object level and the meta level—is such an important issue in the study of logic. For example, shortly we will be dealing with (object level) *proofs* that are nothing but certain syntactic constructs (namely a tree of formulas); we will then prove some *metatheorems* that claim e.g. "no proof exists." The "proof" of the latter metatheorem is a proof on the meta level—which is done based on the meta level logic—and must not be confused with an object level (syntactic) proof.

While this distinction is an important one, using different symbols for the two different levels (as we have done, such as x vs. $\mathbf{x}$) is tedious and many textbooks just do not do that.[3] In these lecture notes, only in the current Chap. 2 we shall be careful about this distinction. In the later chapters we will be more sloppy—but you must be always aware of the distinction!

2.2 Equational Logic: Term

In §2.1.1–2.1.2, the first step was to specify which syntactic expressions are legitimate ones. For example, in §2.1.1, $x + (y \cdot \ )$ is not a well-formed term—the second argument of the operation $\cdot$ is missing—leading to a "syntax error."

Recall also that we now aim at a general framework that has two examples in §2.1.1–2.1.2 as two instances. For this purpose we use the following notion as a parameter.

2.2.1 Definition ((Algebraic) signature)**.** An *(algebraic) signature* Σ is a sequence of sets

$$\Sigma \ = \ (\Sigma_n)_{n \in \mathbb{N}} \ .$$

An element $\boldsymbol{\sigma} \in \Sigma_n$ is called an *n-ary operation*.

2.2.2 Definition (The set **Var**)**.** Henceforth we fix a countably infinite set **Var** of *variables*.

An element $\mathbf{x} \in \mathbf{Var}$ is a variable on the object level.

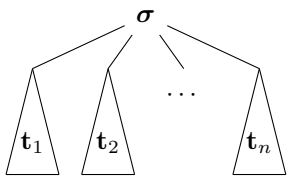
[2] In English common metavariables are `foo`, `bar`, ...; in Japanese they are `hoge`, `fuga`, ...
[3] [6] is a notable exception.

2.2.3 Definition (Σ-term)**.** Let Σ be a signature. The set of Σ-*terms* is defined inductively by the following rules.

$$\frac{\mathbf{x} \text{ is a variable}}{\mathbf{x} \text{ is a } \Sigma\text{-term}} \ (\textsc{Var})$$

$$\frac{\mathbf{t}_1 \text{ is a } \Sigma\text{-term} \quad \mathbf{t}_2 \text{ is a } \Sigma\text{-term} \quad \cdots \quad \mathbf{t}_n \text{ is a } \Sigma\text{-term} \quad \boldsymbol{\sigma} \in \Sigma_n}{\boldsymbol{\sigma}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \text{ is a } \Sigma\text{-term}} \ (\textsc{Opr})$$

Recall (from the end of §2.1.1) that we identify expressions with the abstract syntax trees that they represent. The term $\boldsymbol{\sigma}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ in the conclusion of the (Opr) rule represents the tree



(2.6)

where $\mathbf{t}_i$ in a triangle is the tree represented by the term $\mathbf{t}_i$. Therefore an n-ary function symbol $\boldsymbol{\sigma}$ is a node with n children.

2.2.4 Remark (Inductive definition)**.** This is the first *inductive* definition that we formally made. It means two things:

- those which are shown to be Σ-terms using the two rules are Σ-terms; and
- *only* such are Σ-terms.

The rule (Var) is the base case that generates "atomic" terms; the rule (Opr) is for the step case that, given smaller terms $\mathbf{t}_1, \ldots, \mathbf{t}_n$, generates a bigger term $\boldsymbol{\sigma}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$. Note, however, that when $n = 0$ the rule needs no ingredient terms, yielding a special case

$$\frac{\boldsymbol{\sigma} \in \Sigma_0}{\boldsymbol{\sigma} \text{ is a } \Sigma\text{-term}} \ .$$

We also note that 0-ary function symbols are often called *constants*.

Next we observe the first instance of "definition by induction on the construction of terms." Those who are familiar with functional programming would understand it as a definition by *pattern matching*.

2.2.5 Definition (Free variable)**.** For each Σ-term $\mathbf{t}$, the set $\mathrm{FV}(\mathbf{t})$ of *free variables* of $\mathbf{t}$ is defined as follows.

$$\begin{aligned} \mathrm{FV}(\mathbf{x}) &:= \{\mathbf{x}\} \qquad \text{if } \mathbf{x} \in \mathbf{Var}; \\ \mathrm{FV}\big(\boldsymbol{\sigma}(\mathbf{t}_1, \ldots, \mathbf{t}_n)\big) &:= \mathrm{FV}(\mathbf{t}_1) \cup \cdots \cup \mathrm{FV}(\mathbf{t}_n) \ . \end{aligned}$$

Observe that the above definition amounts to a simple (but informal) definition: "$\mathrm{FV}(\mathbf{t})$ is the set of variables that occur in $\mathbf{t}$." In equational logic every variable occurring in a term is *free*. This is true in propositional logic too; in predicate logic there are *variable binders* (namely $\forall\mathbf{x}$ and $\exists\mathbf{x}$) that make some variable occurrences *bound*.

2.2.6 Example. The signature Σ_{p} that we used in §2.1.1 is such that

$$(\Sigma_{\mathrm{p}})_2 = \{\cdot, +\} \ , \quad (\Sigma_{\mathrm{p}})_0 = (\Sigma_{\mathrm{p}})_1 = (\Sigma_{\mathrm{p}})_3 = (\Sigma_{\mathrm{p}})_4 = \cdots = \emptyset \ .$$

That is, there are two binary operations $\cdot$ and $+$ (and no more).

The signature Σ_{g} used in §2.1.2 is

$$(\Sigma_{\mathrm{g}})_0 = \{e\} \ , \quad (\Sigma_{\mathrm{g}})_1 = \{\, (\_)^{-1} \,\} \ , \quad (\Sigma_{\mathrm{g}})_2 = \{\cdot\} \ ,$$
$$(\Sigma_{\mathrm{g}})_3 = (\Sigma_{\mathrm{g}})_4 = \cdots = \emptyset \ .$$

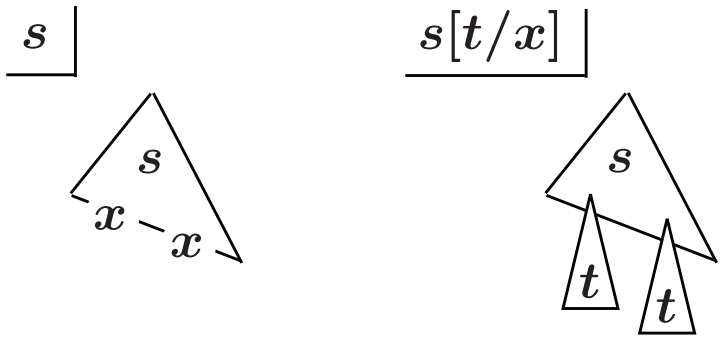
For readability, we use the prefix notation (like $+(\mathbf{t}, \mathbf{s})$) and the infix notation (like $\mathbf{t} + \mathbf{s}$) interchangeably.

2.2.1 Substitution

2.2.7 Definition. Let Σ be a signature, $\mathbf{s}, \mathbf{t}$ be Σ-terms, and $\mathbf{x}$ be a variable. The *substitution* of $\mathbf{s}$ for $\mathbf{x}$ in $\mathbf{t}$, denoted by

$$\mathbf{t}[\mathbf{s}/\mathbf{x}]$$

is the Σ-term obtained by replacing all the (free) occurrences of $\mathbf{x}$ in $\mathbf{t}$ with $\mathbf{s}$. Pictorially:



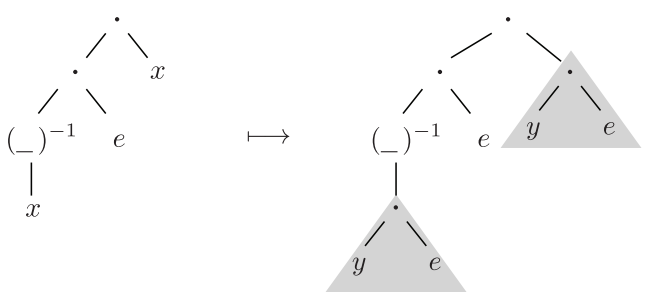
A precise (inductive) definition is possible; see Exercise 2.2. Recall that in equational logic, every variable occurrence is free. Also note the usage of the word "substitute": it means "use instead," and

$$\text{substitute } A \text{ for } B \text{ in } C \quad \simeq \quad \text{replace } B \text{ with } A \text{ in } C.$$

2.2.8 Example.

$$\big(\,(x^{-1} \cdot e) \cdot x\,\big)[y \cdot e/x] \equiv \big(\,((y \cdot e)^{-1} \cdot e) \cdot (y \cdot e)\,\big) \ ,$$

where $\equiv$ denotes the syntactic equality (see Def. 2.2.9).



Note that the expression $\mathbf{t}[\mathbf{s}/\mathbf{x}]$ lives on the meta level. The meta level expression $\big((x^{-1} \cdot e) \cdot x\big)[y \cdot e/x]$ is, concretely on the object level, the expression $\big(((y \cdot e)^{-1} \cdot e) \cdot (y \cdot e)\big)$.

2.2.9 Definition. We denote by $\equiv$ the *syntactic equality*, that is, equality between two syntactic expressions ($\mathbf{s} \equiv \mathbf{t}$ if they are identical "as ASCII sequences").

2.3 Equational Logic: Axiom and Derivation Rule

In the scenario of §2.1.1–2.1.2, the second step was to specify the assumed equalities—equational *axioms*—and establish a basis on which we can derive further equalities. We now do it in a general setting, i.e. for an arbitrary signature Σ.

2.3.1 Equational Formula

2.3.1 Definition (Equational formula)**.** Let Σ be a signature. An *equational formula* over Σ is a pair $(\mathbf{s}, \mathbf{t})$ of Σ-terms delimited by the symbol $=$, written as

$$\mathbf{s} = \mathbf{t} \ . \tag{2.7}$$

An equational formula is also called simply a *formula*.

A remark is in order. The symbol $=$ in (2.7) is *nothing but a delimiting symbol*; our choice of $=$ is arbitrary and it could have been $\simeq$, $\bigstar$ or ©. After all, an equational formula is a mere syntactic expression that represents the following tree.

We emphasize that we are yet to interpret equational formulas, that is, formulas live independently of their meanings or truth values.

2.3.2 Example. Assume a signature Σ is such that $\cdot \in \Sigma$; then examples of equational formulas are: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ and $x \cdot x = x$. We see (informally) that the former is always true in groups, and the latter is not necessarily true.

2.3.2 Axiom and Derivation Rule

Derivation in equational logic is conducted based on mutable *axioms* (the set E of which is a parameter) and the other, fixed, *derivation rules*.

2.3.3 Definition (Algebraic specification)**.** An *algebraic specification* is a pair

$$(\Sigma, E)$$

of a signature Σ, and a set E of equational formulas over Σ. An element $(\mathbf{s} = \mathbf{t}) \in E$ is called an *axiom*.

An algebraic specification (Σ, E) covers the complete set of parameters (i.e. what are "specified" in the scenario of §2.1.1–2.1.2).

2.3.4 Definition (Derivation rules in equational logic)**.** Let (Σ, E) be an algebraic specification. The *derivation rules*[4] over (Σ, E) consist of the following.

$$\frac{}{\mathbf{s}=\mathbf{t}}\ (\textsc{Axiom}), (\mathbf{s}=\mathbf{t}) \in E$$

$$\frac{}{\mathbf{t}=\mathbf{t}}\ (\textsc{Refl}) \qquad \frac{\mathbf{t}=\mathbf{s}}{\mathbf{s}=\mathbf{t}}\ (\textsc{Sym}) \qquad \frac{\mathbf{s}=\mathbf{t} \quad \mathbf{t}=\mathbf{u}}{\mathbf{s}=\mathbf{u}}\ (\textsc{Trans})$$

$$\frac{\mathbf{s}_1=\mathbf{t}_1 \quad \cdots \quad \mathbf{s}_n=\mathbf{t}_n}{\boldsymbol{\sigma}(\mathbf{s}_1,\ldots,\mathbf{s}_n)=\boldsymbol{\sigma}(\mathbf{t}_1,\ldots,\mathbf{t}_n)}\ (\textsc{Cong}), \boldsymbol{\sigma} \in \Sigma_n \qquad \frac{\mathbf{s}=\mathbf{t}}{\mathbf{s}[\mathbf{u}/\mathbf{x}]=\mathbf{t}[\mathbf{u}/\mathbf{x}]}\ (\textsc{Subst}) \tag{2.8}$$

Recall that $\mathbf{s}[\mathbf{u}/\mathbf{x}]$ is a substitution (Def. 2.2.7). The rule (Cong) is called the *congruence rule*.

In (2.8), the rules (Axiom) and (Cong) are the only ones that rely on the parameter (Σ, E); the others are fixed regardless.

2.3.5 Notation (Axiom scheme)**.** Consider, as an example, the axiom(s) that express: "the binary operation $\cdot$ is commutative." Then the set E must contain all of the following formulas (and more):

$$x \cdot y = y \cdot x, \quad y \cdot x = x \cdot y, \quad z \cdot y = y \cdot z, \quad (x \cdot y) \cdot z = z \cdot (x \cdot y), \quad (x \cdot y) \cdot (z \cdot u) = (z \cdot u) \cdot (x \cdot y), \quad \ldots$$

There are infinitely many of such formulas.

We use metavariables and let a single "formula"

$$\mathbf{s} \cdot \mathbf{t} = \mathbf{t} \cdot \mathbf{s}$$

stand for all these (concrete) formulas. Note that $\mathbf{s} \cdot \mathbf{t} = \mathbf{t} \cdot \mathbf{s}$ itself is not a formula unless the metavariables $\mathbf{s}$ and $\mathbf{t}$ are instantiated with concrete terms. Such "formulas" with metavariables (like $\mathbf{s} \cdot \mathbf{t} = \mathbf{t} \cdot \mathbf{s}$) are called *axiom schemes*.

In the same sense, (Refl) in (2.8) (and others) is a *rule scheme*: what is used in a proof tree is a *rule instance* which has the metavariable $\mathbf{t}$ instantiated.

2.3.6 Example. The set E_{p} of axioms that we used in §2.1.1 consists of five axiom schemes (Distr)–(MultAssoc) in §2.1.1.

The axiom schemes (Assoc), (Unit) and (Inv) in §2.1.2 constitute a set of axioms, which we denote by E_{g}.

2.4 Equational Logic: Derivation

Finally we derive equalities—the last step in the scenario of §2.1.1–2.1.2. We formalize a derivation itself as a mathematical object; more specifically as a *tree*.

2.4.1 Definition (Proof tree; derivability)**.** A *proof tree* (also called a *derivation tree* or simply a *proof*) in an algebraic specification (Σ, E) is

– a finite depth tree,

[4] Also called: *deduction rules*, or *deductive rules*.

– each node of which is a legitimate instance of the derivation rules over (Σ, E) (Def. 2.3.4).

An equational formula $\mathbf{s} = \mathbf{t}$ is said to be *derivable* (or *provable*) if there exists a proof tree Π whose root is $\mathbf{s} = \mathbf{t}$. We denote this fact by $\vdash_{(\Sigma,E)} \mathbf{s} = \mathbf{t}$.

2.4.2 Example. The following is a proof tree in (Σ_g, E_g). It witnesses $\vdash_{(\Sigma_g,E_g)} ((xy)^{-1}x)y = e$.

$$\dfrac{\dfrac{\text{Axiom scheme (Assoc)}}{((xy)^{-1}x)y = (xy)^{-1}(xy)}\ \text{(Axiom)} \qquad \dfrac{\text{Axiom scheme (Inv)}}{(xy)^{-1}(xy) = e}\ \text{(Axiom)}}{((xy)^{-1}x)y = e}\ \text{(Trans)}$$

2.4.3 Remark. Note that, in the current setting, axioms (i.e. what are assumed to be true) are always (plain) equations. A more general setting allows *Horn clauses* as axioms: they are formulas of the form

$$\big(\mathbf{s}_1 = \mathbf{t}_1 \wedge \cdots \wedge \mathbf{s}_n = \mathbf{t}_n\big) \supset \mathbf{s} = \mathbf{t}\ ,$$

that is, *conditional equalities*. (We do not even have logical connectives $\wedge$ or $\supset$ yet!)

2.4.4 Proposition. *Assume E is* substitution-closed*, that is, for any $\mathbf{s}, \mathbf{t}, \mathbf{u}$ and $\mathbf{x}$*

$$(\mathbf{s} = \mathbf{t}) \in E \quad \Longrightarrow \quad (\mathbf{s}[\mathbf{u}/\mathbf{x}] = \mathbf{t}[\mathbf{u}/\mathbf{x}]) \in E\ .$$

Then the (Subst) *rule in Def. 2.3.4 is dispensable. That is, for any equational formula $\mathbf{s} = \mathbf{t}$ that is derivable in (Σ, E), there is a proof tree with $\mathbf{s} = \mathbf{t}$ its root that does not use the* (Subst) *rule.*

Proof. See Exercise 2.4. □

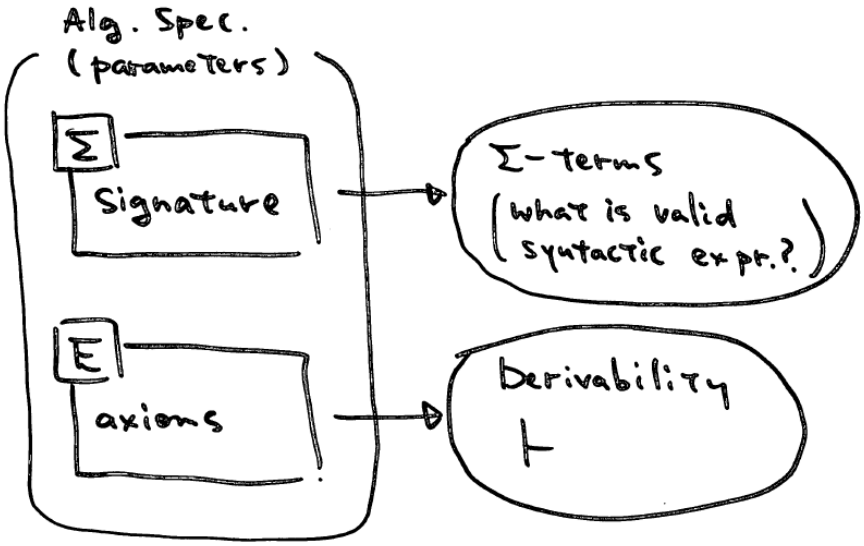


Figure 2.1: Syntactic side of equational logic: an overview

It is remarkable that, by formalizing the notion of proof using derivation rules, we can now speak of proofs as *object level entities*. An example is: "let Π be a proof of a formula A." It is for this reason that logic is often called *metamathematics*.

2.5 Equational Logic: Semantics

What we have discussed up to now are the *syntactic* components of equational logic. They are terms, formulas (recall that these are merely two terms delimited by $=$), and proofs (that are trees labeled with formulas). Note, in particular, that we are yet to assign any "meaning" to terms or formulas. Derivation is a syntactic activity—mechanically applying (syntactic) derivation rules—that can be done without any worry whether the formulas are "true" or "false."

Of course the aim of the syntactic machinery of derivation rules is to derive formulas that are "true." We shall show that this is indeed the case (in the form of the *soundness* and *completeness* results); before that, however, we need a mathematical definition of "truth," or "meaning." This is what *semantics* is about.

2.5.1 Model I: Σ-algebra

2.5.1 Definition. Let Σ be a signature. A Σ-*algebra* $\mathbb{X}$ (also called a *model* of Σ) consists of:

- a nonempty set X (called the *carrier set*); and
- for each $n \in \mathbb{N}$ and for each n-ary operation $\boldsymbol{\sigma} \in \Sigma_n$, its *interpretation* given by a function

$$[\![\boldsymbol{\sigma}]\!]_{\mathbb{X}} \;:\; X^n \longrightarrow X \;. \tag{2.9}$$

We write $\mathbb{X} = \big(X, ([\![\boldsymbol{\sigma}]\!]_{\mathbb{X}})_{\boldsymbol{\sigma}\in\Sigma}\big)$ for a Σ-algebra.

Consider $\Sigma = \Sigma_{\mathrm{g}}$; a Σ_{g}-algebra $\mathbb{X}$ determines

- what the term e designates (since $e \in (\Sigma_{\mathrm{g}})_0$); and
- what the term $e \cdot e$ designates.

Note, however, that at the current stage it is not necessarily the case that $e \cdot e$ and e designate the same element of X. (Thus a Σ-algebra, instead of a (Σ, E)-algebra)

Moreover, if a term contains a variable (e.g. $x^{-1} \cdot e$), its meaning is not yet determined (what does x designate?). We use the following notion.

2.5.2 Definition (Valuation)**.** A *valuation* on a Σ-algebra $\mathbb{X} = \big(X, ([\![\boldsymbol{\sigma}]\!]_{\mathbb{X}})_{\boldsymbol{\sigma}\in\Sigma}\big)$ is a function

$$J \;:\; \mathbf{Var} \longrightarrow X \;.$$

Recall that **Var** is a fixed set of variables (Def. 2.2.2).

2.5.2 Denotation

Given a Σ-algebra—which determines the meaning $[\![\boldsymbol{\sigma}]\!]$ of each operation—and a valuation J—which determines the meaning of variables—we can now assign a meaning to each Σ-term.

2.5.3 Definition (Denotation)**.** Let $\mathbb{X} = \big(X, ([\![\boldsymbol{\sigma}]\!]_{\mathbb{X}})_{\boldsymbol{\sigma}\in\Sigma} \big)$ be a Σ-algebra, and $J : \mathbf{Var} \to X$ be a valuation on $\mathbb{X}$. For each Σ-term $\mathbf{t}$, we define its *denotation*

$$[\![\mathbf{t}]\!]_{\mathbb{X},J} \quad \in X$$

as follows, inductively on the construction of $\mathbf{t}$.

$$\begin{aligned} [\![\mathbf{x}]\!]_{\mathbb{X},J} &:= J(\mathbf{x}) \qquad \text{if } \mathbf{x} \in \mathbf{Var}; \\ [\![\, \boldsymbol{\sigma}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \,]\!]_{\mathbb{X},J} &:= [\![\boldsymbol{\sigma}]\!]_{\mathbb{X}}\big([\![\mathbf{t}_1]\!]_{\mathbb{X},J}, \ldots, [\![\mathbf{t}_n]\!]_{\mathbb{X},J} \big) \ . \end{aligned}$$

Note here that the second line also accounts for the denotation of a constant (i.e. a 0-ary operation). Recall that the term $\boldsymbol{\sigma}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ should be understood as a tree (2.6).

2.5.4 Example. Let us take $\Sigma = \Sigma_{\mathrm{g}}$, the one in §2.1.2 for groups. Let X be the set S_3 of permutations (i.e. bijections) of three elements, that is

$$X := S_3 = \big\{\ (1,2,3),\ (1,3,2),\ (2,1,3),\ (2,3,1),\ (3,1,2),\ (3,2,1)\ \big\} \ ,$$

where for example $(2,1,3)$ denotes the permutation

$$\big[\ 1 \mapsto 2,\ 2 \mapsto 1,\ 3 \mapsto 3\ \big] \ .$$

The set X can be equipped with the following interpretations of the operations in Σ_{g}:

$$\begin{aligned} [\![e]\!]_{\mathbb{X}} \quad &:= \quad (1,2,3) = \mathrm{id} \ , \\ [\![\cdot]\!]_{\mathbb{X}}\ (s,t) \quad &:= \quad t \circ s \ , \\ [\![(\_)^{-1}]\!]\ (s) \quad &:= \quad s^{-1} \ , \end{aligned}$$

where $t \circ s$ denotes the *composition* of permutations

$$(t \circ s)(i) = t(s(i)) \ ;$$

and s^{-1} denotes the *inverse* of the permutation s (i.e. the inverse of the bijective function s). For example:

$$(3,1,2)^{-1} = \left[\begin{array}{l} 1 \mapsto 3 \\ 2 \mapsto 1 \\ 3 \mapsto 2 \end{array}\right]^{-1} = \left[\begin{array}{l} 1 \mapsto 2 \\ 2 \mapsto 3 \\ 3 \mapsto 1 \end{array}\right] = (2,3,1) \ .$$

This determines a Σ_{g}-algebra $\mathbb{X}$.

Let J be a valuation on $\mathbb{X}$ such that

$$J(x) \ = \ (2,1,3) \quad \text{ and } \quad J(y) \ = \ (3,1,2) \ ;$$

then we have

$$\begin{aligned} [\![x \cdot (y^{-1})]\!]_{\mathbb{X},J} &= [\![\cdot]\!]_{\mathbb{X}}\big(J(x),\ [\![(\_)^{-1}]\!]_{\mathbb{X}}(J(y)) \big) \\ &= (3,1,2)^{-1} \circ (2,1,3) \\ &= (2,3,1) \circ (2,1,3) \\ &= (3,2,1) \ . \end{aligned}$$

The denotation of a term $\mathbf{t}$ relies only on the valuations of the variables that occur in $\mathbf{t}$. To put it precisely:

2.5.5 Lemma. *Let J, J' be two valuations on $\mathbb{X}$, and $\mathbf{t}$ be a term. Assume that*

$$J(\mathbf{x}) \;=\; J'(\mathbf{x}) \qquad \textit{for each } \mathbf{x} \in \mathrm{FV}(\mathbf{t}),$$

where $\mathrm{FV}(\mathbf{t})$ is from Def. 2.2.5. Then we have

$$[\![\mathbf{t}]\!]_{\mathbb{X},J} \;=\; [\![\mathbf{t}]\!]_{\mathbb{X},J'} \;.$$

Proof. By induction on the construction of a term $\mathbf{t}$. □

We briefly discuss the relationship between denotation $[\![\_]\!]$ and substitution $\mathbf{s}[\mathbf{t}/\mathbf{x}]$.

In the next definition, the valuation $J[\mathbf{x} \mapsto a]$ is almost J, but the value assigned to $\mathbf{x}$ is updated.

2.5.6 Definition (Update of a valuation)**.** Let $J : \mathbf{Var} \to X$ be a valuation over a Σ-algebra $\mathbb{X}$; $\mathbf{x} \in \mathbf{Var}$; and $a \in X$. We define a new valuation $J[\mathbf{x} \mapsto a]$ by

$$\begin{array}{rccc} J[\mathbf{x} \mapsto a] \;:\; & \mathbf{Var} & \longrightarrow & X \\ & \mathbf{y} & \longmapsto & \begin{cases} J(\mathbf{y}) & \text{if } \mathbf{y} \not\equiv \mathbf{x}; \\ a & \text{if } \mathbf{y} \equiv \mathbf{x}. \end{cases} \end{array}$$

2.5.7 Proposition. $[\![\, \mathbf{s}[\mathbf{t}/\mathbf{x}] \,]\!]_{\mathbb{X},J} \;=\; [\![\, \mathbf{s} \,]\!]_{\mathbb{X},J[\mathbf{x} \mapsto [\![\mathbf{t}]\!]_{\mathbb{X},J}]}$.

Proof. By induction on the construction of the term $\mathbf{s}$. □

2.5.3 Truth Value of a Formula

Now that we know which element a term $\mathbf{t}$ designates (namely its denotation $[\![\mathbf{t}]\!]$), we can *mathematically define* whether an equational formula—itself a pair of terms delimited by =—is true or not.

2.5.8 Definition (Truth Value)**.** Let Σ be a signature; $\mathbf{s} = \mathbf{t}$ be an equational formula over Σ; $\mathbb{X}$ be a Σ-algebra; and J be a valuation over $\mathbb{X}$.

- The formula $\mathbf{s} = \mathbf{t}$ is said to be *true under $\mathbb{X}$ and J* if we have

  $$[\![\mathbf{s}]\!]_{\mathbb{X},J} \;=\; [\![\mathbf{t}]\!]_{\mathbb{X},J} \;.$$

  Note that $=$ here means the equality between elements of the set X (unlike that in $\mathbf{s} = \mathbf{t}$ which is merely a delimiting symbol.)

- The formula $\mathbf{s} = \mathbf{t}$ is said to be *valid in $\mathbb{X}$* if, under any valuation $J : \mathbf{Var} \to X$ on $\mathbb{X}$, the formula $\mathbf{s} = \mathbf{t}$ is true. We write

  $$\mathbb{X} \models \mathbf{s} = \mathbf{t}$$

  for this fact.

The word "valid" roughly means: "always true." The symbol $\models$ is here used for *semantical* validity/truth; recall that $\vdash$ denotes *syntactic* derivability (Def. 2.4.1).

2.5.9 Example. In the Σ_{g}-algebra $\mathbb{X}$ in Example 2.5.4, the formula

$$x^{-1} \cdot x = e$$

is valid; it is true regardless of what element of X is assigned to the variable x.

In contrast, the formula

$$x \cdot x = x$$

is not valid. Under some valuations it becomes true coincidentally (for example: if $J(x) = \mathrm{id} = (1,2,3)$); but such is not always the case.

2.5.10 Remark. Are you fed up with all the "bureaucracy" around here? (Modern) mathematics is a constant fight between *intuition* and *precision*. Without precision what you do is no longer mathematics. However, without intuition you never "understand" mathematics. (Think: what is to understand mathematics?)

What you usually find in mathematics books is a monster of precision: it champions logical rigor over intuitions since the latter are easily subject to objections. You need to learn to read out the intuitions hidden behind the text—this task is much like finding out the real intention of bureaucrats and/or politicians behind an official government document. (As UT students you must be good at it `:-P`)

One recommendable way of doing it is to follow the path of the "founding father" who built the theory—i.e. to be driven by the original motivations. (This is probably why some mathematicians say students should learn original papers, not textbooks) I intend the current notes to be written that way, too, first explaining motivations and intuitions ("we want to do this!") and then laying out mathematical definitions ("to do it precisely, ...").

2.5.4 Model II: (Σ, E)-algebra

We have defined the notions of Σ-algebra, and truth of formulas. With these we can finally define (the generalization of the notion of) group: it is a set with certain operations that satisfy certain equations.

2.5.11 Definition ((Σ, E)-algebra)**.** Let (Σ, E) be an algebraic specification, and $\mathbb{X} = \big(X, (\llbracket \boldsymbol{\sigma} \rrbracket_{\mathbb{X}})_{\boldsymbol{\sigma} \in \Sigma}\big)$ be a Σ-algebra.

The data $\mathbb{X}$ is said to be a (Σ, E)-*algebra* if it satisfies all the axioms in E. More precisely: for each axiom $(\mathbf{s} = \mathbf{t}) \in E$,

$$\mathbb{X} \models \mathbf{s} = \mathbf{t} \ .$$

2.5.12 Definition (Validity)**.** Let (Σ, E) be an algebraic specification, and $\mathbf{s} = \mathbf{t}$ be an equational formula over Σ. We say $\mathbf{s} = \mathbf{t}$ is *valid* over (Σ, E), and write

$$\models_{(\Sigma, E)} \ \mathbf{s} = \mathbf{t} \ ,$$

if for any (Σ, E)-algebra $\mathbb{X}$ we have

$$\mathbb{X} \models \mathbf{s} = \mathbf{t} \ .$$

2.6 Equational Logic: Syntax vs. Semantics

We have introduced

- *derivation rules*, that are syntactic machinery that derive equational formulas, and
- *semantics* of formulas, that mathematically defines whether a formula is true or not.

Our next goal is to see if the former machinery works as expected. (For example: is the formula $((xy)^{-1}x)y = e$, derived in Example 2.4.2, always true?)

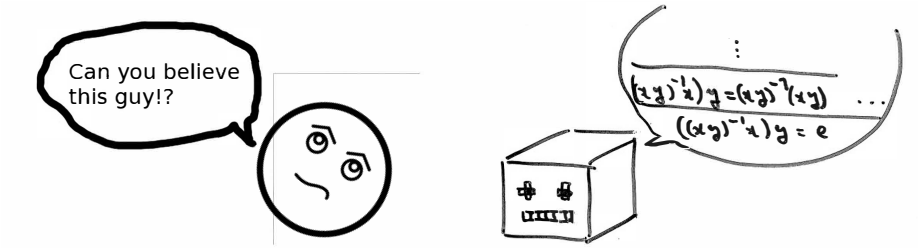


Figure 2.2: Derivation rules vs. semantics

More specifically, our interest is in the following properties.

- *Soundness*: what is derived is true, that is

$$\models_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \quad \Longleftarrow \quad \vdash_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \ .$$

- *Completeness*: what is true is derived, that is

$$\models_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \quad \Longrightarrow \quad \vdash_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \ .$$

A bit more of general discussion is due. Soundness says "the machine never lies" and is considered to be a mandatory property of a deductive system. Completeness[5], in contrast, is like a bonus. It is desirable but not mandatory; in fact there are many "truths" that are just too complicated for any deductive system to be complete against. *Gödel's incompleteness theorem* is a celebrated result that states impossibility to be complete.

2.6.1 Remark. Here we are using the terms "machines," "machinery," "complicated," etc. in very sloppy ways. After finishing the second half of this course, you are supposed to be able to make all these precise.

In equational logic we do have soundness and completeness. We take a detailed look at the proofs: each of them employs a (different) characteristic method whose importance in computer science is paramount.

[5] In this sentence, "completeness" includes (the above kind and) others such as the notion of completeness used in Gödel's in*completeness* theorem, which is stronger than the above kind of completeness in some sense (this "sense" will be explained around (9.5) in Section 9.2).

2.6.1 Soundness

2.6.2 Theorem (Soundness)**.** *Let* (Σ, E) *be an algebraic specification, and* $\mathbf{s} = \mathbf{t}$ *be an equational formula over* Σ*. We have*

$$\vdash_{(\Sigma,E)} \ \mathbf{s} = \mathbf{t} \quad \Longrightarrow \quad \models_{(\Sigma,E)} \ \mathbf{s} = \mathbf{t} \ .$$

The proof is by: "induction on the derivation of $\mathbf{s} = \mathbf{t}$"; or equivalently: "induction on the height of a proof tree for $\mathbf{s} = \mathbf{t}$." Similar inductive arguments have occurred before (mostly in exercises); here we describe (some of) its details.

Proof. Assume $\vdash_{(\Sigma,E)} \ \mathbf{s} = \mathbf{t}$. By definition of $\vdash_{(\Sigma,E)}$ (Def. 2.4.1), there exists a proof tree Π whose root is $\mathbf{s} = \mathbf{t}$. We choose one such Π.

We shall show:

> for any (Σ, E)-algebra $\mathbb{X}$ and any valuation J on $\mathbb{X}$, we have $[\![\mathbf{s}]\!]_{\mathbb{X},J} = [\![\mathbf{t}]\!]_{\mathbb{X},J}$

by induction on the height $\mathrm{hgt}(\Pi)$ of the proof tree Π.

1. Case: $\mathrm{hgt}(\Pi) = 0$. By inspection of the rules (Def. 2.3.4), this is possible in the following three cases.
   (a) If the last rule applied is (Axiom): in this case $(\mathbf{s} = \mathbf{t}) \in E$. It is the definition of (Σ, E)-algebra itself that, for any (Σ, E)-algebra $\mathbb{X}$, we have $\mathbb{X} \models \mathbf{s} = \mathbf{t}$.
   (b) If the last rule applied is (Refl): in this case it happens that $\mathbf{s} \equiv \mathbf{t}$ (the terms are syntactically identical). Therefore $[\![\mathbf{s}]\!]_{\mathbb{X},J} = [\![\mathbf{t}]\!]_{\mathbb{X},J}$ is trivial.
   (c) If the last rule applied is (Cong), with $\boldsymbol{\sigma}$ being 0-ary: in this case too we have $\mathbf{s} \equiv \mathbf{t} \equiv \boldsymbol{\sigma}$, a constant.
2. Case: $\mathrm{hgt}(\Pi) > 0$. We distinguish cases according to the last applied rule in Π.
   (a) If it is (Cong): we have

$$\mathbf{s} \equiv \boldsymbol{\sigma}(\mathbf{s}_1, \ldots, \mathbf{s}_n) \ , \quad \mathbf{t} \equiv \boldsymbol{\sigma}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \ , \tag{2.10}$$

   and the formulas

$$\mathbf{s}_1 = \mathbf{t}_1 \ , \quad \ldots , \quad \mathbf{s}_n = \mathbf{t}_n$$

   have proof trees $\Pi_1, \ldots, \Pi_n$ whose height is smaller than that of Π. See below.

$$\Pi \quad \equiv \quad \left[\begin{array}{c} \begin{array}{ccccc} \vdots\ \Pi_1 & & \vdots\ \Pi_2 & & \vdots\ \Pi_n \\ \mathbf{s}_1 = \mathbf{t}_1 & \mathbf{s}_2 = \mathbf{t}_2 & \cdots & \mathbf{s}_n = \mathbf{t}_n & \end{array} \\ \hline \mathbf{s} = \mathbf{t} \end{array} \ (\textsc{Cong}) \right]$$

   By the induction hypothesis, we have for each $i = 1, \ldots, n$,

$$[\![\mathbf{s}_i]\!]_{\mathbb{X},J} = [\![\mathbf{t}_i]\!]_{\mathbb{X},J} \qquad \text{for any } \mathbb{X} \text{ and } J. \tag{2.11}$$

Now

$$\begin{aligned}
[\![\mathbf{s}]\!]_{\mathbb{X},J} &= [\![\boldsymbol{\sigma}]\!]_{\mathbb{X}}\big([\![\mathbf{s}_1]\!]_{\mathbb{X},J},\ldots,[\![\mathbf{s}_n]\!]_{\mathbb{X},J}\big) && \text{by Def. 2.5.3 and (2.10)}\\
&= [\![\boldsymbol{\sigma}]\!]_{\mathbb{X}}\big([\![\mathbf{t}_1]\!]_{\mathbb{X},J},\ldots,[\![\mathbf{t}_n]\!]_{\mathbb{X},J}\big) && \text{by (2.11)}\\
&= [\![\mathbf{t}]\!]_{\mathbb{X},J} && \text{by Def. 2.5.3 and (2.10).}
\end{aligned}$$

(b) If it is (Subst): we have

$$\mathbf{s} \equiv \mathbf{s}'[\mathbf{u}/\mathbf{x}] \ , \quad \mathbf{t} \equiv \mathbf{t}'[\mathbf{u}/\mathbf{x}] \ , \tag{2.12}$$

and

$$\Pi \quad \equiv \quad \left[\begin{array}{c} \vdots\ \Pi' \\ \dfrac{\mathbf{s}' = \mathbf{t}'}{\mathbf{s} = \mathbf{t}}\ (\textsc{Subst}) \end{array} \right] .$$

By the induction hypothesis, we have

$$[\![\mathbf{s}']\!]_{\mathbb{X},J} = [\![\mathbf{t}']\!]_{\mathbb{X},J} \qquad \text{for any } \mathbb{X} \text{ and } J. \tag{2.13}$$

Here a valuation J can be anything; thus in particular

$$[\![\mathbf{s}']\!]_{\mathbb{X},J[\mathbf{x}\mapsto[\![\mathbf{u}]\!]_{\mathbb{X},J}]} = [\![\mathbf{t}']\!]_{\mathbb{X},J[\mathbf{x}\mapsto[\![\mathbf{u}]\!]_{\mathbb{X},J}]} \qquad \text{for any } \mathbb{X} \text{ and } J. \tag{2.14}$$

Use Prop. 2.5.7 and the claim follows.

(c) If it is (Sym) or (Trans): exercise (easy). □

Notice that the distinction between the cases 1. and 2. (i.e. if hgt(Π) is 0 or not) is in fact vacuous.

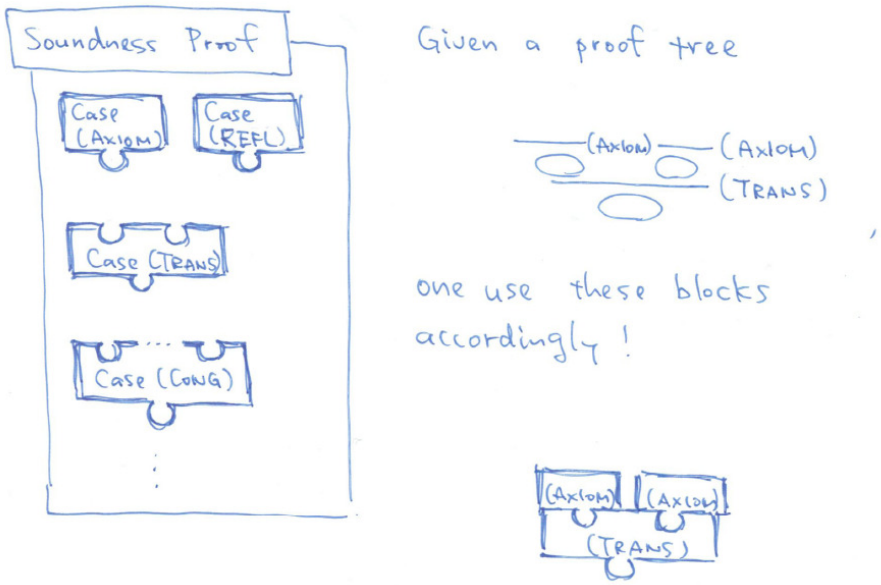
2.6.3 Remark. (You can skip this) The underlying mathematical structure that is relevant to the above inductive proof is that of a *well-founded set*. Here we are appealing to the well-founded structure of

- the set of proof trees over (Σ, E), or
- (via the function hgt) the set $\mathbb{N}$.

2.6.4 Remark. We made seven cases (two for the (Cong) rule; one for each of the other rules) in the above proof. Each case can be understood as a "building block" of the proof that a derivable formula is valid.

Take for example the proof tree in Example 2.4.2. To show that the root formula $((xy)^{-1}x)y = e$ is valid, you use

- the case 1.-(a) to show that $((xy)^{-1}x)y = (xy)^{-1}(xy)$ is valid;
- the case 1.-(a) again to show that $(xy)^{-1}(xy) = e$ is valid;
- and then the case 2.-(c) to show that the root formula $((xy)^{-1}x)y = e$ is valid.



2.6.2 Completeness

Completeness states that the machinery of derivation rules is powerful enough that it derives *all* the valid formulas.

2.6.5 Theorem (Completeness)**.** *Let* (Σ, E) *be an algebraic specification, and* $\mathbf{s} = \mathbf{t}$ *be an equational formula over* Σ*. We have*

$$\models_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \implies \vdash_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \ .$$

Let us stop a bit and think how we would be able to prove this result. A direct proof would require

> to construct a proof tree for a given valid formula

which sounds rather hard. (By induction? On what??) Let us instead consider the *contraposition* of the original statement:

$$\begin{array}{lll} \not\vdash_{(\Sigma,E)} \mathbf{s} = \mathbf{t} & \implies & \not\models_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \ , \qquad \text{that is,} \\ \text{there is no proof for } \mathbf{s} = \mathbf{t} & \implies & \text{there exist } \mathbb{X} \text{ and } J \text{ such that } [\![\mathbf{s}]\!]_{\mathbb{X},J} \neq [\![\mathbf{t}]\!]_{\mathbb{X},J}. \end{array} \tag{2.15}$$

It is this statement that we are going to prove; hence we,

> given an underivable formula $\mathbf{s} = \mathbf{t}$, construct a *counter model* $\mathbb{X}, J$ that makes the formula false.

But how? We use *syntactic ingredients* in the construction of a counter model—the carrier X of $\mathbb{X}$ would be the set of (certain equivalence classes of) Σ-terms.

To summarize: what we do now is a standard strategy in completeness proofs, that is,

construction of a counter model with syntactic ingredients

The counter model $\mathbb{X}, J$ constructed in the current setting will in fact be a *universal* one—meaning that this single model falsifies *all* underivable formulas.[6] That is,

$$\vdash_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \quad \Longleftrightarrow \quad \mathbb{X} \models \mathbf{s} = \mathbf{t} \ . \tag{2.16}$$

(Note that $\Longrightarrow$ is ensured by soundness) To put it differently: $\mathbb{X}$ is a (Σ, E)-algebra that validates

- formulas that are valid in *all the* (Σ, E)-algebras, and
- nothing more.

Such an algebra is said to be *free*.[7]

In the current notes we do not formally define what a free algebra is (a proper definition calls for the notion of *homomorphism* between algebras). Some intuitions, nevertheless: a free algebra is an algebra where the minimal set of equational formulas hold. For example, the one-element set $1 = \{0\}$ can be made into a (Σ, E)-algebra $\mathbf{1}$ for any (Σ, E)—the interpretations $[\![\boldsymbol{\sigma}]\!]_{\mathbf{1}} : 1^n \to 1$ are obvious and the axioms in E are trivially satisfied. In this algebra $\mathbf{1}$, many more equations than required by E are *forced* to hold. In contrast, a free algebra is *free* from such oppression—it satisfies only those which are required by E.

In the following proof, before describing a free (Σ, E)-algebra $\mathbb{X}$ (which is a counter model), we first describe a free Σ-algebra $\mathbb{X}'$.

Proof. (of Thm. 2.6.5) We first define a Σ-algebra

$$\mathbb{X}' = \big(X', ([\![\boldsymbol{\sigma}]\!]_{\mathbb{X}'})_{\boldsymbol{\sigma}\in\Sigma} \big)$$

in the following syntactic manner.

- The carrier set X' is the set of all Σ-terms (using variables from **Var**).
- Each operation $\boldsymbol{\sigma} \in \Sigma_n$ is interpreted syntactically, by

$$\begin{array}{rccc} [\![\boldsymbol{\sigma}]\!]_{\mathbb{X}'} \ : & (X')^n & \longrightarrow & X' \\ & (\,\mathbf{t}_1, \ldots, \mathbf{t}_n\,) & \longmapsto & \boldsymbol{\sigma}(\,\mathbf{t}_1, \ldots, \mathbf{t}_n\,) \ . \end{array} \tag{2.17}$$

  Note here that the expression $\boldsymbol{\sigma}(\,\mathbf{t}_1, \ldots, \mathbf{t}_n\,)$ is a Σ-term and hence indeed belongs to X'.

It is obvious that $\mathbb{X}'$ is indeed a Σ-algebra; however it is in general not (yet) a (Σ, E)-algebra. Consider, for example, $(\Sigma_{\mathrm{g}}, E_{\mathrm{g}})$ in Example 2.3.6. We have

$$e \in (\Sigma_{\mathrm{g}})_0 \ , \quad \cdot \in (\Sigma_{\mathrm{g}})_2 \quad \text{and} \quad (\mathbf{x} \cdot e = \mathbf{x}) \ \in E_{\mathrm{g}} \ .$$

For $\mathbb{X}'$ to be a $(\Sigma_{\mathrm{g}}, E_{\mathrm{g}})$-algebra we need to have $e \cdot e = e$ as elements of X'; this is not the case since the two terms $e \cdot e$ and e are different as syntactic expressions (i.e. $e \cdot e \not\equiv e$).

[6] In other cases (i.e. for other kinds of logic) it is often the case that the constructed counter model is different depending on the given underivable formula.

[7] Recall the notion of free group if you have heard of it. Its construction too bears a strong syntactic flavor.

We therefore quotient (i.e. identify some elements of) the algebra $\mathbb{X}'$ so that it satisfies the required equational axioms. Let us define a binary relation $\sim_E \subseteq X' \times X'$ by

$$\mathbf{s} \sim_E \mathbf{t} \quad \overset{\text{def}}{\iff} \quad \vdash_{(\Sigma,E)} \ \mathbf{s} = \mathbf{t} \ . \tag{2.18}$$

2.6.6 Sublemma. The relation $\sim_E$ is an equivalence relation.

Proof. (Of Sublem. 2.6.6) We show its transitivity; reflexivity and symmetry is left as exercise. Assume $\mathbf{s} \sim_E \mathbf{t}$ and $\mathbf{t} \sim_E \mathbf{u}$; we need to show $\mathbf{s} \sim_E \mathbf{u}$.

By the definition of $\sim_E$ we have

$$\vdash_{(\Sigma,E)} \ \mathbf{s} = \mathbf{t} \quad \text{and} \quad \vdash_{(\Sigma,E)} \ \mathbf{t} = \mathbf{u} \ ,$$

that is, there are proof trees Π, Π' over (Σ, E) such that

$$\Pi \ \equiv \ \left[\begin{array}{c} \vdots \\ \hline \mathbf{s} = \mathbf{t} \end{array} \right] \ ; \qquad \Pi' \ \equiv \ \left[\begin{array}{c} \vdots \\ \hline \mathbf{t} = \mathbf{u} \end{array} \right] \ .$$

Using these two, we construct the following proof tree:

$$\dfrac{\begin{array}{cc} \vdots\ \Pi & \vdots\ \Pi' \\ \mathbf{s} = \mathbf{t} & \mathbf{t} = \mathbf{u} \end{array}}{\mathbf{s} = \mathbf{u}} \ (\textsc{Trans}) \ ,$$

where (Trans) is the rule in Def. 2.3.4. Thus $\vdash_{(\Sigma,E)} \ \mathbf{s} = \mathbf{u}$; this proves the claim. □

Therefore we can take the quotient set by $\sim_E$ and set it to be the set X. That is,

$$X \ := \ X'/\sim_E \ = \ \left\{ \, [\mathbf{s}]_{\sim_E} \, \middle| \, \mathbf{s} \in X' \, \right\} \ .$$

We now endow the set X with a Σ-algebra structure, and prove that it satisfies all the axioms in E. An operation $\boldsymbol{\sigma} \in \Sigma_n$ is interpreted by:

$$\begin{array}{rccc} [\![\boldsymbol{\sigma}]\!]_{\mathbb{X}} \ : & (X)^n & \longrightarrow & X \\ & \big(\, [\mathbf{t}_1]_{\sim_E}, \ldots, [\mathbf{t}_n]_{\sim_E} \, \big) & \longmapsto & \big[\, \boldsymbol{\sigma}(\, \mathbf{t}_1, \ldots, \mathbf{t}_n \,) \, \big]_{\sim_E} \ . \end{array} \tag{2.19}$$

We need to check that this function $[\![\boldsymbol{\sigma}]\!]_{\mathbb{X}}$ is *well-defined* (it appears to depend on the choice of a representative $\mathbf{t}_i$ from the class $[\mathbf{t}_i]_{\sim_E}$).

2.6.7 Sublemma. The function $[\![\boldsymbol{\sigma}]\!]_{\mathbb{X}}$ is well-defined. That is:

$$\begin{array}{l} \mathbf{t}_1 \sim_E \mathbf{t}'_1 \ , \quad \ldots \ , \quad \mathbf{t}_n \sim_E \mathbf{t}'_n \\ \qquad \Longrightarrow \quad \boldsymbol{\sigma}(\, \mathbf{t}_1, \ldots, \mathbf{t}_n \,) \ \sim_E \ \boldsymbol{\sigma}(\, \mathbf{t}'_1, \ldots, \mathbf{t}'_n \,) \ . \end{array}$$

Proof. (Of Sublem. 2.6.7) Like the proof of Sublem. 2.6.6, using the (Cong) rule. □

Therefore the data

$$\mathbb{X} \ := \ \big(\, X, ([\![\boldsymbol{\sigma}]\!]_{\mathbb{X}})_{\boldsymbol{\sigma} \in \Sigma} \, \big)$$

constitutes a Σ-algebra. Before proving that $\mathbb{X}$ is indeed a (Σ, E)-algebra, let us introduce a *canonical valuation* J_{c}:

$$\begin{array}{rccc} J_{\mathrm{c}} \ : & \mathbf{Var} & \longrightarrow & X \\ & \mathbf{x} & \longmapsto & [\, \mathbf{x} \,]_{\sim_E} \ . \end{array}$$

Note here that $\mathbf{x} \in \mathbf{Var}$ is itself a term, thus $[\, \mathbf{x} \,]_{\sim_E}$ is an element of X. This canonical valuation acts like a projection map. Furthermore, this extends as the following.

2.6.8 Sublemma. For any Σ-term $\mathbf{t}$, we have $[\![\mathbf{t}]\!]_{\mathbb{X},J_c} = [\mathbf{t}]_{\sim_E}$.

Proof. (Of Sublem. 2.6.8) By induction on the construction of a term $\mathbf{t}$. □

2.6.9 Sublemma. $\mathbb{X}$ is a (Σ, E)-algebra.

Proof. (Of Sublem. 2.6.9) Let $(\mathbf{s} = \mathbf{t}) \in E$ and $J : \mathbf{Var} \to X$ be an arbitrary valuation over $\mathbb{X}$. We need to show that $[\![\mathbf{s}]\!]_{\mathbb{X},J} = [\![\mathbf{t}]\!]_{\mathbb{X},J}$.

Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be an enumeration of the variables that occur in $\mathbf{s}$ or $\mathbf{t}$ (i.e. $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} = \mathrm{FV}(\mathbf{s}) \cup \mathrm{FV}(\mathbf{t})$). The valuation J assigns an element of X to each of these variables; let us choose a representative $\mathbf{u}_i$ and set

$$J(\mathbf{x}_1) \;=\; [\mathbf{u}_1]_{\sim_E} \;, \quad \ldots \;, \quad J(\mathbf{x}_n) \;=\; [\mathbf{u}_n]_{\sim_E} \;. \tag{2.20}$$

Then we have

$$\begin{aligned}
[\![\mathbf{s}]\!]_{\mathbb{X},J} &= [\![\mathbf{s}]\!]_{\mathbb{X},J_c[\mathbf{x}_i \mapsto J(\mathbf{x}_i)]} && \text{by Lem. 2.5.5} \\
&= [\![\mathbf{s}]\!]_{\mathbb{X},J_c[\mathbf{x}_i \mapsto [\mathbf{u}_i]_{\sim_E}]} && \text{by (2.20)} \\
&= [\![\mathbf{s}]\!]_{\mathbb{X},J_c[\mathbf{x}_i \mapsto [\![\mathbf{u}_i]\!]_{\mathbb{X},J_c}]} && \text{by Sublem. 2.6.8} \\
&= [\![\mathbf{s}[\mathbf{u}_i/\mathbf{x}_i]]\!]_{\mathbb{X},J_c} && \text{by Lem. 2.5.7.}
\end{aligned}$$

Similarly we have: $[\![\mathbf{t}]\!]_{\mathbb{X},J} = [\![\mathbf{t}[\mathbf{u}_i/\mathbf{x}_i]]\!]_{\mathbb{X},J_c}$. Now:

$$\begin{aligned}
[\![\mathbf{s}]\!]_{\mathbb{X},J} &= [\![\mathbf{s}[\mathbf{u}_i/\mathbf{x}_i]]\!]_{\mathbb{X},J_c} && \text{by the above} \\
&= \big[\, \mathbf{s}[\mathbf{u}_i/\mathbf{x}_i] \,\big]_{\sim_E} && \text{by Sublem. 2.6.8} \\
&= \big[\, \mathbf{t}[\mathbf{u}_i/\mathbf{x}_i] \,\big]_{\sim_E} && (*) \\
&= [\![\mathbf{t}[\mathbf{u}_i/\mathbf{x}_i]]\!]_{\mathbb{X},J_c} && \text{by Sublem. 2.6.8} \\
&= [\![\mathbf{t}]\!]_{\mathbb{X},J} && \text{by the above,}
\end{aligned}$$

where the equality $(*)$—i.e. the fact that $\vdash_{(\Sigma,E)} \mathbf{s}[\mathbf{u}_i/\mathbf{x}_i] = \mathbf{t}[\mathbf{u}_i/\mathbf{x}_i]$—is witnessed by the following proof tree.

$$\dfrac{\dfrac{(\mathbf{s} = \mathbf{t}) \in E}{\mathbf{s} = \mathbf{t}}\ (\textsc{Axiom})}{\mathbf{s}[\mathbf{u}_i/\mathbf{x}_i] = \mathbf{t}[\mathbf{u}_i/\mathbf{x}_i]}\ (\textsc{Subst})$$

This concludes the proof of Sublem. 2.6.9. □

Let us turn back to the proof of Thm. 2.6.5. We shall show that

$$\nvdash_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \quad \Longrightarrow \quad [\![\mathbf{s}]\!]_{\mathbb{X},J_c} \neq [\![\mathbf{t}]\!]_{\mathbb{X},J_c} \;,$$

which immediately proves the $\Longleftarrow$ direction of (2.16). But this follows immediately from Sublem. 2.6.8 and the definition of $\sim_E$ (2.18). □

The proof was rather lengthy but its basic idea was simple: the set of terms, quotiented by the derivable equalities, carries a free (Σ, E)-algebra.

2.6.10 Remark. The above construction may be hard to understand for those who are not used to abstract algebra. For some intuition, we list some elements

of the free Σ_g-algebra $\mathbb{X}'$ and those of the (Σ_g, E_g)-algebra $\mathbb{X}$, for the algebraic specification (Σ_g, E_g) in Example 2.2.6.

$$
\begin{aligned}
X' &= \{e, x, e \cdot e, \dots\} \ , \\
X &= \{\ [e]_{\sim_E}, [x]_{\sim_E}, [e \cdot e]_{\sim_E}, \dots\ \} \\
&= \left\{ \begin{array}{l} \{e, e \cdot e, e \cdot (e \cdot e), (y \cdot y^{-1}) \cdot e, \dots\}, \\ \{x, x \cdot e, x \cdot (x^{-1} \cdot x), (e \cdot x) \cdot e, \dots\}, \\ \{e, e \cdot e, e \cdot (e \cdot e), (y \cdot y^{-1}) \cdot e, \dots\}, \\ \dots \end{array} \right\}
\end{aligned}
$$

Note that $[e]_{\sim_E}$ and $[e \cdot e]_{\sim_E}$, are equal as the elements of $X = X'/\sim_E$.

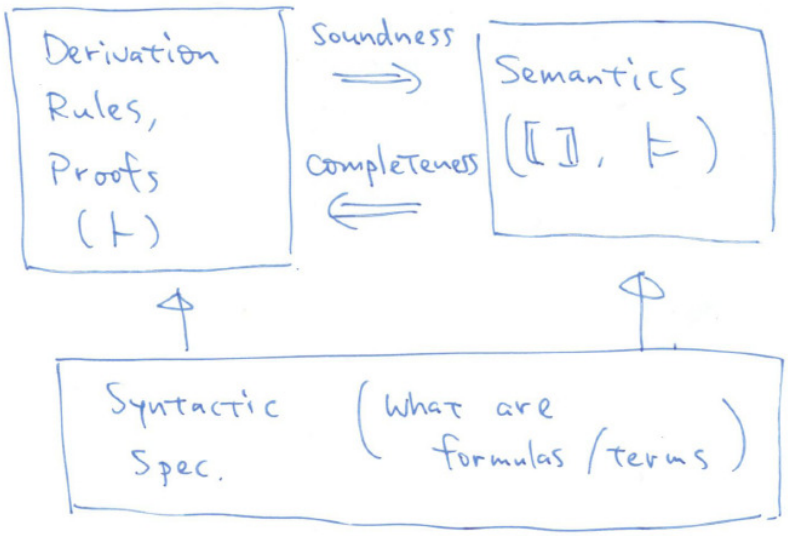
2.7 What is Logic?

To summarize the current chapter—meant to be a showcase of formal logic—we have seen a framework of formal logic consists of the following building blocks.

- *Syntactic specification* that makes clear which syntactic expressions are legitimate formulas (Def. 2.2.3, 2.3.1).
- *Derivation rules* that derive formulas in a syntactic manner (Def. 2.3.4).
- *Semantics* that mathematically defines which formulas are true (Def. 2.5.8).

Connecting the latter two are the *soundness* and *completeness* results. They are proved by characteristic proof methods, by *induction on derivation* and *syntactic construction of a counter model*.

In the following two chapters we exhibit more complicated systems of formal logic—namely propositional logic and predicate logic. The basic overview of a framework (see below) stays the same.



Exercises

2.1. We have observed many instances of arguments "by induction on the construction of Σ-terms." In fact the "highschool induction" (i.e. induction on $n \in \mathbb{N}$) is a special case; describe a suitable choice of the parameter Σ.

2.2. Describe a precise, inductive definition of the substitution $\mathbf{t}[\mathbf{s}/\mathbf{x}]$ (cf. Def. 2.2.7). (Hint: induction on what?)

Based on this precise definition, prove the following: if $\mathbf{x} \notin \mathrm{FV}(\mathbf{t})$, we have $\mathbf{t}[\mathbf{s}/\mathbf{x}] \equiv \mathbf{t}$.

2.3 (From [19])**.** Construct a proof tree in $(\Sigma_\mathrm{g}, E_\mathrm{g})$ for each of the formulas

$$(x^{-1}x)x^{-1} = x^{-1} \quad , \quad x^{-1}(xx^{-1}) = (x^{-1}x)x^{-1} \quad , \quad (xx^{-1})(xx^{-1}) = x(x^{-1}(xx^{-1})) \quad .$$

Let P_1, P_2, P_3 be the proof trees, respectively. Use them in a proof tree for the formula

$$(xx^{-1})(xx^{-1}) = xx^{-1} \quad .$$

2.4. Prove Prop. 2.4.4.

Hint: write $\vdash'_{(\Sigma,E)}$ for the derivability without using (Subst). The fact

$$\vdash_{(\Sigma,E)} \mathbf{s} = \mathbf{t} \quad \Longrightarrow \quad \vdash'_{(\Sigma,E)} \mathbf{s} = \mathbf{t}$$

can be shown by induction on derivation.

2.5. Give a detailed proof of Lem. 2.5.5.

2.6. Give a detailed proof of Prop. 2.5.7.

2.7. Modify the interpretations in Example 2.5.4 so that $\mathbb{X}$ is *not* a $(\Sigma_\mathrm{g}, E_\mathrm{g})$-algebra.

2.8. Fill in the details of the proof of Thm. 2.6.5.

Chapter 3

Propositional Logic

Propositional logic is a system where atomic statements like

- it rains today;
- UT Hongo Campus is located in Tokyo;
- Tokyo Disney Land is located in Tokyo; or
- Dept. Inf. Sci., U. Tokyo is located in Tokyo

are combined using *logical connectives* $\wedge, \vee, \supset, \neg$—whose intuitive meanings are "and," "or," "implies," and "not," respectively.[1]

We will be rather quick in this chapter and the next, since the intuitions and the structure of the theories are the same as in Chap. 2. Note also that from now on we will not express explicitly the distinction between the object level and the meta level (cf. §2.1.4). We will most of the time let a variable x stand for a metavariable $\mathbf{x}$, etc.; however this does not mean that the distinction is gone. It is still there; we just, due to our laziness, do not use different symbols any longer.

3.1 Propositional Logic: Formula

Atomic statements are expressed as a *propositional variable*, which we denote usually by P, Q, R, P_1, P_2, etc.

3.1.1 Definition (The set **PVar**)**.** Henceforth we fix a countably infinite set **PVar** of *propositional variables*.

3.1.2 Definition ((Propositional) formula)**.** The set of *propositional formulas* (also called *formulas of propositional logic* or simply *formulas*)[2] is defined

[1] Precisely speaking, of course, the symbols $\wedge, \vee, \supset, \neg$ are syntactic entities and their *meanings* (such as "and") are determined only when their semantics is defined.

[2] The word "formulae" is also used as a plural form of "formula."

inductively by the following rules.

$$\frac{P \in \mathbf{PVar}}{P \text{ is a formula}}\ (\textsc{Var})$$

$$\frac{A \text{ is a formula} \quad B \text{ is a formula}}{A \wedge B \text{ is a formula}}\ (\wedge) \qquad \frac{A \text{ is a formula} \quad B \text{ is a formula}}{A \vee B \text{ is a formula}}\ (\vee)$$

$$\frac{A \text{ is a formula} \quad B \text{ is a formula}}{A \supset B \text{ is a formula}}\ (\supset) \qquad \frac{A \text{ is a formula}}{\neg A \text{ is a formula}}\ (\neg)$$

We denote the set of propositional formulas by **PFml**.

Equivalently, in a BNF notation:

$$\mathbf{PFml} \ni A \quad ::= \quad P \in \mathbf{PVar} \mid A \wedge A \mid A \vee A \mid A \supset A \mid \neg A \ .$$

The symbols $\wedge, \vee, \supset, \neg$ are called *logical connectives*; their names are *conjunction*, *disjunction*, *implication* and *negation*, respectively. We use metavariables $A, B, C, \ldots$ for formulas.

3.1.3 Notation (Omission of parentheses)**.** We let the binding strength of the logical connectives to be

$$\neg \quad > \quad \begin{pmatrix} \wedge \\ \vee \end{pmatrix} \quad > \quad \supset \ ,$$

that is for example, $\neg P \wedge Q \supset C$ means $((\neg P) \wedge Q) \supset C$.[3]

We let $\supset$ associate from the right. That is, $A \supset B \supset C$ is short for $A \supset (B \supset C)$, not $(A \supset B) \supset C$. This is like in functional programming—think of $\supset$ as $\to$ for function types. The correspondence between $\supset$ (in logic) and $\to$ (in type theory) can in fact be made into a formal one. This is called the *Curry-Howard correspondence*; it is not only foundational in functional programming but also used for techniques like *program extraction*, a notable tool supporting which is Coq.

3.1.4 Definition (Free variable)**.** For each formula $A \in \mathbf{PFml}$, the set $\mathrm{FV}(A)$ of *free variables* of A is defined as follows.

$$\begin{aligned} \mathrm{FV}(P) &:= \{P\} \qquad \text{if } P \in \mathbf{PVar}, \\ \mathrm{FV}\big(A \wedge B\big) &:= \mathrm{FV}(A) \cup \mathrm{FV}(B) \ , \\ \mathrm{FV}\big(A \vee B\big) &:= \mathrm{FV}(A) \cup \mathrm{FV}(B) \ , \\ \mathrm{FV}\big(A \supset B\big) &:= \mathrm{FV}(A) \cup \mathrm{FV}(B) \ , \\ \mathrm{FV}\big(\neg A\big) &:= \mathrm{FV}(A) \ . \end{aligned}$$

We will use the following abbreviation conventions later.

3.1.5 Notation ($\bigwedge\Gamma, \bigvee\Gamma\,; \top, \bot$)**.** Let $\Gamma \equiv A_1, \ldots, A_m$ be a finite sequence of formulas. We let

$$\begin{aligned} (\cdots(A_1 \wedge A_2) \wedge \cdots) \wedge A_m \quad &\text{be abbreviated by} \quad \bigwedge \Gamma \ ; \\ (\cdots(A_1 \vee A_2) \vee \cdots) \vee A_m \quad &\text{be abbreviated by} \quad \bigvee \Gamma \ . \end{aligned}$$

[3] Or, more precisely, the abstract syntax tree designated by this expression.

If $m = 0$ (i.e. Γ is the empty sequence): we define $\bigwedge \Gamma$ to be $P \supset P$ for some fixed propositional variable P; and $\bigvee \Gamma$ to be $\neg(P \supset P)$. We also write

$$\begin{array}{ll} \top & (\text{“top”}) \qquad\qquad \text{for } P \supset P \ ; \\ \bot & (\text{“bottom”}) \qquad\qquad \text{for } \neg(P \supset P) \ . \end{array}$$

The formula $\top \equiv P \supset P$ is a formula that is “always true”; $\bot \equiv \neg(P \supset P)$ is “always false.”

3.2 Propositional Logic: Derivation Rule

We shall now do to propositional logic what we did in §2.3 to equational logic, i.e. introduce a system of derivation rules.

For propositional/predicate logic, there are several “styles” of derivation systems that are known to be equivalent.

- **Hilbert style**, featuring a lot of axioms (i.e. rules without premises) and only a couple of rules with axioms. Often favored by philosophers. Via the Curry-Howard correspondence, this style of logic corresponds to *combinatory logic* (where one sees *combinators* like S, K).
- **Natural deduction**, originally due to Gentzen. You will be (probably) learning about this in the exercise course. The Curry-Howard correspondence establishes the connection between this formalism and simply-typed λ-calculus, therefore it is favored by type theorists.
- **Sequent calculus** which we will be using in this course. It is also due to Gentzen. In doing *proof theory*—mathematics of proofs, or more precisely, mathematics of proof trees—this is probably the most convenient style.
- **Tableau method** which can be understood as a close, semantics-oriented variant of sequent calculus.

In sequent calculus, what is derived is not a single formula but is a construct called a *sequent*. This small—but ingenious—extension makes many arguments much simpler.

3.2.1 Definition (Sequent)**.** A *sequent* is two finite sequences of formulas, separated by a delimiting symbol $\Rightarrow$. That is,

$$A_1, \ldots, A_m \Rightarrow B_1, \ldots, B_n \ . \tag{3.1}$$

Note here that the both sides $A_1, \ldots, A_m$ and $B_1, \ldots, B_n$ are *sequences* of formulas, therefore: 1) the order matters (distinguish A, B, C and A, C, B); 2) the multiplicity matters (distinguish A, B, B and A, B).[4] Also note that m and n can be 0.

The informal “meaning” of the sequent (3.1) is:

> if all of $A_1, \ldots, A_m$ are true, then at least one of $B_1, \ldots, B_n$ is true;

[4] In fact in the current setting of *classical logic*, we could have considered *sets* of formulas instead of sequences. This is not the case for different kinds of logics like *linear logic*.

that is, $\bigwedge_i A_i \supset \bigvee_j B_j$. Later in defining semantics we make this intuition precise.

The following derivation system for propositional logic—in the style of sequent calculus—is historically called (propositional) *LK*. This is short for "logistischer klassischer Kalkül."

3.2.2 Definition (Derivation rules of propositional LK)**.** The *derivation rules* for propositional LK are presented in Fig. 3.1. There $\Gamma, \Delta, \Pi, \Sigma, \ldots$ are metavariables that stand for sequences of formulas.

Initial sequents

$$\frac{}{A \Rightarrow A}\ (\textsc{Init})$$

Structural rules

$$\frac{\Gamma \Rightarrow \Delta}{A, \Gamma \Rightarrow \Delta}\ (\textsc{Weakening-L}) \qquad \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, A}\ (\textsc{Weakening-R})$$

$$\frac{A, A, \Gamma \Rightarrow \Delta}{A, \Gamma \Rightarrow \Delta}\ (\textsc{Contraction-L}) \qquad \frac{\Gamma \Rightarrow \Delta, A, A}{\Gamma \Rightarrow \Delta, A}\ (\textsc{Contraction-R})$$

$$\frac{\Gamma, A, B, \Gamma' \Rightarrow \Delta}{\Gamma, B, A, \Gamma' \Rightarrow \Delta}\ (\textsc{Exchange-L}) \qquad \frac{\Gamma \Rightarrow \Delta, A, B, \Delta'}{\Gamma \Rightarrow \Delta, B, A, \Delta'}\ (\textsc{Exchange-R})$$

$$\frac{\Gamma \Rightarrow \Delta, A \quad A, \Pi \Rightarrow \Sigma}{\Gamma, \Pi \Rightarrow \Delta, \Sigma}\ (\textsc{Cut})$$

Logical rules

$$\frac{A, \Gamma \Rightarrow \Delta}{A \wedge B, \Gamma \Rightarrow \Delta}\ (\wedge\text{-L1}) \qquad \frac{\Gamma \Rightarrow \Delta, A \quad \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \wedge B}\ (\wedge\text{-R})$$

$$\frac{B, \Gamma \Rightarrow \Delta}{A \wedge B, \Gamma \Rightarrow \Delta}\ (\wedge\text{-L2})$$

$$\frac{\Gamma \Rightarrow \Delta, A}{\Gamma \Rightarrow \Delta, A \vee B}\ (\vee\text{-R1})$$

$$\frac{A, \Gamma \Rightarrow \Delta \quad B, \Gamma \Rightarrow \Delta}{A \vee B, \Gamma \Rightarrow \Delta}\ (\vee\text{-L}) \qquad \frac{\Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \vee B}\ (\vee\text{-R2})$$

$$\frac{\Gamma \Rightarrow \Delta, A \quad B, \Pi \Rightarrow \Sigma}{A \supset B, \Gamma, \Pi \Rightarrow \Delta, \Sigma}\ (\supset\text{-L}) \qquad \frac{A, \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \supset B}\ (\supset\text{-R})$$

$$\frac{\Gamma \Rightarrow \Delta, A}{\neg A, \Gamma \Rightarrow \Delta}\ (\neg\text{-L}) \qquad \frac{A, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \neg A}\ (\neg\text{-R})$$

Figure 3.1: Derivation rules of propositional LK

3.2.3 Definition (Proof tree; derivability)**.** A *proof tree* (also called a *derivation tree* or simply a *proof*) in propositional LK is

- a finite depth tree,
- each node of which is a legitimate instance of the derivation rules of propositional LK (Fig. 3.1).

A sequent $\Gamma \Rightarrow \Delta$ is said to be *derivable* (or *provable*) if there exists a proof tree Π whose root is $\Gamma \Rightarrow \Delta$. We denote this fact by $\vdash \Gamma \Rightarrow \Delta$.

A formula $A \in \mathbf{PFml}$ is *derivable* if the sequent

$$\Rightarrow A$$

(with the empty sequence on the left hand side) is derivable. We write $\vdash A$ for this.

Note that in LK, the rule (Init) is the only one that has no premises. Therefore in a proof tree, any leaf is an instance of (Init).

3.2.4 Example. Let $A, B \in \mathbf{PFml}$. Let us give a proof tree for $A \supset B \Rightarrow \neg(A \wedge \neg B)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow A}\,(\text{Init}) \quad \dfrac{}{B \Rightarrow B}\,(\text{Init})}{B, \neg B \Rightarrow}\,(\neg\text{-L})}{A \supset B, A, \neg B \Rightarrow}\,(\supset\text{-L})}{A \supset B, A \wedge \neg B, A \wedge \neg B \Rightarrow}\,(\wedge\text{-L1}),(\wedge\text{-L2})}{A \supset B, A \wedge \neg B \Rightarrow}\,(\text{Contraction-L})}{A \supset B \Rightarrow \neg(A \wedge \neg B)}\,(\neg\text{-R})$$

Here we suppressed the use of the (Exchange) rules.

In the previous example, one wonders if we could just use the following variant of the ($\wedge$-L) rules

$$\dfrac{A, B, \Gamma \Rightarrow \Delta}{A \wedge B, \Gamma \Rightarrow \Delta}\,(\wedge\text{-L'}) \tag{3.2}$$

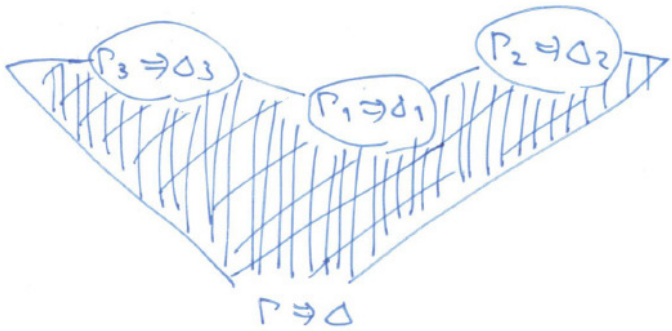
in place of the application of ($\wedge$-L1), ($\wedge$-L2) and (Contraction-L). This new rule is *admissible* in the following sense.

3.2.5 Definition (Admissible rule)**.** A rule

$$\dfrac{\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}$$

is said to be *admissible* if there is a "proof tree with premises $\Gamma_1 \Rightarrow \Delta_1, \ldots, \Gamma_n \Rightarrow \Delta_n$," that is,

- a finite depth tree,
- each node of which is either
  - a legitimate instance of the derivation rules of propositional LK (Fig. 3.1), or
  - a leaf whose label is one of the sequents $\Gamma_1 \Rightarrow \Delta_1, \ldots, \Gamma_n \Rightarrow \Delta_n$, and
- whose root is $\Gamma \Rightarrow \Delta$.



Thus an admissible rule is like a "macro."

The following lemma supports the intuition about a sequent $\Gamma \Rightarrow \Delta$ as $\bigwedge_i A_i \supset \bigvee_j B_j$.

3.2.6 Lemma. *The following are equivalent.*

1. $\vdash \Gamma \Rightarrow \Delta$.
2. $\vdash \bigwedge \Gamma \Rightarrow \bigvee \Delta$.
3. $\vdash \bigwedge \Gamma \supset \bigvee \Delta$.

Here $\bigwedge \Gamma$ *and* $\bigvee \Delta$ *are as in Notation 3.1.5.*

3.3 Propositional Logic: Semantics

We now define the "meaning" of the syntactic expressions in propositional logic. The expressions are formulas—atomic statements as propositional variables combined using logical connectives—therefore their meaning is whether *true* or *false*.

First, much like in Def. 2.5.2, we fix the meaning of the variables.

3.3.1 Definition (Valuation)**.** A *valuation* is a function

$$J \ : \ \mathbf{PVar} \longrightarrow \{\mathrm{tt}, \mathrm{ff}\}$$

that maps each propositional variable either to tt (for "true") or to ff (for "false").

A valuation is extended to the "meaning" of more complicated formulas. This is like in Def. 2.5.3.

3.3.2 Definition (Denotation)**.** Let $J : \mathbf{PVar} \to \{\mathrm{tt}, \mathrm{ff}\}$ be a valuation. For each formula A, we define its *denotation*

$$[\![A]\!]_J \quad \in \{\mathrm{tt}, \mathrm{ff}\}$$

as follows, inductively on the construction of A.

$$\begin{aligned}
[\![P]\!]_J = \mathtt{tt} &\overset{\text{def}}{\iff} J(P) = \mathtt{tt} && \text{if } P \in \mathbf{PVar}\\
[\![A \wedge B]\!]_J = \mathtt{tt} &\overset{\text{def}}{\iff} [\![A]\!]_J = \mathtt{tt} \text{ and } [\![B]\!]_J = \mathtt{tt}\\
[\![A \vee B]\!]_J = \mathtt{tt} &\overset{\text{def}}{\iff} [\![A]\!]_J = \mathtt{tt} \text{ or } [\![B]\!]_J = \mathtt{tt}\\
[\![A \supset B]\!]_J = \mathtt{tt} &\overset{\text{def}}{\iff} [\![A]\!]_J = \mathtt{ff} \text{ or } [\![B]\!]_J = \mathtt{tt}\\
[\![\neg A]\!]_J = \mathtt{tt} &\overset{\text{def}}{\iff} [\![A]\!]_J = \mathtt{ff}
\end{aligned}$$

Note the definition for $A \supset B$. It is unlike the word "implies" in the everyday sense. For example,

> if $0 = 1$, the world is mine

is true in the above "mathematical" sense of implication.

Truth table is a useful formalism in the semantics of propositional formulas. Its row corresponds to a possible choice of a valuation, i.e. possible assignment of truth values to the relevant propositional variables. For example, the following are the truth tables for the formulas $\neg P \vee Q$ and $P \supset Q$.

P	Q	$\neg P$	$\neg P \vee Q$
tt	tt	ff	tt
tt	ff	ff	ff
ff	tt	tt	tt
ff	ff	tt	tt

P	Q	$P \supset Q$
tt	tt	tt
tt	ff	ff
ff	tt	tt
ff	ff	tt

(3.3)

Validity is the word for being "always true." In propositional logic, for historical reasons, the word *tautology*—which means "saying the same thing twice"—is more commonly used.

3.3.3 Definition (Tautology)**.** A formula A is a *tautology* if, for any valuation $J : \mathbf{PVar} \to \{\mathtt{tt}, \mathtt{ff}\}$, we have $[\![A]\!]_J = \mathtt{tt}$.

How can one check if a given formula is tautology? The set **PVar** is countably infinite so the set of all valuations is of the cardinality $\aleph$. The following lemma says we can just use truth tables.

3.3.4 Lemma. *Let J, J' be two valuations, and A be a formula. Assume that*

$$J(P) \;=\; J'(P) \qquad \textit{for each } P \in \mathrm{FV}(A),$$

where $\mathrm{FV}(A)$ *is from Def. 3.1.4. Then we have*

$$[\![A]\!]_J \;=\; [\![A]\!]_{J'} \;.$$

Proof. By induction on the construction of a formula A. □

3.3.5 Definition (Satisfiability; logical equivalence)**.** A formula A is *satisfiable* if there exists a valuation J such that $[\![A]\!]_J = \mathtt{tt}$.

Formulas A and B are said to be *logically equivalent* if, for any valuation J, we have $[\![A]\!]_J = [\![B]\!]_J$. We write $A \cong B$ for logical equivalence.

3.3.6 Lemma. *1. A formula A is not satisfiable if and only if the formula $\neg A$ is a tautology.*

2. Formulas A and B are logically equivalent if and only if the formula $(A \supset B) \wedge (B \supset A)$ is a tautology. □

Finally, the denotation of a sequent is defined.

3.3.7 Definition. Let $\Gamma \Rightarrow \Delta$ be a sequent. Its *denotation* $[\![\Gamma \Rightarrow \Delta]\!]_J$ under a valuation J is defined by

$$[\![\Gamma \Rightarrow \Delta]\!]_J \quad := \quad [\![\bigwedge \Gamma \supset \bigvee \Delta]\!]_J \ ,$$

where the formulas $\bigwedge \Gamma, \bigvee \Delta$ are as in Notation 3.1.5.

A sequent $\Gamma \Rightarrow \Delta$ is *valid* if the formula $\bigwedge \Gamma \supset \bigvee \Delta$ is a tautology. We write $\models \Gamma \Rightarrow \Delta$.

In particular: the sequent

$$A \Rightarrow$$

means $A \supset \neg(P \supset P)$ (Notation 3.1.5), that is, $\neg A$.

Using the following result we can "push negation inwards." The first two are commonly called the *de Morgan law*.

3.3.8 Proposition. *Let A, B be formulas. We have the following logical equivalences.*

$$\neg(A \wedge B) \cong \neg A \vee \neg B \qquad \neg(A \vee B) \cong \neg A \wedge \neg B$$
$$\neg(A \supset B) \cong A \wedge \neg B \qquad \neg\neg A \cong A$$ □

3.4 Propositional Logic: Syntax vs. Semantics

The situation is the same as in §2.6: we now look at the soundness and completeness properties of propositional LK. We do get completeness.

Soundness is again by induction.

3.4.1 Theorem (Soundness)**.** $\vdash \Gamma \Rightarrow \Delta$ *implies* $\models \Gamma \Rightarrow \Delta$. □

Note: unless you do hardcore theoretical work, *most* of the proofs that you write in computer science will be by induction. Familiarize yourself by doing Exercise 3.12!

3.4.2 Corollary. *For any formula A, $\vdash A$ implies that A is a tautology.* □

3.4.3 Theorem (Completeness)**.** $\models \Gamma \Rightarrow \Delta$ *implies* $\vdash \Gamma \Rightarrow \Delta$.

The proof strategy is the same as in Chap. 2: given an underivable formula, we construct a *counter model* that makes the formula false. In propositional logic, a model is a valuation; and we again use syntactic ingredients to construct a counter model.

We use the following notions.

3.4.4 Definition (Consistent pair)**.** Let (U, V) be a pair of sets of formulas (i.e. $U, V \subseteq \mathbf{PFml}$). The pair (U, V) is said to be *consistent* if: for any finite sequences Γ from U and Δ from V, $\nvdash \Gamma \Rightarrow \Delta$.

The intuition is:

- there is a guy (say Charlie) who claims that
  - the formulas in U are true and
  - those in V are false;
- and the deduction system LK tries to "detect Charlie's lie" by proving $\vdash \Gamma \Rightarrow \Delta$ for some Γ from U and Δ from V. If LK succeeds in derivation, then it means (by soundness) that some formula in Δ must be true (or some in Γ is false), revealing that Charlie is lying. Consistency asserts that LK does not succeed.

3.4.5 Definition (Maximally consistent pair)**.** A *maximally consistent pair* is a consistent pair (U, V) such that, for each $A \in \mathbf{PFml}$, $A \in U$ or $A \in V$.

In a maximally consistent pair, Charlie has a say for any formula—you give him any formula and he would claim either that it is true, or that it is false.

3.4.6 Lemma. *Let (U, V) be a consistent pair.*

1. *The sets U and V have no formula in common: $U \cap V = \emptyset$.*
2. *The pair (U, V) can be extended to a maximally consistent pair. That is, there are $U', V' \subseteq \mathbf{PFml}$ such that $U \subseteq U'$, $V \subseteq V'$ and (U', V') is maximally consistent.*

Proof. 1. Assume not; then there is $A \in U \cap V$. Take $\Gamma \equiv \Delta \equiv A$; then the sequent $\Gamma \Rightarrow \Delta$ is derivable (obvious from the (Init) rule). This contradicts with consistency.

2. There are only countably many formulas in **PFml**; therefore we can take an enumeration $A_0, A_1, A_2, \dots$. That is,

$$\{A_0, A_1, A_2, \dots\} \;=\; \mathbf{PFml} \;.$$

Let $U_0 := U$ and $V_0 := V$; for each $i \in \mathbb{N}$, we inductively add A_i to either U_i or V_i and obtain a new consistent pair (U_{i+1}, V_{i+1}).

3.4.7 Sublemma. Let $i \in \mathbb{N}$; assume (U_i, V_i) is a consistent pair. Then at least one of

$$\big(U_i \cup \{A_i\},\, V_i\big) \quad \textit{or} \quad \big(U_i,\, V_i \cup \{A_i\}\big)$$

is consistent.

Proof. (Of Sublem. 3.4.7) Assume not. By the definition of consistent pair, we can take four finite sequences

$$\Gamma \text{ from } U_i \cup \{A_i\} \;,\quad \Delta \text{ from } V_i \;; \qquad \Pi \text{ from } U_i \;,\quad \Sigma \text{ from } V_i \cup \{A_i\} \;;$$

such that $\vdash \Gamma \Rightarrow \Delta$ and $\vdash \Pi \Rightarrow \Sigma$. We observe that Γ necessarily contains A_i: otherwise, $\vdash \Gamma \Rightarrow \Delta$ proves inconsistency of (U_i, V_i), violating the assumption. Therefore we can write

$$\Gamma \;\equiv\; \Gamma', A_i, \Gamma'' \;.$$

Similarly, Σ contains A_i and we set $\Sigma \equiv \Sigma', A_i, \Sigma''$.

Since $\vdash \Gamma', A_i, \Gamma'' \Rightarrow \Delta$ and $\vdash \Pi \Rightarrow \Sigma', A_i, \Sigma''$, by the (Cut) rule we see that $\vdash \Pi, \Gamma', \Gamma'' \Rightarrow \Sigma', \Sigma'', \Delta$ (see below).

$$\dfrac{\dfrac{\vdots}{\Pi \Rightarrow \Sigma', A_i, \Sigma''}\ (\textsc{Exchange-R}) \quad \dfrac{\vdots}{\Gamma', A_i, \Gamma'' \Rightarrow \Delta}\ (\textsc{Exchange-L})}{\Pi, \Gamma', \Gamma'' \Rightarrow \Sigma', \Sigma'', \Delta}\ (\textsc{Cut})$$

This proves inconsistency of (U_i, V_i), which is a contradiction. □

We turn back to the proof of Lem. 3.4.6.2. By Sublem. 3.4.7, an inductive construction of consistent pairs

$$(U_0, V_0)\ , \quad (U_1, V_1)\ , \quad \dots$$

is indeed possible. (Note it is not unique; you may be able to choose which of U_i or V_i you add A_i to) By construction, for each $i \in \mathbb{N}$:

- (U_i, V_i) is consistent;
- $U_i \subseteq U_{i+1}$ and $V_i \subseteq V_{i+1}$; and
- $A_i \in U_{i+1} \cup V_{i+1}$.

Now we define

$$U' := \bigcup_{i \in \mathbb{N}} U_i\ , \quad V' := \bigcup_{i \in \mathbb{N}} V_i\ ,$$

and claim that (U', V') is a maximally consistent pair. Any formula A_i belongs either to U' or V' because $A_i \in U_{i+1} \cup V_{i+1}$.

3.4.8 Sublemma. The pair (U', V') is consistent.

Proof. (Of Sublem. 3.4.8) Assume it is not consistent; then we can take $B_1, \dots, B_m$ from U' and $C_1, \dots, C_n$ from V' such that

$$\vdash B_1, \dots, B_m \Rightarrow C_1, \dots, C_n\ .$$

Since there are only finitely many formulas in $B_1, \dots, B_m$ and $C_1, \dots, C_n$, and that U_i and V_i are increasing, for a sufficiently large $k \in \mathbb{N}$ we have

$$B_1, \dots, B_m \in U_k \quad \text{and} \quad C_1, \dots, C_n \in V_k\ .$$

Therefore the pair (U_k, V_k) is inconsistent; this is a contradiction. □

Finally, $U \subseteq U'$ and $V \subseteq V'$ follow from $U = U_0$ and $V = V_0$. This concludes the proof of Lem. 3.4.6. □

A maximally consistent pair can be identified with a valuation. To show that, we need the following facts.

3.4.9 Lemma. *Let (U', V') be a maximally consistent pair.*

1. *$A \wedge B \in U'$ if and only if $A \in U'$ and $B \in U'$.*
2. *$A \vee B \in U'$ if and only if ($A \in U'$ or $B \in U'$).*

3. $A \supset B \in U'$ *if and only if* ($A \notin U'$ *or* $B \in U'$).

4. $\neg A \in U'$ *if and only if* $A \notin U'$, *that is,* $A \in V'$.

Proof. Use logical rules. □

3.4.10 Lemma. *Let* (U', V') *be a maximally consistent pair; and let us define a valuation* $J : \mathbf{PVar} \to \{\mathrm{tt}, \mathrm{ff}\}$ *by*

$$J(P) = \mathrm{tt} \quad \overset{\text{def}}{\Longleftrightarrow} \quad P \in U' \ .$$

Then we have, for any formula A,

$$[\![A]\!]_J = \mathrm{tt} \quad \Longleftrightarrow \quad A \in U' \ .$$

Proof. By induction on the construction of a formula A, using Lem. 3.4.9. □

Finally:

Proof. (Of Thm. 3.4.3) Assume $\nvdash \Gamma \Rightarrow \Delta$. Let U be the set of formulas in the sequence Γ; V be those in Δ. Then (U, V) is a consistent pair. By Lem. 3.4.6.2, we can extend it to a maximally consistent pair (U', V'). Define a valuation $J : \mathbf{PVar} \to \{\mathrm{tt}, \mathrm{ff}\}$ by

$$J(P) = \mathrm{tt} \quad \overset{\text{def}}{\Longleftrightarrow} \quad P \in U' \ .$$

Then by Lem. 3.4.10 we have

$$[\![A]\!]_J = \mathrm{tt} \quad \text{for any } A \in U'; \quad \text{and} \quad [\![B]\!]_J = \mathrm{ff} \quad \text{for any } B \in V'.$$

Therefore, under J, all the formulas in Γ are true and all in Δ are false. Thus $[\![\Gamma \Rightarrow \Delta]\!]_J = \mathrm{ff}$; hence $\not\models \Gamma \Rightarrow \Delta$. □

To summarize the proof: an underivable sequent $\Gamma \Rightarrow \Delta$ is extended to a maximally consistent pair; it is then used to induce a counter model J. This completeness proof in fact has a strong tableau method flavor. Its merit is that it generalizes smoothly to other kinds of logics (predicate, modal, intuitionistic, etc.).

Exercises

3.1. In LK, restricting the rule (Init) to

$$\frac{P \in \mathbf{PVar}}{P \Rightarrow P} \ (\text{Init'})$$

does not lessen the set of derivable formulas. Prove this fact.

3.2. Give proof trees for the following sequents.

1. $A \Rightarrow \neg\neg A$
2. $\neg\neg A \Rightarrow A$
3. $\Rightarrow A \supset B \supset A$

4. $(A \supset B) \supset B \supset C \Rightarrow A \supset B \supset C$
5. $\Rightarrow (A \wedge B) \supset \neg(\neg A \vee \neg B)$

3.3. Show that the rule ($\wedge$-L') in (3.2) is admissible.

3.4. Prove Lem. 3.3.6.

3.5. Prove that each of the following sets of formulas are logically equivalent.

1. $P \supset Q$, $(\neg P) \vee Q$, and $\neg(P \wedge (\neg Q))$
2. $(P \wedge Q) \supset R$ and $P \supset Q \supset R$

3.6. For each of the following formulas, decide if it is valid/satisfiable/unsatisfiable.

1. $A \supset B \supset A$
2. $(A \supset B) \supset B$
3. $\neg A \supset A$
4. $((P \supset Q) \supset P) \supset P$ (*Peirce's law*)

3.7. Although illegitimate, it most of the time causes no problem if we write $A \wedge B \wedge C$ or $A \vee B \vee C$. Why? Argue from the semantic point of view.

3.8. The set $\{\wedge, \neg\}$ is *functionally complete*: using these two we can "encode" any logical connective. Give formulas that use only these connectives and are equivalent to the following formulas.

$$A \supset B \qquad A \vee B$$

What about other subsets of $\{\wedge, \vee, \neg, \supset\}$?

3.9. Are

$$\not\models A \quad \text{and} \quad \models \neg A$$

equivalent? Why (not)? How about the following?

$$\nvdash A \quad \text{and} \quad \vdash \neg A$$

3.10 (Conjunctive normal form; disjunctive normal form)**.** These normal forms play an important role in complexity theory, where a negative literal $\neg P$ is often denoted by $\overline{P}$.

A *literal* is either a propositional variable $P \in \mathbf{PVar}$ or its negation $\neg P$ (with $P \in \mathbf{PVar}$). A propositional formula A is in *conjunctive normal form (CNF)* if it is a conjunction of disjunctions of literals, that is,

$$A \equiv \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} L_{ij} \ ,$$

where L_{ij} is a literal. A propositional formula A is in *disjunctive normal form (DNF)* if it is a disjunction of conjunctions of literals, that is,

$$A \equiv \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} L_{ij} \ .$$

1. Convert the following formulas to CNF/DNF, preserving logical equivalence.
$$P \supset Q \quad (P \supset Q) \supset R \supset P \wedge (Q \vee \neg R)$$
2. (Might be hard) Prove that, for any formula A, there are formulas in CNF/DNF that are logically equivalent to A.
3. Conversion to DNF can also be semantically. Give a truth table for the formula $\neg(P \supset (Q \wedge R))$; and derive a DNF of the formula.

3.11. Among the following symbols in this chapter, which are meta level ones?
$$P, \quad \models, \quad \wedge, \quad \Rightarrow, \quad \vdash, \quad A$$

3.12. Prove Thm. 3.4.1.

3.13. Prove Lem. 3.4.9.

3.14. Prove Lem. 3.4.10.

3.15. Give another proof of Lem. 3.4.6.2 using *Zorn's lemma*.

Chapter 4

Predicate Logic

Predicate logic is an extension of propositional logic, where

- aside from formulas (expressing statements), in syntax we have *terms* that express *individuals* (such as "Alice," "Bob," "Bob's father," "Alice and Bob's first child," "the natural number 0," etc.); and
- we have additional connectives $\forall$ ("for all," *universal quantification*) and $\exists$ ("there exists," *existential quantification*). These are called *quantifiers*.

4.1 Predicate Logic: Term and Formula

The syntax of predicate logic depends of the following parameters: *function symbols* and *predicate symbols*. Both of these are given in the form of signatures (Def. 2.2.1)—i.e. each function/predicate symbol comes with a (fixed) arity $n \in \mathbb{N}$. Convention: we write $f, g, h, \ldots$ for function symbols and $P, Q, R, \ldots$ for predicate symbols. The signatures for function and predicate symbols are denoted by **FnSymb** and **PdSymb**, respectively.

4.1.1 Example. – 「海老蔵と海老蔵の父は共演したことがある」:

$$P(c, f(c)) \ , \qquad \text{ここで,} \quad \begin{cases} P(x,y): & \text{「}x, y \text{ は共演したことがある」} \\ f(x): & \text{「}x \text{ の父」} \\ c: & \text{「海老蔵」} \end{cases}$$

- 「共演したことがあるという関係は対称的」:

$$\forall x.\, \forall y.\, \big(P(x,y) \supset P(y,x) \big) \ .$$

- 「役者にはかならず共演者がいる」:

$$\forall x.\, \exists y.\, P(x,y) \ .$$

- 「すべての役者と共演したことのある大役者がいる」:

$$\exists x.\, \forall y.\, P(x,y) \ .$$

We proceed to the formal definition of syntax. Firstly, variables in predicate logic range over individuals (like "Alice," "Bob" and 「海老蔵」.) This is much like in equational logic; and is unlike in propositional logic where a propositional variable is an atomic statement.

4.1.2 Definition (The set **Var**)**.** Henceforth we fix a countably infinite set **Var** of *variables*.

4.1.3 Definition (Term; formula)**.** Let **FnSymb** and **PdSymb** be signatures; we assume that they are disjoint. The set of *terms* over **FnSymb**, denoted by **Terms**,[1] is defined inductively by the following BNF notation.

$$\mathbf{Terms} \ni t_1, \ldots, t_n \quad ::= \quad x \in \mathbf{Var} \mid f(t_1, \ldots, t_n) \ ,$$

where $f \in \mathbf{FnSymb}$ is a function symbol which is n-ary.

The set of (predicate) *formulas* over **FnSymb** and **PdSymb**, denoted by **Fml**, is defined inductively by the following BNF notation.

$$\begin{aligned} \mathbf{Fml} \ni A \quad ::= \quad & P(t_1, \ldots, t_n) \mid A \wedge A \mid A \vee A \mid A \supset A \mid \neg A \\ & \mid \forall x.\, A \mid \exists x.\, A \ , \end{aligned}$$

where $t_1, \ldots, t_n \in \mathbf{Terms}$ are terms over **FnSymb**; $P \in \mathbf{PdSymb}$ is a predicate symbol of arity n; and x is a variable.

A formula in the form $P(t_1, \ldots, t_n)$ is said to be *atomic*; one in the form $\forall x.\, A$ is *universally quantified*; and $\exists x.\, A$ is *existentially quantified*.

4.1.4 Remark. You are invited to spot metavariables in Def. 4.1.3 (we have been lazy in using different fonts).

4.1.5 Notation (Omission of parentheses)**.** We let quantifiers $\forall x.$ and $\exists x.$ bind stronger than the other connectives.

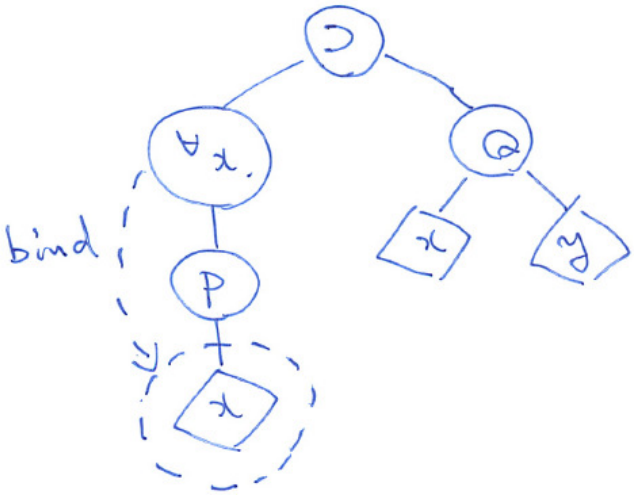
Mathematically/structurally speaking, the current syntax is a much more complicated one than those for equational/propositional logics. The distinguishing feature is *variable binder*: the quantifiers $\forall x.$ and $\exists x$ *binds* certain occurrences of the variable x. This is like *scoping* in programming.

4.1.6 Definition (Free/bound occurrence)**.** Let A be a formula. An occurrence of a variable x in A is said to be *bound* if it is inside a quantifier $\forall x.$ or $\exists x.$ An occurrence that is not bound is said to be *free*.

4.1.7 Example. In a formula $(\forall x.P(x)) \supset Q(x, y)$,

- the first occurrence of x is bound;
- the second occurrence of x and the occurrence of y are both free.

[1] To be precise this should be denoted by something like **Terms**(**FnSymb**); we do not do so for readability.



4.1.8 Definition (Free variable)**.** A variable x is *free* in a formula A if it has a free occurrence in A.

To put it more precisely: for each term $t \in \mathbf{Terms}$, the set $\mathrm{FV}(t)$ of *free variables* of t is defined as follows.

$$\begin{aligned} \mathrm{FV}(x) &:= \{x\} \qquad \text{if } x \in \mathbf{Var}; \\ \mathrm{FV}\big(f(t_1,\dots,t_n)\big) &:= \mathrm{FV}(t_1) \cup \cdots \cup \mathrm{FV}(t_n) \ . \end{aligned}$$

Here $f \in \mathbf{FnSymb}$ is an n-ary function symbol.

For each formula $A \in \mathbf{Fml}$, the set $\mathrm{FV}(A)$ of *free variables* of A is defined as follows.

$$\begin{aligned} \mathrm{FV}\big(P(t_1,\dots,t_n)\big) &:= \mathrm{FV}(t_1) \cup \cdots \cup \mathrm{FV}(t_n) \ , \\ \mathrm{FV}\big(A \wedge B\big) &:= \mathrm{FV}(A) \cup \mathrm{FV}(B) \ , \\ \mathrm{FV}\big(A \vee B\big) &:= \mathrm{FV}(A) \cup \mathrm{FV}(B) \ , \\ \mathrm{FV}\big(A \supset B\big) &:= \mathrm{FV}(A) \cup \mathrm{FV}(B) \ , \\ \mathrm{FV}\big(\neg A\big) &:= \mathrm{FV}(A) \ , \\ \mathrm{FV}\big(\forall x.\, A\big) &:= \mathrm{FV}(A) \setminus \{x\} \ , \\ \mathrm{FV}\big(\exists x.\, A\big) &:= \mathrm{FV}(A) \setminus \{x\} \ . \end{aligned}$$

A *closed formula* is a formula that has no free variables.

Bound variables/variable binders are tricky especially in relation to substitution.[2]

4.1.9 Definition (α-equivalence)**.** Two formulas A, B are said to be *α-equivalent* if they are the same except for renaming of the bound variables with fresh variables.

[2] We circumvent the trickiness by being informal—you can try to make the subsequent definitions totally formal and see how awkward the task is! This becomes a real problem when you implement logic for a theorem prover/proof assistant. Some theoretical gadgets to deal with it: *de Bruijn index*; *presheaf category*; *nominal set*.

The intuition is: it is the link between a bound (occurrence of a) variable and a variable binder that matters. The name of a bound variable is just an "alias" for this link; therefore renaming it (causing α-equivalence) results in an "equivalent" formula.

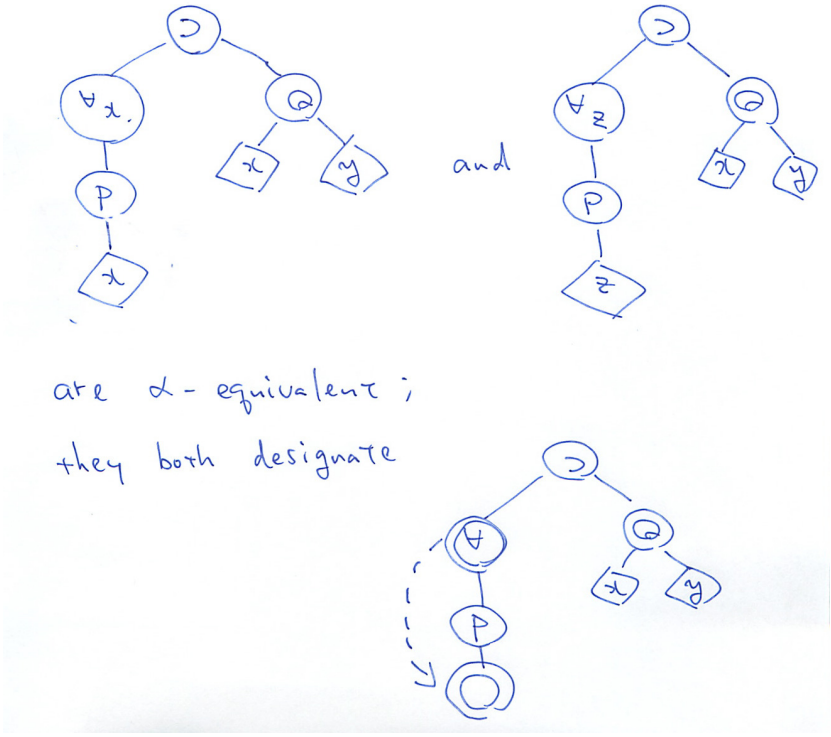
4.1.10 Notation. In what follows, we regard two α-equivalent formulas to be *syntactically equal*. Therefore we have, for example,

$$\forall x.\, P(x) \equiv \forall y.\, P(y) \ .$$

4.1.11 Example. The formulas

$$(\forall x.P(x)) \supset Q(x,y) \ , \quad (\forall y.P(y)) \supset Q(x,y) \quad \text{and} \quad (\forall z.P(z)) \supset Q(x,y)$$

are all mutually α-equivalent.



However,

$$(\forall x.Q(x,y)) \quad \text{and} \quad (\forall y.Q(y,y))$$

are *not* α-equivalent—for renaming we must use a *fresh* (i.e. occurring nowhere) variable.

In the presence of variable binders, a careless substitution causes an undesired "crush" of variables—as is familiar to you in a programming language with variable scoping. Consider substituting $f(x)$ for y in $\forall x.\, R(x,y)$. The variable x in $f(x)$ has nothing to do with that in $\forall x.\, R(x,y)$—after all the latter formula is "the same thing" as $\forall w.\, R(w,y)$. However, by simple replacement we obtain a formula $\forall x.\, R(x, f(x))$, where the last x is undesirably *captured* by a quantifier.

It is therefore customary to use *capture-avoiding substitution*.

4.1.12 Definition (Capture-avoiding substitution)**.** Let $x \in \mathbf{Var}$ be a variable, $t \in \mathbf{Terms}$ be terms and A be a formula. By $A[t/x]$ we denote *capture-avoiding substitution*: we replace every *free* occurrence of x in A with t; but, in case it results in capture of a variable in t, we suitably rename bound variables.

4.1.13 Example. We have

$$\big(\forall x.R(x,y)\big)[f(x)/y] \equiv \big(\forall w.R(w,f(x))\big) \ .$$

Note that this is further (syntactically) equal to $\forall z.R(z,f(x))$ (Notation 4.1.10).

4.1.14 Remark. The syntax we presented is that of the *first-order* predicate logic; in it there is only one level of individuals that are quantified. In the *second-order* predicate logic, a predicate symbol can also be quantified, resulting e.g. in a formula

$$\forall x.\,(P(x) \vee Q(x)) \ \supset \ \forall R.\,\forall x.\,\big(\,(P(x) \supset R(x)) \supset (Q(x) \supset R(x)) \supset R(x)\,\big) \ .$$

4.2 Predicate Logic: Derivation Rule

A derivation system for predicate logic can be given as an extension of one for propositional logic. Here we present *predicate LK*, that is an extension of propositional LK (§3.2). Other styles of derivation systems—Hilbert style, natural deduction, etc.—are also possible.

The definition of sequent is the same.

4.2.1 Definition (Sequent)**.** A *sequent* is two finite sequences of formulas in **Fml**, separated by a delimiting symbol $\Rightarrow$. That is,

$$A_1, \ldots, A_m \Rightarrow B_1, \ldots, B_n \ . \tag{4.1}$$

4.2.2 Definition (Derivation rules of predicate LK)**.** The *derivation rules* for predicate LK are

- the same rules as in propositional LK (Fig. 3.1), augmented with
- the following rules for quantifiers.

$$\frac{A[t/x], \Gamma \Rightarrow \Delta}{\forall x.\,A, \Gamma \Rightarrow \Delta}\ (\forall\text{-L}) \qquad \frac{\Gamma \Rightarrow \Delta, A[z/x]}{\Gamma \Rightarrow \Delta, \forall x.A}\ (\forall\text{-R}),\ (\text{VC})$$

$$\frac{A[z/x], \Gamma \Rightarrow \Delta}{\exists x.\,A, \Gamma \Rightarrow \Delta}\ (\exists\text{-L}),\ (\text{VC}) \qquad \frac{\Gamma \Rightarrow \Delta, A[t/x]}{\Gamma \Rightarrow \Delta, \exists x.\,A}\ (\exists\text{-R})$$

Here the two rules with (VC) come with the following side condition:

> **Eigenvariable condition (VC)**: the variable z does not have any free occurrence in the sequent on the bottom.

For a deduction rule, it is very important that the legitimacy of the rule's application can be checked "easily." This often means "syntactically" (*recursively* to be general and precise—we will come back to this later). For example, a "rule"

$$\frac{}{\Rightarrow A}\ (\text{If } A \text{ is a valid formula})$$

derives all the valid formulas. But this is obviously cheating: we cannot "easily" or "effectively" check the side condition. In contrast, all the rules that we have seen can be applied "easily." The new rules for quantifiers are not exceptions: the side condition (VC) is a syntactic condition and can be checked in a straightforward manner.

4.2.3 Example. Assume x does not freely occur in A. Here is an LK proof for the sequent $\forall x.(A \supset B) \Rightarrow A \supset \forall x.B$.[3]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow A}\,(\textsc{Init}) \quad \dfrac{}{B \Rightarrow B}\,(\textsc{Init})}{A \supset B, A \Rightarrow B}\,(\supset\text{-L})}{\forall x.(A \supset B), A \Rightarrow B}\,(\forall\text{-L})}{\forall x.(A \supset B), A \Rightarrow \forall x.B}\,(\forall\text{-R}),\ (\text{VC})}{\forall x.(A \supset B) \Rightarrow A \supset \forall x.B}\,(\supset\text{-R})$$

In the application of ($\forall$-L), we took x as t in the rule (Def. 4.2.2); in the application of ($\forall$-R), we took x as z in the rule (Def. 4.2.2). Since x does not occur freely in A, it does not occur freely in the sequent $\forall x.(A \supset B), A \Rightarrow \forall x.B$ either, satisfying (VC).

In contrast, the following is *not* a correct proof tree for the sequent $\forall x.\, \exists y.\, R(x,y) \Rightarrow \exists y.\, \forall x.\, R(x,y)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{R(x,y) \Rightarrow R(x,y)}\,(\textsc{Init})}{R(x,y) \Rightarrow \forall x.\, R(x,y)}\,(\forall\text{-R}),\ (\text{VC})}{R(x,y) \Rightarrow \exists y.\, \forall x.\, R(x,y)}\,(\exists\text{-R})}{\exists y.\, R(x,y) \Rightarrow \exists y.\, \forall x.\, R(x,y)}\,(\exists\text{-L}),\ (\text{VC})}{\forall x.\, \exists y.\, R(x,y) \Rightarrow \exists y.\, \forall x.\, R(x,y)}\,(\forall\text{-L}) \tag{4.2}$$

Why not? (Exercise 4.3) After all, is this sequent valid? (See §4.3 later).

4.2.4 Definition (Proof tree; derivability)**.** The same as in propositional logic (Def. 3.2.3).

4.3 Predicate Logic: Semantics

The scenario is the same: we fix a *model*—a basic set of data that specifies the "meaning" of atomic expressions—and extend it to the "meaning" of more complicated terms and formulas.

The name *structure* is a historical one. It is the notion of model for predicate logic.

4.3.1 Definition ((First-order) structure)**.** A *(first-order) structure* $\mathbb{S}$ over **FnSymb** and **PdSymb** is a tuple

$$\mathbb{S} = \left(U,\ \left([\![f]\!]_{\mathbb{S}} \right)_{f \in \mathbf{FnSymb}},\ \left([\![P]\!]_{\mathbb{S}} \right)_{P \in \mathbf{PdSymb}} \right) \ ,$$

where

[3] Say: A means "it is fine today"; B means "x is happy."

– U is a nonempty[4] set called *domain* (or *universe*);

– for each $f \in \mathbf{FnSymb}_n$, $[\![f]\!]_{\mathbb{S}} : U^n \to U$ is the *interpretation* of f; and

– for each $P \in \mathbf{PdSymb}_n$, $[\![P]\!]_{\mathbb{S}} : U^n \to \{\mathrm{tt}, \mathrm{ff}\}$ is the *interpretation* of P.

Note that the interpretation $[\![P]\!]_{\mathbb{S}} : U^n \to \{\mathrm{tt}, \mathrm{ff}$ of a predicate symbol P can be thought of as the characteristic function (Def. 1.2.13) of some subset of U^n, hence can be identified with a subset $[\![P]\!]_{\mathbb{S}} \subseteq U^n$.

4.3.2 Definition (Valuation)**.** A *valuation* on a structure $\mathbb{S} = (U, \dots)$ is a function

$$J \;:\; \mathbf{Var} \longrightarrow U \;\; .$$

Recall that **Var** is a fixed set of variables (Def. 4.1.2).

Compared to the semantics of equational logic: the parameters (structure and valuation) consist of the same set of data as in equational logic (algebra and valuation), augmented with the interpretation $[\![P]\!]_{\mathbb{S}}$ of predicate symbols.

4.3.3 Definition (Denotation)**.** Let

$$\mathbb{S} = \big(U,\, \big([\![f]\!]_{\mathbb{S}}\big)_{f \in \mathbf{FnSymb}},\, \big([\![P]\!]_{\mathbb{S}}\big)_{P \in \mathbf{PdSymb}}\big) \;\; ,$$

be a structure over **FnSymb** and **PdSymb**; and $J : \mathbf{Var} \to X$ be a valuation on $\mathbb{S}$. For each term t, we define its *denotation*

$$[\![t]\!]_{\mathbb{S},J} \quad \in U$$

as follows, inductively on the construction of t.

$$\begin{aligned} [\![x]\!]_{\mathbb{S},J} &:= J(x) \qquad \text{if } x \in \mathbf{Var}; \\ [\![\, f(t_1, \dots, t_n) \,]\!]_{\mathbb{S},J} &:= [\![f]\!]_{\mathbb{S}}\big([\![t_1]\!]_{\mathbb{S},J}\,, \dots, [\![t_n]\!]_{\mathbb{S},J} \big) \;\; . \end{aligned}$$

Furthermore, for each formula A, we define its *denotation*

$$[\![A]\!]_{\mathbb{S},J} \quad \in \{\mathrm{tt}, \mathrm{ff}\}$$

[4]This condition of nonemptiness is convenient (especially in classical logic): this is related to (the current style of) rule ($\exists$-R), by which e.g., we can derive $(\forall x.A) \supset B \Rightarrow \exists x.(A \supset B)$ ($x \notin \mathrm{FV}(B)$), which is used in Lemma 5.5.6. However, there are also other styles of predicate logic where we do not impose this nonemptiness for structures.

If we allow the empty domain and keep the current syntax, some confusions arise; e.g., $\forall x.\bot \Rightarrow \bot$ is derivable, and $\forall x.\bot$ should hold for the empty structure, but $\bot$ should not hold even for the empty structure! One way for avoiding this kind of incorrect reasoning is to use the notion of an *environment*, which explicitly declares what variables are used in a sequent: with environments, it is $y \mid \forall x.\bot \Rightarrow \bot$ that can be derivable, where y is an environment and a required term t for $\forall$-L is prepared under this environment; then, $\bot$ above is in fact $y \mid \bot$, which means "for all y, $\bot$ holds", a valid statement in the empty structure. See, e.g., [4][end of Section 4.1] for this style of predicate logic and another example. In this style with explicit environments, $\Gamma \Rightarrow \Delta$ in the current syntax corresponds to $\mathbf{Var} \mid \Gamma \Rightarrow \Delta$ rather than $\mathrm{FV}(\Gamma) \cup \mathrm{FV}(\Delta) \mid \Gamma \Rightarrow \Delta$ or $\emptyset \mid \Gamma \Rightarrow \Delta$.

as follows, inductively on the construction of A.

$$\begin{aligned}
[\![P(t_1,\dots,t_n)]\!]_{\mathbb{S},J} = \mathrm{tt} &\iff^{\mathrm{def}} [\![P]\!]_{\mathbb{S}}\big([\![t_1]\!]_{\mathbb{S},J},\ \dots,\ [\![t_n]\!]_{\mathbb{S},J}\big) = \mathrm{tt} && \text{where } P \in \mathbf{PdSymb}_n\\
[\![A \wedge B]\!]_{\mathbb{S},J} = \mathrm{tt} &\iff^{\mathrm{def}} [\![A]\!]_{\mathbb{S},J} = \mathrm{tt} \text{ and } [\![B]\!]_{\mathbb{S},J} = \mathrm{tt}\\
[\![A \vee B]\!]_{\mathbb{S},J} = \mathrm{tt} &\iff^{\mathrm{def}} [\![A]\!]_{\mathbb{S},J} = \mathrm{tt} \text{ or } [\![B]\!]_{\mathbb{S},J} = \mathrm{tt}\\
[\![A \supset B]\!]_{\mathbb{S},J} = \mathrm{tt} &\iff^{\mathrm{def}} [\![A]\!]_{\mathbb{S},J} = \mathrm{ff} \text{ or } [\![B]\!]_{\mathbb{S},J} = \mathrm{tt}\\
[\![\neg A]\!]_{\mathbb{S},J} = \mathrm{tt} &\iff^{\mathrm{def}} [\![A]\!]_{\mathbb{S},J} = \mathrm{ff}\\
[\![\forall x.\, A]\!]_{\mathbb{S},J} = \mathrm{tt} &\iff^{\mathrm{def}} [\![A]\!]_{\mathbb{S},J[x\mapsto u]} = \mathrm{tt} && \text{for any } u \in U\\
[\![\exists x.\, A]\!]_{\mathbb{S},J} = \mathrm{tt} &\iff^{\mathrm{def}} [\![A]\!]_{\mathbb{S},J[x\mapsto u]} = \mathrm{tt} && \text{for some } u \in U
\end{aligned}$$

On the first line, $[\![t_i]\!]_{\mathbb{S},J} \in U$ is the denotation of a term t_i defined earlier; and recall that $[\![P]\!]_{\mathbb{S}}$ is a function $U^n \to \{\mathrm{tt}, \mathrm{ff}\}$. On the last two lines, recall that $J[x \mapsto u]$ is an updated valuation (Def. 2.5.6).

The denotation of a sequent $\Gamma \Rightarrow \Delta$ is defined in the same way as in propositional logic.

4.3.4 Definition (Validity; satisfiability)**.** A formula A (over **FnSymb** and **PdSymb**) is *valid* under a structure $\mathbb{S}$ if, for any valuation J over $\mathbb{S}$, we have $[\![A]\!]_{\mathbb{S},J} = \mathrm{tt}$. We write $\mathbb{S} \models A$ for this.

A formula A is said to be *valid* if it is valid under any structure $\mathbb{S}$ over **FnSymb** and **PdSymb**.

A formula A is said to be *satisfiable* if there exist a structure $\mathbb{S}$ and a valuation J such that $[\![A]\!]_{\mathbb{S},J} = \mathrm{tt}$.

Logical equivalence $\cong$ is defined in the same way: $A \cong B$ if and only if

$$[\![A]\!]_{\mathbb{S},J} = [\![B]\!]_{\mathbb{S},J} \qquad \text{for each } \mathbb{S} \text{ and } J.$$

Continuing Proposition 3.3.8, we have the following result. It can be thought of as an infinitary version of the *de Morgan law*.

4.3.5 Proposition. *We have the following logical equivalences.*

$$\neg\forall x.\, A \cong \exists x.\, \neg A \qquad \neg\exists x.\, A \cong \forall x.\, \neg A$$

Proof. Exercise 4.4.

Here are a couple of technical lemmas that are used later. The first one is much like Lem. 3.3.4.

4.3.6 Lemma. *Let $\mathbb{S}$ be a structure; J, J' be two valuations over $\mathbb{S}$; and A be a formula. Assume that*

$$J(x) \;=\; J'(x) \qquad \textit{for each } x \in \mathrm{FV}(A),$$

where $\mathrm{FV}(A)$ *is from Def. 4.1.8. Then we have*

$$[\![A]\!]_{\mathbb{S},J} \;=\; [\![A]\!]_{\mathbb{S},J'}\ .$$

Proof. By induction on the construction of a formula A. (This involves careful handling of free/bound variables; do Exercise 4.7!) □

The second one is like Prop. 2.5.7.

4.3.7 Lemma. *Let A be a formula, s, t be a term, x be a variable, $\mathbb{S}$ be a structure and J be a valuation over $\mathbb{S}$, all over the same* **FnSymb** *and* **PdSymb**. *We have*

$$[\![\, s[t/x] \,]\!]_{\mathbb{S},J} \;=\; [\![\, s \,]\!]_{\mathbb{S},J[x\mapsto [\![t]\!]_{\mathbb{S},J}]} \qquad \textit{and} \quad [\![\, A[t/x] \,]\!]_{\mathbb{S},J} \;=\; [\![\, A \,]\!]_{\mathbb{S},J[x\mapsto [\![t]\!]_{\mathbb{S},J}]} \;.$$

Proof. By induction on the construction of a term s, and a formula A. Let us here do one of the trickiest cases, where A is a universally quantified formula.

By renaming a bound variable (i.e. taking an α-equivalent formula), we can assume that A is of the form

$$A \;\equiv\; \forall y.\, B \;,$$

where $y \in \mathbf{Var}$ is a variable such that

$$y \not\equiv x \qquad \text{and} \qquad y \notin \mathrm{FV}(t) \;. \tag{4.3}$$

Now,

$$\begin{aligned}
&[\![\, (\forall y.\, B)[t/x] \,]\!]_{\mathbb{S},J} = \mathtt{tt} \\
\iff\quad & [\![\, \forall y.\, (B[t/x]) \,]\!]_{\mathbb{S},J} = \mathtt{tt} \\
&\qquad \text{by def. of capture-avoiding substitution and (4.3)} \\
\iff\quad & [\![\, B[t/x] \,]\!]_{\mathbb{S},J[y\mapsto u]} = \mathtt{tt} \quad \text{for any } u \in U \\
\iff\quad & [\![\, B \,]\!]_{\mathbb{S},\left(J[y\mapsto u]\right)[x\mapsto [\![t]\!]_{\mathbb{S},J[y\mapsto u]}]} = \mathtt{tt} \quad \text{for any } u \in U \\
&\qquad \text{by induction hypothesis (B is ``smaller'' than A)} \\
\iff\quad & [\![\, B \,]\!]_{\mathbb{S},\left(J[y\mapsto u]\right)[x\mapsto [\![t]\!]_{\mathbb{S},J}]} = \mathtt{tt} \quad \text{for any } u \in U \\
&\qquad \text{by } y \notin \mathrm{FV}(t) \text{ from (4.3), and Lem. 4.3.6} \\
\iff\quad & [\![\, B \,]\!]_{\mathbb{S},\left(J[x\mapsto [\![t]\!]_{\mathbb{S},J}]\right)[y\mapsto u]} = \mathtt{tt} \quad \text{for any } u \in U \\
&\qquad \text{by } x \not\equiv y \text{ from (4.3)} \\
\iff\quad & [\![\, \forall y.\, B \,]\!]_{\mathbb{S},J[x\mapsto [\![t]\!]_{\mathbb{S},J}]} = \mathtt{tt} \;.
\end{aligned}$$

This proves $[\![\, (\forall y.\, B)[t/x] \,]\!]_{\mathbb{S},J} = [\![\, \forall y.\, B \,]\!]_{\mathbb{S},J[x\mapsto [\![t]\!]_{\mathbb{S},J}]}$. The other cases are left as an exercise (Exercise 4.8). □

4.4 Predicate Logic: Syntax vs. Semantics

4.4.1 Theorem (Soundness)**.** $\vdash \Gamma \Rightarrow \Delta$ *implies* $\models \Gamma \Rightarrow \Delta$.

Proof. The proof is again by induction, and is mostly the same as for propositional logic. Here we do just one case (which needs careful handling of free/bound variables and hence is tricky), for the rule (∀-R).

$$\frac{\Gamma \Rightarrow \Delta, A[z/x]}{\Gamma \Rightarrow \Delta, \forall x.A}\ (\forall\text{-R}),\ (\text{VC})$$

Let $\mathbb{S}$ be an arbitrary structure; and J be an arbitrary valuation over $\mathbb{S}$. We are to prove $[\![\Gamma \Rightarrow \Delta, \forall x.A]\!]_{\mathbb{S},J} = \mathtt{tt}$. Assume $[\![\bigwedge \Gamma]\!]_{\mathbb{S},J} = \mathtt{tt}$ and $[\![\bigvee \Delta]\!]_{\mathbb{S},J} = \mathtt{ff}$—otherwise the goal is trivial. We need to show that $[\![\forall x.\, A]\!]_{\mathbb{S},J} = \mathtt{tt}$, that is,

$$[\![A]\!]_{\mathbb{S},J[x\mapsto u]} = \mathtt{tt} \qquad \text{for any } u \in U. \tag{4.4}$$

4.4.2 Sublemma. $[\![A]\!]_{\mathbb{S},J[x\mapsto u]} = [\![A]\!]_{\mathbb{S},\left(\,J[z\mapsto u]\,\right)\left[\,x\mapsto(\,J[z\mapsto u]\,)(z)\,\right]}$.

Proof. (Of Sublem. 4.4.2) If $z \equiv x$, then the valuation $\left(\,J[z\mapsto u]\,\right)\left[\,x\mapsto(\,J[z\mapsto u]\,)(z)\,\right]$ on the right is obviously equal to the valuation $J[x\mapsto u]$. If $z \not\equiv x$, we have

$$\begin{aligned}[\![A]\!]_{\mathbb{S},\left(\,J[z\mapsto u]\,\right)\left[\,x\mapsto(\,J[z\mapsto u]\,)(z)\,\right]} &= [\![A]\!]_{\mathbb{S},\left(\,J[z\mapsto u]\,\right)[x\mapsto u]} \\ &= [\![A]\!]_{\mathbb{S},\left(\,J[x\mapsto u]\,\right)[z\mapsto u]} \\ &\overset{(*)}{=} [\![A]\!]_{\mathbb{S},J[x\mapsto u]} \ ,\end{aligned}$$

where $(*)$ holds due to Lem. 4.3.6—if $z \in \mathrm{FV}(A)$, then $z \in \mathrm{FV}(\forall x.\, A)$ since $z \not\equiv x$; and this violates (VC). This concludes the proof of the sublemma. □

We turn back to the proof of Thm. 4.4.1. We have

$$\begin{aligned}[\![A]\!]_{\mathbb{S},J[x\mapsto u]} &= [\![A]\!]_{\mathbb{S},\left(\,J[z\mapsto u]\,\right)\left[\,x\mapsto(\,J[z\mapsto u]\,)(z)\,\right]} && \text{by Sublem. 4.4.2} \\ &= [\![A]\!]_{\mathbb{S},\left(\,J[z\mapsto u]\,\right)\left[\,x\mapsto[\![z]\!]_{\mathbb{S},\,J[z\mapsto u]}\,\right]} && \text{by def. of } [\![z]\!] \\ &= [\![A[z/x]]\!]_{\mathbb{S},J[z\mapsto u]} && \text{by Lem. 4.3.7.}\end{aligned}$$

Now:

$$\begin{aligned}[\![\,\Gamma \Rightarrow \Delta, A[z/x]\,]\!]_{\mathbb{S},J[z\mapsto u]} &= \mathrm{tt} && \text{by induction hypothesis;} \\ [\![\textstyle\bigwedge\Gamma]\!]_{\mathbb{S},J[z\mapsto u]} &= [\![\textstyle\bigwedge\Gamma]\!]_{\mathbb{S},J} && \text{since } z \text{ is not free in } \Gamma \text{ by (VC)} \\ &= \mathrm{tt} && \text{by assumption;} \\ [\![\textstyle\bigwedge\Delta]\!]_{\mathbb{S},J[z\mapsto u]} &= \mathrm{ff} && \text{similarly.}\end{aligned}$$

Therefore we have $[\![A[z/x]]\!]_{\mathbb{S},J[z\mapsto u]} = \mathrm{tt}$, for each $u \in U$. Combined with $[\![A]\!]_{\mathbb{S},J[x\mapsto u]} = [\![A[z/x]]\!]_{\mathbb{S},J[z\mapsto u]}$ that we showed above, we obtain (4.4). This concludes the ($\forall$-R) case of the soundness proof. □

4.4.3 Theorem (Completeness)**.** $\models \Gamma \Rightarrow \Delta$ *implies* $\vdash \Gamma \Rightarrow \Delta$*. Moreover: for any formula* A*,* $\models A$ *implies* $\vdash A$*.*

Proof. The proof strategy is the same as before: given an underivable sequent, we construct a counter model using syntactic ingredients. The construction is involved and we do not present it here. Interested readers are referred to e.g. [6, 8]. □

Completeness for predicate logic was first shown by K. Gödel; this is *Gödel's completeness theorem*.

But what about the famous *Gödel's incompleteness theorem*? The difference is in their statements: in Thm. 4.4.3, it is shown that

> a deductive system (like predicate LK) proves all the sequents that are true *under any model (i.e. structure).*

Recall the definition of $\models$. In contrast, what the incompleteness theorem states is (roughly):

there is no sound and complete set of deductive rules for the formulas that are true in *one specific structure* $\mathbb{N}$, namely that of *natural numbers.*

$$\vdash A \quad \underset{\text{completeness}}{\overset{\text{soundness}}{\rightleftharpoons}} \quad \models A \quad \overset{\text{obvious}}{\not\Longleftarrow} \quad \mathbb{N} \models A$$

This means that there are *nonstandard structures*, besides $\mathbb{N}$, that falsifies what is true in $\mathbb{N}$. We will briefly come back to this point later in this course. An interested reader can look at [6, 19], among other textbooks.

Exercises

4.1. Assume that $f \in \mathbf{FnSymb}_1$ and $g \in \mathbf{FnSymb}_2$. Which of the following are (syntactically legitimate) terms?

$$f(c, g(d)) \qquad x(g(c, f(y))) \qquad g(f(c), g(x, f(y)))$$

4.2. Give explicit formulas for the following (capture-avoiding) substitutions.

$\big(\, R(x,y) \,\big)[f(x)/y] \quad \big(\, Q(x) \wedge \exists x.\, R(x,y) \,\big)[g(y)/x] \quad \big(\, Q(y) \wedge \exists x.\, R(x,y) \,\big)[f(x)/y]$
$\big(\forall x. R(x,y)\big)[f(x)/x]$

4.3. Point out what is wrong with the "proof" in (4.2).

4.4. Prove Proposition 4.3.5.

4.5. Let P, Q, R be predicate symbols of arity $0, 1, 1$, respectively. Are these formulas valid/satisfiable/unsatisfiable?

- $\forall x.(P \vee Q(x)) \supset P \vee (\forall x.Q(x))$
- $\forall x.(R(x) \vee Q(x)) \supset (\forall x.R(x)) \vee (\forall x.Q(x))$
- $\forall x.(P \wedge Q(x)) \supset P \wedge (\forall x.Q(x))$
- $\forall x.(R(x) \wedge Q(x)) \supset (\forall x.R(x)) \wedge (\forall x.Q(x))$
- $\big(\, (\forall x.\, Q(x)) \supset P \,\big) \supset \exists x.\, \big(\, Q(x) \supset P \,\big)$

Present proof trees in LK for those which are valid.

4.6. In Def. 4.3.1, a universe U is assumed to be nonempty. Show that, if we change the definition and allow U to be empty, soundness (Thm. 4.4.1) no longer holds. (Hint: consider the formula $\forall x.\, P(x) \supset \exists x.\, P(x)$)

4.7. Prove Lem. 4.3.6.

4.8. Complete the proof of Lem. 4.3.7.

4.9. Complete the proof of Thm. 4.4.1.

Chapter 5

Some More *Meta*-Theorems

In this chapter we discuss some further topics on propositional/predicate logic.

5.1 Cut Elimination

As you have done in exercises, derivation rules are usually used *from bottom to top*: given a formula to be derived, you try to construct a proof tree in the bottom-up manner. In that aspect, among the rules in Fig. 3.1, the (Cut) rule distinguishes itself.

$$\frac{\Gamma \Rightarrow \Delta, A \quad A, \Pi \Rightarrow \Sigma}{\Gamma, \Pi \Rightarrow \Delta, \Sigma} \text{ (Cut)}$$

In the (Cut) rule, even if you are given the lower sequent $\Gamma, \Pi \Rightarrow \Delta, \Sigma$, you have *absolutely no idea* what the formula A in the upper sequent—called the *cut formula*—should be. In the other rules, in contrast, what appears in the upper sequent is always a subformula of a formula in the lower sequent. This property is called the *subformula property* of a rule (or a derivation system). In summary: the (Cut) rule makes the subformula property of LK—a property much desired in proof search—fail.

5.1.1 Remark. Note that in the predicate case the definition of *subformula* is a bit tricky. Consider e.g. the following rule.

$$\frac{A[t/x], \Gamma \Rightarrow \Delta}{\forall x.\, A, \Gamma \Rightarrow \Delta} \text{ ($\forall$-L)}$$

We define that $A[t/x]$, with any term t, is a subformula of $\forall x.\, A$. This is another burden in proof search; it is mainly due to this fact that validity of a predicate formula is undecidable.

We are saved by the following result, originally due to Gentzen.

5.1.2 Theorem (Cut elimination)**.** *LK, whether propositional or predicate, admits cut-elimination. That is, each derivable sequent has a cut-free derivation.*

Proof. There are syntactic conversion procedures that turn an LK proof (with cuts) into a cut-free LK proof. See e.g. [14, 8]. □

On a superficial level, the theorem states "any proof can be converted into a proof in some normal form." Notice that it is a *meta* result, on proofs as *object level* entities. Cut elimination is central in *proof theory*, that is (meta)mathematics of proofs.

5.1.3 Remark. The cut elimination result also exists for natural deduction, where it is often called *proof normalization*. Via the Curry-Howard correspondence, cut elimination corresponds to β*-reduction* of λ-terms. See [3] if you are interested.

5.2 Theory and Compactness

Compactness reveals the limitation of propositional/predicate logic: their expressive power is limited due to their *finitary* nature.

In this section we present definitions and theorems at the same time for both propositional and predicate logics.

5.2.1 Definition (Theory)**.** A *(propositional) theory* Φ is a set of propositional formulas.

A *(predicate) theory* Φ is a set of closed predicate formulas.

A formula $A \in \Phi$ is often called a *non-logical axiom*.[1]

This notion is similar to the set E of axioms in Chap. 2. Note that with this definition, we have dragged "the notion of theory" down to the object level.

5.2.2 Definition (Derivability within a theory)**.** A sequent $\Gamma \Rightarrow \Delta$ is said to be *derivable within a theory* Φ if there is an proof tree of $\Gamma \Rightarrow \Delta$ in LK, where we are allowed to use the following additional rule scheme.

$$\frac{}{\Rightarrow B}\ (\textsc{Axiom}), B \in \Phi$$

We write $\Phi \vdash \Gamma \Rightarrow \Delta$ if the sequent $\Gamma \Rightarrow \Delta$ is derivable within Φ.

5.2.3 Lemma (Deduction Theorem)**.** $\Phi \vdash \Gamma \Rightarrow \Delta$ *if and only if there exist finitely many formulas* $A_1, \ldots, A_n \in \Phi$ *such that* $\vdash A_1, \ldots, A_n, \Gamma \Rightarrow \Delta$.

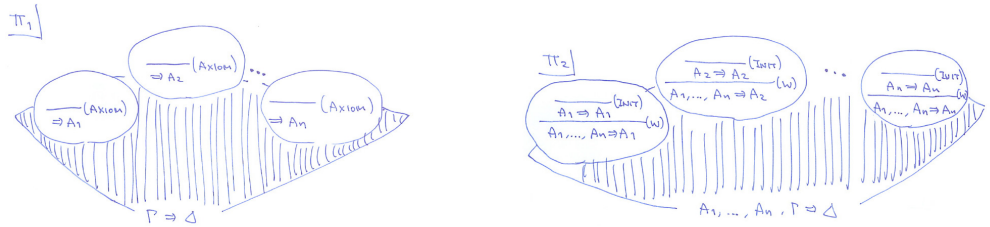
Proof. The 'if' part: let Π be a proof of $A_1, \ldots, A_n, \Gamma \Rightarrow \Delta$ in LK (without any non-logical axioms). Then

$$\dfrac{\dfrac{}{\Rightarrow A_n}\ (\textsc{Axiom}) \qquad \dfrac{\dfrac{\dfrac{}{\Rightarrow A_1}\ (\textsc{Axiom}) \qquad \begin{array}{c}\vdots\ \Pi\\ A_1, A_2, \ldots, A_n, \Gamma \Rightarrow \Delta\end{array}}{A_2, \ldots, A_n, \Gamma \Rightarrow \Delta}\ (\textsc{Cut}) \qquad \begin{array}{c}\vdots\\ A_n, \Gamma \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta}\ (\textsc{Cut})$$

(that eliminates $A_1, \ldots, A_n$ by (Axiom) and (Cut)) is a proof tree within the theory Φ. Thus $\Phi \vdash \Gamma \Rightarrow \Delta$.

The 'only if' part: let Π_1 be a proof tree of $\Gamma \Rightarrow \Delta$ within Φ.

[1] In contrast, an initial sequent $A \Rightarrow A$ may well be called a *logical axiom*.



To this tree we apply the following operations.

- To the left hand side of each of its nodes, we add formulas $A_1, \ldots, A_n$; and
- each (Axiom) node is transformed into a suitable combination of the (Init) and (Weakening-L) rules (see above right).

Then the resulting tree is an LK proof tree without any non-logical axioms—note that the added formulas $A_1, \ldots, A_n$ cause no problem in rule applications. Therefore $\vdash A_1, \ldots, A_n, \Gamma \Rightarrow \Delta$. □

A theory can be any set of formulas—it can even contain both P and $\neg P$!

5.2.4 Definition (Consistency)**.** A theory Φ is *consistent* if $\Phi \nvdash \Rightarrow$. Otherwise it is *inconsistent*.

Note here that the sequent $\Rightarrow$ (with both hands empty) means "false" (cf. Lem. 3.2.6). From this, using (Weakening-R), one can derive any formula A. Therefore: an inconsistent theory derives *anything*.[2]

5.2.5 Lemma. *1. Φ is consistent if and only if any finite subset Φ' of Φ is consistent.*

2. A theory Φ is consistent if and only if $(\Phi, \emptyset)$ is a consistent pair (Def. 3.4.4).

Proof. 1. Immediate from Lem. 5.2.3.

2.

$$
\begin{array}{lll}
(\Phi, \emptyset) \text{ is a consistent pair} & & \\
\iff & \forall \Phi' \subseteq_{\text{fin.}} \Phi.\ \nvdash \Phi' \Rightarrow & \text{by def. of consistent pair} \\
\iff & \forall \Phi' \subseteq_{\text{fin.}} \Phi.\ \Phi' \nvdash \Rightarrow & \text{by Lem. 5.2.3} \\
\iff & \forall \Phi' \subseteq_{\text{fin.}} \Phi.\ \Phi' \text{ is consistent} & \\
\iff & \Phi \text{ is consistent} & \text{by Lem. 5.2.3.}
\end{array}
$$
□

Consistency is a syntactic notion; the corresponding semantic notion is that of satisfiability.

5.2.6 Definition (Satisfiable theory)**.** A propositional theory Φ is said to be *satisfiable* if there exists a valuation J such that

$$[\![A]\!]_J = \mathrm{tt} \qquad \text{for } \textit{every } A \in \Phi.$$

[2] `http://togetter.com/li/37411`

A predicate theory Φ is said to be *satisfiable* if there exist a structure $\mathbb{S}$ and a valuation J over $\mathbb{S}$ such that

$$[\![A]\!]_{\mathbb{S},J} = \mathrm{tt} \qquad \text{for } \textit{every } A \in \Phi.$$

We now show that consistency and satisfiability are equivalent. First, the easier direction:

5.2.7 Lemma. *If Φ is satisfiable, then it is consistent.*

Proof. Assume not, that is, $\Phi \vdash \Rightarrow$. By Lem. 5.2.3, there exist $A_1, \ldots, A_n \in \Phi$ such that $\vdash A_1, \ldots, A_n \Rightarrow$.

By soundness of LK, $A_1 \wedge \cdots \wedge A_n \supset \bot$ is valid (cf. Lem. 3.2.6 and Notation 3.1.5); this is logically equivalent to

$$\neg(A_1 \wedge \cdots \wedge A_n) \ , \quad \text{and thus to} \quad \neg A_1 \vee \cdots \vee \neg A_n \ .$$

This contradicts with satisfiability of Φ. □

The other direction is:

5.2.8 Theorem (Strong completeness)**.** *If a theory Φ is consistent, then it is satisfiable.*

Proof. For the propositional case, the proof is much like that of completeness (Thm. 3.4.3). We present its outline.

We proved in Lem. 5.2.5.2 that $(\Phi, \emptyset)$ is a consistent pair. We use Lem. 3.4.6 to extend it to a maximally consistent pair (U', V'); and then derive a valuation J as in Lem. 3.4.10. The resulting valuation makes all the formulas in U' true; thus in particular all the formulas in Φ.

For the predicate case, too, a completeness proof (which we skipped) can be easily turned into a proof of strong completeness. □

Thm. 5.2.8 is called *strong completeness* since it yields completeness (Exercise 5.1).

Finally we come to compactness. It states satisfiability—i.e. consistency, by Lem. 5.2.7 and Thm. 5.2.8)—is "finitely determined."

5.2.9 Theorem (Compactness)**.** *Let Φ be a theory. Then the following are equivalent:*

1. *any finite subset Φ' of Φ is satisfiable;*
2. *Φ is satisfiable.*

Proof. Immediate from Lem. 5.2.7, Thm. 5.2.8 and Lem. 5.2.5.1. □

Note here that Thm. 5.2.9 does not mention the derivation system LK.

5.3 Axiomatizable Class of Structures—Consequence of Compactness

Here we follow [15, §2.7] and use compactness to exhibit a limitation of predicate logic. Specifically, we show that no theory characterizes well-ordered sets.

5.3.1 Well-Ordered Set

First, let us recall:

5.3.1 Definition (Well-ordered set)**.** A *well-ordered set* is a poset $(S, \leq)$ such that:

any nonempty subset $S' \subseteq S$ has its minimum.

An example is $\mathbb{N}$. Non-examples are

– $\mathbb{Z}$ (consider $\{-1, -2, -3, \dots\}$); and

– $\mathbb{R}$ (consider $\{x \in \mathbb{R} \mid 0 < x\}$).

A poset $(X, \leq)$ is called *totally ordered* if for any $x, y \in X$, at least one of $x \leq y$ or $y \leq x$ holds. Then we have:

5.3.2 Lemma. *For a poset $(X, \leq)$, the following are equivalent.*

1. *$(X, \leq)$ is a well-ordered set.*
2. *$(X, \leq)$ is a totally ordered set, and there is no infinite descending chain $x_0 > x_1 > x_2 > \dots$.* □

We will be using the characterization of Lem. 5.3.2.2.

5.3.2 Model

5.3.3 Definition. In the current section we overload previous definitions (including Def. 4.3.1) and assume that:

– **PdSymb** contains a special binary symbol $=$, and

– its interpretation $[\![=]\!]_{\mathbb{S}}$ in a structure $\mathbb{S}$ is fixed to be the actual equality. That is,

$$[\![=]\!]_{\mathbb{S}} \;=\; \big\{\, (u, u) \;\mid\; u \in U \,\big\} \;.$$

Now let us fix

$$\mathbf{FnSymb} = \emptyset \quad \text{and} \quad \mathbf{PdSymb} = \{R, =\} \;, \tag{5.1}$$

where we let R and $=$ be binary predicate symbols.

A structure

$$\mathbb{S} = \big(\, U,\, \big([\![f]\!]_{\mathbb{S}}\big)_{f \in \mathbf{FnSymb}},\, \big([\![P]\!]_{\mathbb{S}}\big)_{P \in \mathbf{PdSymb}} \,\big) \;,$$

for these **FnSymb** and **PdSymb** has an *underlying binary relation*

$$[\![R]\!]_{\mathbb{S}} \;\subseteq\; U \times U \;.$$

The following predicate formulas state that the underlying binary relation is indeed a total order.

$$\begin{aligned}
A_1 &:\equiv \quad \forall x.\, R(x, x) \\
A_2 &:\equiv \quad \forall x.\, \forall y.\, \big(\, R(x, y) \wedge R(y, x) \supset x = y \,\big) \\
A_3 &:\equiv \quad \forall x.\, \forall y.\, \forall z.\, \big(\, R(x, y) \wedge R(y, z) \supset R(x, z) \,\big) \\
A_4 &:\equiv \quad \forall x.\, \forall y.\, \big(\, R(x, y) \vee R(y, x) \,\big)
\end{aligned}$$

We define a theory Φ by

$$\Phi := \{A_1, A_2, A_3, A_4\}$$

5.3.4 Definition (Model of a theory)**.** A *model* of a theory Φ is a structure $\mathbb{S}$ such that

$$\mathbb{S} \models A \qquad \text{for each } A \in \Phi.$$

We denote the set of models of Φ by $\mathrm{Mod}(\Phi)$.

Thus a model is a structure where all the (non-logical) axioms are valid. Recalling Chap. 2: model is to structure what (Σ, E)-algebra is to Σ-algebra.

5.3.5 Lemma. *Assume that* $\mathbb{S}$ *is the structure. Then:*

$\mathbb{S}$ *is a model of* $\Phi \iff$ *the underlying binary relation is a total order.* □

Thus we have seen that

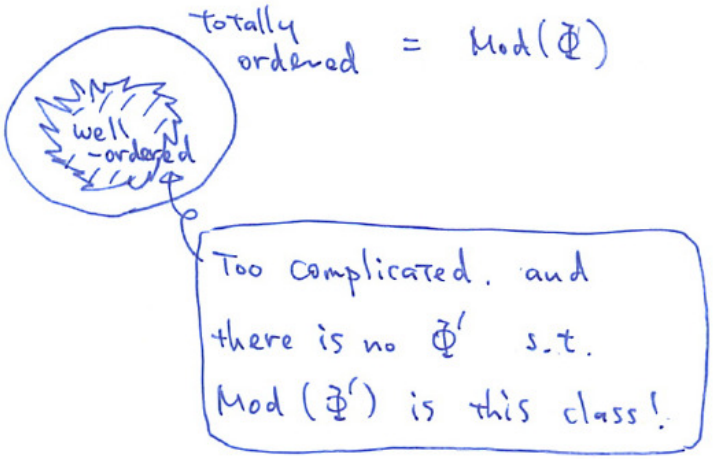
> the class of totally ordered sets can be characterized by a theory;

in other words,

> predicate logic is expressive enough to characterize totally ordered sets.

5.3.3 Non-Axiomatizable Class of Structures

We now show that there is no theory Φ' such that $\mathrm{Mod}(\Phi')$ is exactly the class of well-ordered sets.[3]



5.3.6 Definition (Axiomatizability)**.** Let any **FnSymb** and **PdSymb** be fixed. A class $\mathcal{A}$ of structures is said to be *axiomatizable* if there is a theory Φ' such that

$$\mathcal{A} = \mathrm{Mod}(\Phi') \ .$$

The next trivial lemma is used in the theorem after this.

[3] Here "class" means "set"; we used the word "class" simply to avoid "set of sets" that sounds cumbersome.

5.3.7 Lemma. *Let* $\mathbf{FnSymb} \subseteq \mathbf{FnSymb}'$ *and* $\mathbf{PdSymb}$ *be signatures,* $\mathbb{S}'$ *be a structure over* $\mathbf{FnSymb}'$ *and* $\mathbf{PdSymb}$*, and* $\mathbb{S}$ *be the structure over* $\mathbf{FnSymb}$ *and* $\mathbf{PdSymb}$ *obtained from* $\mathbb{S}$ *by forgetting the interpretations of symbols in* $\mathbf{FnSymb}' \setminus \mathbf{FnSymb}$*. Then, for a formula* A *and a theory* Φ *over* $\mathbf{FnSymb}$ *and* $\mathbf{PdSymb}$*, we have*

$$[\![A]\!]_{\mathbb{S}} = [\![A]\!]_{\mathbb{S}'} \qquad \text{and} \qquad \mathbb{S} \models \Phi \Longleftrightarrow \mathbb{S}' \models \Phi \ .$$

5.3.8 Theorem. *Let* $\mathbf{FnSymb}$ *and* $\mathbf{PdSymb}$ *be as in* (5.1)*. The class of well-ordered sets is not axiomatizable.*[4]

The proof relies on compactness (Thm. 5.2.9).

Proof. We argue by contradiction. Assume that a theory Φ' is such that, for any structure $\mathbb{S}$,

$$\mathbb{S} \models \Phi' \quad \Longleftrightarrow \quad \mathbb{S} \text{ is a well-ordered set.}$$

Now let us consider $\mathbf{FnSymb}' := \{c_1, c_2, \dots\}$ where all c_i are nullary. For each $n \in \mathbb{N}$, consider the following formula:

$$B_n \ :\equiv \ R(c_{n+1}, c_n) \wedge \neg R(c_n, c_{n+1}) \ ,$$

which intuitively means "$c_{n+1} < c_n$." We consider the theory

$$\Phi'' \ := \ \Phi' \cup \{B_n \mid n \in \mathbb{N}\}$$

over the signatures $\mathbf{FnSymb}'$ and $\mathbf{PdSymb}$. By Lemma 5.3.7, for any structure $\mathbb{S}' = (U, ([\![c_i]\!]_{\mathbb{S}'})_i, [\![R]\!]_{\mathbb{S}'})$,

$$\mathbb{S}' \models \Phi' \quad \Longleftrightarrow \quad (U, , [\![R]\!]_{\mathbb{S}'}) \models \Phi' \quad \Longleftrightarrow \quad (U, , [\![R]\!]_{\mathbb{S}'}) \text{ is a well-ordered set.}$$

The theory Φ'' must not be satisfiable: if Φ'' is satisfied by $\mathbb{S}'$, then Φ' means that there is no infinite descending chain (Lem. 5.3.2); but by satisfying all the B_n's there is a descending chain

$$[\![c_0]\!]_{\mathbb{S}'} > [\![c_1]\!]_{\mathbb{S}'} > [\![c_2]\!]_{\mathbb{S}'} > \cdots \ ,$$

which is a contradiction.

Let Φ''' be an arbitrary finite subset of Φ'', and we claim that Φ''' is satisfiable. Since Φ''' is finite, there are only finitely many B_n's in it; so it suffices to show that

$$\Phi' \cup \{B_0, \dots, B_{N-1}\}$$

is satisfiable for any $N \geq 0$. To see this, consider a structure $\mathbb{S}'_N$ with the universe

$$\{0, 1, \dots, N\}$$

and

$$[\![c_i]\!]_{\mathbb{S}'_N} = \begin{cases} N - i & \text{if } i \in [0, N] \\ 0 & \text{otherwise,} \end{cases}$$

and $[\![R]\!]_{\mathbb{S}'_N}$ is $<$, the usual inequality between natural numbers. Then $\mathbb{S}'_N$ is a model of Φ'—since $\mathbb{S}'_N$ is well-ordered—and satisfies $B_0, \dots, B_{N-1}$.

Thus any finite subset Φ''' of Φ'' is satisfiable; thus by compactness (Thm. 5.2.9), Φ'' is satisfiable. This is a contradiction. □

[4] To be precise, we here consider only *nonempty* well-ordered sets, just because our semantics of predicate logic targets only nonempty structures.

A question about *expressivity* of a (finitary) formalism—like the one that we have just seen—is everywhere in (theoretical) computer science. You must find some of the results that you learned in formal language theory bear the same flavor.

5.4 The Resolution Principle: Propositional Case

LK is a clever formalism that is convenient for many meta-theorems (completeness, cut-elimination, etc.) We now introduce another formalism for derivation in propositional logic—that of *resolution*—that is suited for proof search. It is originally due to J.A. Robinson, and is a foundation of many proof assistants as well as logic programming.

Here is an example of resolution.

$$\dfrac{\dfrac{\dfrac{\{Q,\underline{R}\}\quad\{Q,\underline{\neg R}\}}{\{\underline{Q}\}}\qquad\dfrac{\{\underline{P},\neg Q,R\}\quad\{\underline{\neg P},R\}}{\{\underline{\neg Q},R\}}}{\{\underline{R}\}}\qquad\{\underline{\neg R}\}}{\emptyset} \tag{5.2}$$

5.4.1 Definition (Literal; clause; Horn clause)**.** Among propositional formulas/sequents:

- A *literal* is either a propositional variable $P \in \mathbf{PVar}$, or its negation $\neg P$.
- A *clause* is a finite set
$$c \;=\; \{L_1, \ldots, L_n\}$$
of literals.
- A *Horn clause* is a sequent of the form
$$P_1, \ldots, P_n \Rightarrow P$$
where P_i and P are propositional variables. The number n can be 0.

The intuition is: the clause $\{L_1, \ldots, L_n\}$ means

$$L_1 \vee \cdots \vee L_n \;\;.$$

A Horn clause can be thought of as a special case of clause, since

$$\begin{aligned}(P_1 \wedge \cdots \wedge P_n) \supset P \;&\cong\; \neg(P_1 \wedge \cdots \wedge P_n) \vee P \\ &\cong\; \neg P_1 \vee \cdots \vee \neg P_n \vee P \;\;.\end{aligned}$$

5.4.2 Definition (Complement L^* of a literal)**.** For each literal L, its *complement* L^* is defined by:

$$\begin{array}{ll}\text{if } L \equiv P, \text{ then} & L^* :\equiv \neg P \;\;; \\ \text{if } L \equiv \neg P, \text{ then} & L^* :\equiv P \;\;.\end{array}$$

5.4.3 Definition (Resolvent)**.** Let c, c_1, c_2 be clauses. We say c is a *resolvent* of c_1 and c_2 if there exists a literal L such that

– $L \in c_1$,

– $L^* \in c_2$, and

– $c = \big(c_1 \setminus \{L\}\big) \cup \big(c_2 \setminus \{L^*\}\big)$.

Note that, given c_1 and c_2, their resolvent is not necessarily unique since multiple choices of a literal L might be possible.

5.4.4 Example. Let $c_1 = \{P, \neg Q, \neg R\}$ and $c_2 = \{\neg P, \neg Q, R, S\}$. Then each of

$$\{\neg Q, \neg R, R, S\} \quad \text{and} \quad \{P, \neg P, \neg Q, S\}$$

is their resolvent.

In resolution, we derive clauses from a finite set S of clauses:

$$\begin{aligned} S &= \{c_1, \ldots, c_n\} \\ &= \left\{ \begin{array}{c} \{L_{1,1}, \ldots, L_{1,m_1}\}, \\ \vdots \\ \{L_{n,1}, \ldots, L_{n,m_n}\} \end{array} \right\} , \end{aligned}$$

where each $L_{i,j}$ is a literal. The intuition is that S stands for an assumption

$$\big(\bigvee c_1\big) \wedge \cdots \wedge \big(\bigvee c_n\big) \ , \quad \text{i.e.} \quad \begin{array}{l} (L_{1,1} \vee \cdots \vee L_{1,m_1}) \\ \wedge (L_{2,1} \vee \cdots \vee L_{2,m_2}) \\ \wedge \ldots \\ \wedge (L_{n,1} \vee \cdots \vee L_{n,m_n}) \end{array} .$$

5.4.5 Definition (Resolution tree)**.** Let S be a finite set of clauses. A *resolution tree* under S is a finite tree—each of whose node is labeled with a clause—defined inductively as follows.

– If $c_i \in S = \{c_1, \ldots, c_n\}$, the one-node tree

$$\overline{c_i}$$

is a resolution tree under S.

– If

$$\begin{array}{c} \vdots \ \Sigma_1 \\ c_1 \end{array} \quad \text{and} \quad \begin{array}{c} \vdots \ \Sigma_2 \\ c_2 \end{array}$$

are both resolution trees under S, and c is a resolvent of c_1 and c_2, then

$$\dfrac{\begin{array}{cc} \vdots \ \Sigma_1 & \vdots \ \Sigma_2 \\ c_1 & c_2 \end{array}}{c}$$

is a resolution tree under S.

Roughly speaking: in resolution one starts with a clause in S and proceeds by taking resolvents.

5.4.6 Example. The tree in (5.2) is a resolution tree under the set

$$\{\ \{Q,R\},\{Q,\neg R\},\{P,\neg Q,R\},\{\neg P,R\},\{\neg R\}\ \}\ .$$

5.4.7 Notation ($c(S)$)**.** Let $S=\{c_1,\dots,c_n\}$ be a finite set of clauses. The formula $c(S)$ is defined by

$$(\bigvee c_1)\wedge\cdots\wedge(\bigvee c_n)\ ,\quad \text{i.e.}\quad \begin{array}{l}(L_{1,1}\vee\cdots\vee L_{1,m_1})\\ \wedge(L_{2,1}\vee\cdots\vee L_{2,m_2})\\ \wedge\dots\\ \wedge(L_{n,1}\vee\cdots\vee L_{n,m_n})\end{array}\ ,$$

where $\bigwedge$ and $\bigvee$ are from Notation 3.1.5, and the order of clauses and formulas are chosen arbitrarily. Note that $c(S)$ is in conjunctive normal form (CNF, Exercise 3.10).

5.4.8 Theorem (Soundness of resolution)**.** *Let $S=\{c_1,\dots,c_n\}$ be a finite set of clauses. If there is a resolution tree under S whose root is a clause c, then the sequent*

$$c(S)\Rightarrow\bigvee c$$

is derivable in LK. Therefore by Thm. 3.4.1, $c(S)\Rightarrow\bigvee c$ is valid.

Proof. By induction on the construction of a resolution tree. If it is a one-node tree with $c\in S$, it is easy to see that $\vdash c(S)\Rightarrow c$.

For the step case, assume

$$\vdash c(S)\Rightarrow\bigvee c_1\ ,\quad \vdash c(S)\Rightarrow\bigvee c_2\ ,\quad \text{and}\quad c \text{ is a resolvent of } c_1 \text{ and } c_2.$$

Let L be the literal such that $L\in c_1$, $L^*\in c_2$ and $c=(c_1\setminus\{L\})\cup(c_2\setminus\{L^*\})$. Writing $A_1:\equiv\bigvee\big(c_1\setminus\{L\}\big)$ and $A_2:\equiv\bigvee\big(c_2\setminus\{L^*\}\big)$, by assumption we have

$$\vdash c(S)\Rightarrow A_1\vee L\ ,\quad \vdash c(S)\Rightarrow A_2\vee L^*\ ,\quad\text{and}\quad \bigvee c\ \cong\ A_1\vee A_2\ .$$

The first two, together with $\vdash\Rightarrow L,L^*$, derive

$$\vdash c(S)\Rightarrow A_1\vee A_2$$

by applying the (Cut) rule a few times. Using $\bigvee c\ \cong\ A_1\vee A_2$ we obtain the claim. □

In the actual use of resolution, what is usually done is *refutation-based reasoning*. That is,

- given assumptions, and a formula A to be derived from the assumptions,
- we add $\neg A$ to the assumption and try to derive contradiction.

5.4.9 Notation. In resolution it is customary to denote the empty clause $\{\}=\emptyset$ by $\square$. We adopt this convention in what follows. Semantically, it is the disjunction of zero formulas—i.e. the unit for $\vee$—thus it means $\bot$ (Notation 3.1.5).

The completeness property also holds.

5.4.10 Theorem (Completeness of resolution)**.** *Let $S = \{c_1, \ldots, c_n\}$ be a finite set of clauses. If*

$$c(S) \Rightarrow$$

is derivable in LK—i.e. if the formula $\neg c(S)$ is valid—then there exists a resolution tree under S whose root is the empty clause $\square$.

For the proof we use the following lemmas.

5.4.11 Lemma (Weakening an assumption)**.** *Assume there is a resolution tree under $S = \{c_1, \ldots, c_n\}$ whose root is c; and let d be a finite set of formulas.*

Then there is some $d' \subseteq d$ such that there exists a resolution tree under

$$S' := \{c_1 \cup d, c_2, \ldots, c_n\}$$

whose root is $c \cup d'$.

In S', an assumption $c_1 \in S$ is *weakened* into $c_1 \cup d$. The lemma claims that some weakened conclusion $c \cup d'$ can always be derived.

Proof. (Of Lem. 5.4.11) By induction on the construction of a resolution tree for c. (Exercise 5.5) $\square$

5.4.12 Lemma (Strengthening an assumption)**.** *Assume there is a resolution tree under $S = \{c_1, \ldots, c_n\}$ whose root is c; and that $\widetilde{c_1} \subseteq c_1$.*

Then there is some $\widetilde{c} \subseteq c$ such that there exists a resolution tree under

$$S' := \{\widetilde{c_1}, c_2, \ldots, c_n\}$$

whose root is $\widetilde{c}$.

In this lemma, an assumption c_1 is strengthened; and the claim is that it necessarily derives some stronger clause $\widetilde{c} \subseteq c$.

Proof. (Of Lem. 5.4.12) By induction on the construction of a resolution tree for c. We only show the step case.

Assume c's resolution tree is of the form

$$\dfrac{\begin{array}{cc}\vdots\ \Sigma_1 & \vdots\ \Sigma_2 \\ c_1 & c_2\end{array}}{c}$$

with $L \in c_1$, $L^* \in c_2$ and $c = (c_1 \setminus \{L\}) \cup (c_2 \setminus \{L^*\})$. By induction hypothesis, we have the following resolution trees under S':

$$\begin{array}{cc}\vdots\ \Sigma_1' & \vdots\ \Sigma_2' \\ \widetilde{c_1} & \widetilde{c_2}\end{array} \quad \text{and} \tag{5.3}$$

where $\widetilde{c_1} \subseteq c_1$ and $\widetilde{c_2} \subseteq c_2$. Now we make the following three cases.

– If $L \in \widetilde{c_1}$ and $L^* \in \widetilde{c_2}$, then

$$\widetilde{c} := (\widetilde{c_1} \setminus \{L\}) \cup (\widetilde{c_2} \setminus \{L^*\})$$

is a resolvent of $\widetilde{c_1}$ and $\widetilde{c_2}$. Therefore by (5.3) there is a resolution tree for $\widetilde{c}$. Furthermore, it is easy to see that $\widetilde{c} \subseteq c$; thus we obtained the claim.

– If $L \notin \widetilde{c_1}$, then

$$\begin{aligned} c &= (c_1 \setminus \{L\}) \cup (c_2 \setminus \{L^*\}) \\ &\supseteq (c_1 \setminus \{L\}) \\ &\supseteq \widetilde{c_1} \qquad \text{since } \widetilde{c_1} \subseteq c_1 \text{ and } L \notin \widetilde{c_1}. \end{aligned}$$

Thus we can take $\widetilde{c_1}$ as $\widetilde{c}$ in the claim.

– If $L^* \notin \widetilde{c_2}$, by similar arguments, we see that $\widetilde{c_2}$ can be taken as $\widetilde{c}$ in the claim.

This concludes the proof. □

Proof. (Of Thm. 5.4.10) By induction on the number of $\vee$'s occurring in the formula $c(S)$.

If there is none, then each c_i must be a singleton: $c_i = \{L_i\}$ for each i (where L_i is some literal). The assumption then says that the sequent

$$L_1, \ldots, L_n \Rightarrow$$

is derivable in LK. It is straightforward (Exercise 5.7) that this is only possible when there are two literals in $L_1, \ldots, L_n$ which are complement to each other. That is: $L_j = L_k^*$ with some $j \neq k$. In this case, since $c_j = \{L_j\}$ and $c_k = \{L_k\}$

$$\frac{c_j \quad c_k}{\square}$$

is a resolution tree under S.

Assume there is at least one $\vee$ occurring in $c(S)$. We can assume, without loss of generality, that c_1 contains more than one elements: $c_1 = \{L\} \amalg c_1'$. By assumption we have

$$\vdash L \vee (\bigvee c_1'), \bigvee c_2, \ldots, \bigvee c_n \Rightarrow \quad . \tag{5.4}$$

It is easy to see that

$$\vdash L \Rightarrow L \vee (\bigvee c_1') \quad \text{and} \quad \vdash (\bigvee c_1') \Rightarrow L \vee (\bigvee c_1') \ ;$$

combining with (5.4) and applying the (Cut) rule, we obtain

$$\vdash L, \bigvee c_2, \ldots, \bigvee c_n \Rightarrow \quad \text{and} \quad \vdash (\bigvee c_1'), \bigvee c_2, \ldots, \bigvee c_n \Rightarrow \quad .$$

These two sequents contains less $\vee$'s than $c(S)$; therefore we can use the induction hypothesis and obtain

– a resolution tree Σ_1 under $\left\{ \{L\}, c_2, \ldots, c_n \right\}$ whose root is $\square$; and

– a resolution tree Σ_2 under $\left\{ c_1', c_2, \ldots, c_n \right\}$ whose root is $\square$.

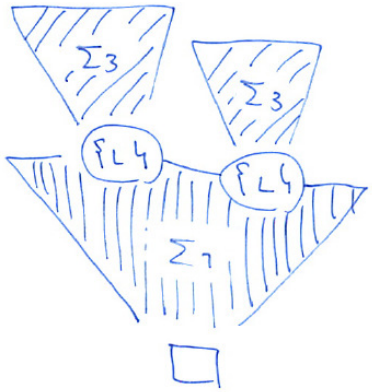
We apply Lem. 5.4.11 to Σ_2; and see that for some $d \subseteq \{L\}$, there is a resolution tree under

$$\left\{ \{L\} \cup c_1', c_2, \ldots, c_n \right\} = \{c_1, \ldots, c_n\} = S$$

whose root is d.

– If $d = \emptyset$: we are done.

– If $d \neq \emptyset$: necessarily $d = \{L\}$. Let us denote this resolution tree (under S, leading to $\{L\}$) by Σ_3. We can combine (copies of) Σ_3 and Σ_1 in the following way.



The result is a resolution tree under S leading to $\square$. $\square$

5.5 The Resolution Principle: Predicate Case

Here we present only a sketch of the predicate version of resolution.

The notions of literal, clause and resolvent are defined in the same way, replacing propositional variables (in the propositional case) with atomic formulas $P(t_1, \ldots, t_n)$ (in the predicate case).

In resolution we have one additional rule of *substitution*.

5.5.1 Definition (Resolution tree)**.** Let S be a finite set of clauses. A *resolution tree* under S is a finite tree—each of whose node is labeled with a clause—defined inductively as follows.

– If $c_i \in S = \{c_1, \ldots, c_n\}$, the one-node tree

$$\overline{c_i}$$

is a resolution tree under S.

– If

$$\begin{array}{c}\vdots\\ c_1\end{array}\ \Sigma_1 \quad \text{and} \quad \begin{array}{c}\vdots\\ c_2\end{array}\ \Sigma_2$$

are both resolution trees under S, and c is a resolvent of c_1 and c_2, then

$$\dfrac{\begin{array}{c}\vdots\\ c_1\end{array}\ \Sigma_1 \qquad \begin{array}{c}\vdots\\ c_2\end{array}\ \Sigma_2}{c}$$

is a resolution trees under S.

– If

$$\begin{array}{c} \vdots\ \Sigma \\ c \end{array}$$

is a resolution tree under S, then

$$\dfrac{\begin{array}{c} \vdots\ \Sigma \\ c \end{array}}{C[t/x]}$$

where x is any variable and t is any term, is a resolution tree under S.

The soundness and completeness properties hold here as well. We skip their proofs; if you are interested see e.g. [15].

5.5.2 Theorem (Soundness and completeness of resolution)**.** *Let $S = \{c_1, \ldots, c_n\}$ be a finite set of clauses, and $x_1, \ldots, x_m$ are variables occurring in S. The sequent*

$$\forall x_1 \ldots \forall x_m.\, c(S) \Rightarrow$$

is derivable in LK—i.e. the formula $\neg\forall x_1 \ldots \forall x_m.\, c(S)$ is valid—if and only if there exists a resolution tree under S whose root is the empty clause $\square$.

Proof. Soundness is easy by induction. For completeness *Skolemization* and *Herbrand's theorem*; Skolemization is explained later in §5.5.1. □

Although the above soundness and completeness are restricted to formulas of the form $\neg\forall x_1 \ldots \forall x_m.\, c(S)$, in fact we see in Section 5.5.1 that every formula can be transformed to a formula of the above form so that essentially the above theorem applicable to all formulas.

5.5.3 Example. (Taken from [15, pp. 142]) Consider the following assumptions:

– Isaac is a boy and Kate is a girl.

– Joe's friends are all tall.

– Harry loves any girl that is tall.

– Isaac and Kate are friends of Joe.

We are to examine the following question:

Does Harry love any of Joe's friends? If he does, who does he love? (5.5)

We can express these assumptions by the following clauses.

$$\{\, B(i) \,\} \quad \{\, G(k) \,\} \quad \{\, \neg F(j,x), T(x) \,\} \quad \{\, \neg G(y), \neg T(y), L(h,y) \,\}$$
$$\{\, F(j,i) \,\} \quad \{\, F(j,k) \,\}$$

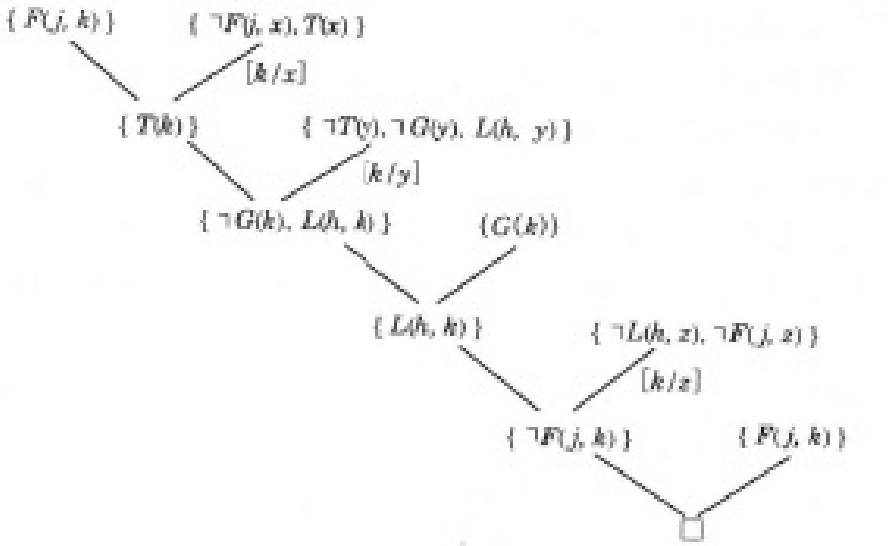
Here h, i, j, k are constants; and x, y are variables.

The question can now be formulated as a clause

$$\{\, \neg L(h,z), \neg F(j,z) \,\}$$

by taking the negation of the desired property. If we manage to derive a contradiction $\square$ from these clauses, then it means a positive answer to (5.5)—Harry loves some of Joe's friend.

Indeed, $\square$ is derived by the following resolution tree.



In the resolution tree, the variable z (the person the question (5.5) asks about) gets replaced with k. Therefore we see that Kate is an answer.

As we saw in the previous example, in resolution for predicate logic finding a suitable substitution is a key—given two formulas A and B, find a substitution θ so that $A\theta \equiv B\theta$. Such a substitution θ is called a *unifier* of A and B; the problem of finding a unifier is called *unification*.

5.5.4 Example (Unification)**.** An example from [15].

	$P(x, f(y))$	$P(g(z,z), z)$	$P(g(f(u), v), w)$
$g(z,z)/x$	$P(g(z,z), f(y))$	$P(g(z,z), z)$	$P(g(f(u), v), w)$
$f(y)/z$	$P(g(f(y), f(y)), f(y))$	$P(g(f(y), f(y)), f(y))$	$P(g(f(u), v), w)$
y/u	$P(g(f(y), f(y)), f(y))$	$P(g(f(y), f(y)), f(y))$	$P(g(f(y), v), w)$
$f(y)/v$	$P(g(f(y), f(y)), f(y))$	$P(g(f(y), f(y)), f(y))$	$P(g(f(y), f(y)), w)$
$f(y)/w$	$P(g(f(y), f(y)), f(y))$	$P(g(f(y), f(y)), f(y))$	$P(g(f(y), f(y)), f(y))$

Thus the substitution

$$\begin{aligned}\theta :=& [g(z,z)/x]\,[f(y)/z]\,[y/u]\,[f(y)/v]\,[f(y)/w] \\ =& [g(f(y), f(y))/x, f(y)/z, y/u, f(y)/v, f(y)/w]\end{aligned} \tag{5.6}$$

is a unifier. In fact it is the *most general unifier (mgu)*, in the following sense. The following substitution

$$\xi := [g(f(g(y, f(y))), f(g(y, f(y))))/x, f(g(y, f(y)))/z, \\ g(y, f(y))/u, f(g(y, f(y)))/v, f(g(y, f(y)))/w]$$

is another unifier but is more specialized; and it is obtained from the mgu θ by

$$\xi = \theta[g(y, f(y))/y] \ .$$

Any unifier σ *factors through* the mgu: there exists a substitution σ' such that $\sigma = \theta\sigma'$.

5.5.1 Prenex Normal Form and Skolemization

It must be noted that, in the resolution principle for predicate logic, there are *absolutely no* quantifiers around. Still Thm. 5.5.2 states that the formalism of resolution is powerful enough to derive all the valid formulas.[5] How is that possible?

We shall now exhibit one of the two crucial results for this—namely *Skolemization*.[6] It states that satisfiability of any formula can be reduced to that of a closed formula of the form

$$\forall x_1. \ldots \forall x_n. B \tag{5.7}$$

where B is quantifier-free.

5.5.5 Definition (Prenex normal form)**.** A predicate formula A is said to be in *prenex normal form* if it is in the following form:

$$A \equiv \mathsf{Q}x_1. \ldots \mathsf{Q}x_n. B$$

where Q is either $\forall$ or $\exists$, $x_1, \ldots, x_n$ are different variables, and B is a quantifier-free formula.

5.5.6 Lemma. *For any predicate formula A, there is a formula A^{pnf} in prenex normal form that is logically equivalent to A (i.e. $A \cong A^{\mathrm{pnf}}$).*

Proof. Use the following logical equivalences to pull quantifiers outwards. We assume that x does not occur freely in B.

$$\begin{array}{cc}
\forall x.A \wedge B \cong \forall x.(A \wedge B) & \exists x.A \wedge B \cong \exists x.(A \wedge B) \\
\forall x.A \vee B \cong \forall x.(A \vee B) & \exists x.A \vee B \cong \exists x.(A \vee B) \\
\forall x.A \supset B \cong \exists x.(A \supset B) & \exists x.A \supset B \cong \forall x.(A \supset B) \\
\neg\forall x.A \cong \exists x.\neg A & \neg\exists x.A \cong \forall x.\neg A
\end{array}$$

□

Further we eliminate all the existential quantifiers. Note that, unlike in Lem. 5.5.6, we lose logical equivalence (Exercise 5.11).

5.5.7 Definition (Skolemization)**.** Let

$$\begin{array}{l}
\forall x_{1,1}. \ldots . \forall x_{1,n_1}. \\
\exists y_1. \forall x_{2,1}. \ldots . \forall x_{2,n_2}. \\
\ldots \\
\exists y_m. \forall x_{m+1,1}. \ldots . \forall x_{m+1,n_{m+1}}. \quad B
\end{array}$$

be a formula in prenex normal form, with B quantifier-free. A *Skolemization* is a formula

$$\begin{array}{l}
\forall x_{1,1}. \ldots . \forall x_{1,n_1}. \\
\forall x_{2,1}. \ldots . \forall x_{2,n_2}. \\
\ldots \\
\forall x_{m+1,1}. \ldots . \forall x_{m+1,n_{m+1}}. \\
\qquad B\,[\, f_1(\overrightarrow{x_1}) \,/\, y_1,\ f_2(\overrightarrow{x_1}, \overrightarrow{x_2}) \,/\, y_2,\ \ldots,\ f_m(\overrightarrow{x_1}, \overrightarrow{x_2}, \ldots, \overrightarrow{x_m}) \,/\, y_m\,] \ .
\end{array}$$

[5] More precisely: resolution refutes all the unsatisfiable formulas.

[6] Named after a Norwegian logician, T. Skolem.

Here $f_1, \dots, f_m$ are "fresh" function symbols (they are not in the original **FnSymb**) of suitable arities; $f_m(\overrightarrow{x_1}, \overrightarrow{x_2}, \dots, \overrightarrow{x_m})$ is short for

$$f_m \ \big(x_{1,1}, \dots, x_{1,n_1}, \ \dots, \ x_{m,1}, \dots, x_{m,n_m} \big) \ .$$

The previous definition is a "hell of syntactic bureaucracy" but the idea is simple: you remove all the existential quantifiers, and replace each existentially quantified variable y_i by $f_i(\dots)$. The arguments of the fresh function symbol f_i are the universally quantified variables that are outside $\exists y_i$.

5.5.8 Example. A formula

$$\forall x.\forall y.\exists z.\forall u.\exists w.(P(f(x,w),y) \supset Q(y,z,g(u)))$$

is in a prenex normal form; we can use fresh function symbols h, i to Skolemize it:

$$\forall x.\forall y.\forall u.(P(f(x,i(x,y,u)),y) \supset Q(y,h(x,y),g(u))) \ .$$

5.5.9 Remark. Skolemization is in fact a common thing to do when you write a (not necessarily formal) mathematical proof. For example, let's say $f : \mathbb{R} \to \mathbb{R}$ is a continuous function. By definition,

> For each $x \in \mathbb{R}$ and $\varepsilon > 0$, there exists $\delta > 0$ such that $|x' - x| < \delta$ implies $|f(x') - f(x)| < \varepsilon$.

In using this fact in a proof, say, of the fact that $f + g$ is continuous if f and g are, you would say:

> Let $x \in \mathbb{R}$ and $\varepsilon > 0$; then we can take $\delta > 0$ such that: $|x' - x| < \delta$ implies $|f(x') - f(x)| < \varepsilon/2$.

This δ in fact depends on x and ε—thus a function $\delta(x, \varepsilon)$. This is the "fresh function symbol" used in Skolemization!

5.5.10 Proposition. *Let A be a formula in a prenex normal form; and A^{Sko} be a Skolemization of A. Then we have*

$$A \text{ is satisfiable} \quad \Longleftrightarrow \quad A^{\mathrm{Sko}} \text{ is satisfiable} \ .$$

Proof. Here we just sketch the simple case that A is of the form $\forall x.\exists y.B$ with B quantifier-free; a general proof is a straightforward extension and left as an exercise.

There is a fresh function symbol f of arity 1, such that

$$A^{\mathrm{Sko}} \ \equiv \ \forall x.\, B\big[f(x)/y\big] \ .$$

Assume A is satisfiable. We take $\mathbb{S}, J$ so that $[\![A]\!]_{\mathbb{S},J} = [\![\forall x.\exists y.B]\!]_{\mathbb{S},J} = \mathrm{tt}$. We extend $\mathbb{S}$ and J, and obtain a new structure $\mathbb{S}'$ for $\mathbf{FnSymb}' := \mathbf{FnSymb} \amalg \{f\}$, and J' over $\mathbb{S}'$. This is done by:

$$[\![f]\!]_{\mathbb{S}'}(u) := (v \in U \text{ such that } [\![B]\!]_{\mathbb{S},J[x \mapsto u, y \mapsto v]} = \mathrm{tt}) \ ;$$

note that, by $[\![\forall x.\exists y.B]\!]_{\mathbb{S},J} = \mathrm{tt}$, such v always exists for any u. We set $J' := J$. Then it is easy to see that $[\![A^{\mathrm{Sko}}]\!]_{\mathbb{S}',J'} = [\![\forall x.B[f(x)/y]]\!]_{\mathbb{S}',J'} = \mathrm{tt}$.

Conversely, assume A^{Sko} is satisfiable. Let $\mathbb{S}'$ and J' be such that $[\![A^{\mathrm{Sko}}]\!]_{\mathbb{S}',J'} = [\![\forall x.B[f(x)/y]]\!]_{\mathbb{S}',J'} = \mathrm{tt}$. Then the structure $\mathbb{S}$ for the original **FnSymb**, defined as the restriction of $\mathbb{S}'$, and $J' = J$ obviously satisfy $[\![A]\!]_{\mathbb{S},J} = [\![\forall x.\exists y.B]\!]_{\mathbb{S},J} = \mathrm{tt}$. □

Combining the results presented so far, we obtain the following procedure for checking by Thm. 5.5.2 if a given predicate formula A is valid.

$$\begin{aligned} A \text{ is valid} &\iff \neg A \text{ is not satisfiable} \\ &\iff (\neg A)^{\mathrm{pnf}} \text{ is not satisfiable} && \text{by Lem. 5.5.6} \\ &\iff ((\neg A)^{\mathrm{pnf}})^{\mathrm{Sko}} \text{ is not satisfiable} && \text{by Prop. 5.5.10.} \\ &\iff \neg((\neg A)^{\mathrm{pnf}})^{\mathrm{Sko}} \text{ is valid} \end{aligned}$$

(We remark that this also constitutes part of the completeness proof (Thm. 5.5.2).)

Exercises

5.1. Use Thm. 5.2.8 (strong completeness) to give another proof of completeness of propositional LK (Thm. 3.4.3).

5.2. Prove Thm. 5.2.9.

5.3. Prove Lem. 5.3.2.

5.4 (From [5])**.** Given the following hypotheses:

- If it rains, Joe brings his umbrella.
- If Joe has an umbrella, he doesn't get wet.
- If it doesn't rain, Joe doesn't get wet.

Use resolution to prove that Joe doesn't get wet.

Hint: express "it rains" by R, "Joe has an umbrella" by U, "Joe gets wet" by W.

5.5. Prove Lem. 5.4.11.

5.6. Complete the proof of Lem. 5.4.12.

5.7. Assume

$$\vdash L_1, \ldots, L_n \Rightarrow$$

where each L_i is a literal. Prove that there are $j, k \in [1, n]$ such that $L_j = L_k^*$.

5.8. Prove the soundness part of Thm. 5.5.2.

5.9. Prove the completeness part of Thm. 5.5.2. (Hard. You will most probably have to rely much on literature)

5.10. In the proof of Lem. 5.5.6:

1. Prove the listed logical equivalences. (The first one for $\supset$ can be understood as: "when everybody has finished eating, a taxi is coming")

2. Show that, if x occurs freely in B, then

$$\forall x.A \vee \forall B \cong \forall x.(A \vee B)$$

does not hold.

3. In the proof it is assumed that x does not occur freely in B. Justify this assumption.

5.11. Describe the relationship between the logical equivalence $A \cong B$ and the fact that

$$A \text{ is satisfiable} \iff B \text{ is satisfiable} .$$

Is there any implication between the two?

5.12. 1. Express the following (informal) statement as a predicate formula: "for any number, there is another number that is strictly smaller."

2. Skolemize the formula.

3. Show that the two formulas thus obtained are *not* logically equivalent.

5.13. Complete the proof of Prop. 5.5.10.

5.14 (Universal closure)**.** Let B be a formula, with $\mathrm{FV}(B) = \{x_1, \ldots, x_n\}$. Show that

$$B \text{ is valid} \iff \forall x_1. \ldots \forall x_n. B \text{ is valid.}$$

The formula $\forall x_1. \ldots \forall x_n. B$ is said to be a *universal closure* of B. Note that it is indeed a closed formula.

Part II

Computability

Preface to Part II

The central question in the second part of this textbook is simple:

What can "machines" do?

We will discuss, *in mathematically rigorous terms*, what are "machines," what they can do, and what they cannot. Remarkable points are:

- There are, indeed, things that machines *cannot do*. Moreover we can mathematically *prove* this.
- Different natural "definitions of machine"—Turing machine, λ-calculus, recursive function, while program—coincide in their capabilities.

As our definitions/formalizations/representations of machine, we use two:

- *recursive functions*,[7] which are convenient in mathematical reasoning; and
- *while-programs*, which are intuitive for many of the modern programmers.

The former are *mathematical* and *abstract* representation; the latter are more *concrete* and *operational* (you can more easily imagine how they operate step by step). We go back and forth between these two representations.[8]

In this part we will also see some basic and important techniques in theoretical computer science. To name a couple of them:

- The *Gödel numbers* that translates any syntactic formalism into natural numbers;
- the *diagonal method*.

The latter loosely means "negative self-reference," such as in the liar's paradox.

[7] Also called: *μ-recursive functions* or simply *computable functions*. The last name is more widely accepted in recent years. See [7].

[8] In the (numerous) textbooks on the theory of computability, it is also standard to use two formalisms, one abstract and one concrete. For the latter *Turing machines* are very often used.

Chapter 6

Recursive Function

First we introduce our first "definition" of machine, namely recursive function. We have

$$\begin{array}{rcl} \text{recursive function} & = & \text{primitive recursive function} \\ & + & \text{(unbounded) } \mu\text{-operator.} \end{array}$$

6.1 Primitive Recursive Function

6.1.1 Definition

We begin with introducing a more basic class of functions.

6.1.1 Notation ((Meta) $\boldsymbol{\lambda}$-notation)**.** We will be using a bold-style symbol $\boldsymbol{\lambda}$ to denote functions from $\mathbb{N}$ to $\mathbb{N}$. For example, $\boldsymbol{\lambda}x.\, x+1$ denotes a function

$$\boldsymbol{\lambda}x.\, x+1 \quad : \quad \mathbb{N} \longrightarrow \mathbb{N} \ , \qquad x \longmapsto x+1 \ .$$

More generally, the same $\boldsymbol{\lambda}$-notation is used for functions $\mathbb{N}^m \to \mathbb{N}$, as in

$$\boldsymbol{\lambda}(x_1, x_2, x_3).\, x_1 + x_2 + x_3 \quad : \quad \mathbb{N}^3 \longrightarrow \mathbb{N} \ .$$

Note that $\boldsymbol{\lambda}$ is a *meta* symbol, unlike the object level symbol λ in the λ-calculus (like in $\lambda x.\lambda y.\, xy$).

6.1.2 Definition (Primitive recursive function)**.** The class of *primitive recursive functions* is defined inductively as follows. We write "PR" for "primitive recursive."

- (Base cases)
  - The *zero function* $\mathsf{zero} : \mathbb{N}^0 \to \mathbb{N}$, defined by $\mathsf{zero}() = 0$, is PR.
  - The *successor function* $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$, defined by $\mathsf{succ}(x) = x + 1$, is PR.
  - A *projection* $\mathsf{proj}_i^n : \mathbb{N}^n \to \mathbb{N}$, defined by $\mathsf{proj}_i^n(x_0, \ldots, x_{n-1}) = x_i$ is PR. Here $i \in n = \{0, 1, \ldots, n-1\}$.

– (Composition) Assume $g : \mathbb{N}^m \to \mathbb{N}$ is PR, and the functions $g_0, \ldots, g_{m-1} : \mathbb{N}^n \to \mathbb{N}$ (with the same arity) are all PR, too. Then the function

$$\boldsymbol{\lambda}(x_0, \ldots, x_{n-1}).\ g\big(\, g_0(x_0, \ldots, x_{n-1}), \ldots, g_{m-1}(x_0, \ldots, x_{n-1})\,\big)$$

is PR.

– (Primitive recursion) Let $g : \mathbb{N}^n \to \mathbb{N}$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}$ be PR functions. Then the function $f : \mathbb{N}^{n+1} \to \mathbb{N}$, defined by

$$\begin{aligned} f(\overrightarrow{x}, 0) &:= g(\overrightarrow{x})\ , \\ f(\overrightarrow{x}, y+1) &:= h(\overrightarrow{x}, y, f(\overrightarrow{x}, y)) \end{aligned} \tag{6.1}$$

is PR. Here $\overrightarrow{x}$ is short for $x_0, \ldots, x_{n-1}$.

That is, in a simple rule-based presentation:

$$\frac{}{\mathsf{zero}\text{ is PR}}\ (\mathsf{zero}) \qquad \frac{}{\mathsf{succ}\text{ is PR}}\ (\mathsf{succ}) \qquad \frac{}{\mathsf{proj}_i^n\text{ is PR}}\ (\mathsf{proj}_i^n)$$

$$\frac{g \text{ is PR} \quad g_1 \text{ is PR} \quad \cdots \quad g_m \text{ is PR}}{\boldsymbol{\lambda}\overrightarrow{x}.\, g(g_1(\overrightarrow{x}), \ldots, g_m(\overrightarrow{x})) \text{ is PR}}\ (\text{Comp})$$

$$\frac{g \text{ is PR} \quad h \text{ is PR} \quad f(\overrightarrow{x}, 0) = g(\overrightarrow{x}) \quad f(\overrightarrow{x}, y+1) = h(\overrightarrow{x}, y, f(\overrightarrow{x}, y))}{f \text{ is PR}}\ (\text{PR})$$

Recall that $\mathbb{N}^0 \cong 1$; hence a 0-ary function $\mathbb{N}^0 \to \mathbb{N}$ can be thought of as an element of $\mathbb{N}$.

6.1.3 Remark. Observe that

$$\{\text{primitive recursive functions}\} \subseteq \bigcup_{m \in \mathbb{N}} (\mathbb{N}^m \to \mathbb{N})\ ,$$

where $\mathbb{N}^m \to \mathbb{N} = \mathbb{N}^{(\mathbb{N}^m)}$ is the function space from $\mathbb{N}^m$ to $\mathbb{N}$.

6.1.4 Remark. Note that, in Def. 6.1.2, the functions zero, f, g, etc. are all *mathematical* entities. There are no syntactic symbols, or "syntax," there.

6.1.2 Some Examples

6.1.5 Example (Identity)**.** The *identity function*

$$\mathrm{id}_{\mathbb{N}} \quad : \quad \mathbb{N} \longrightarrow \mathbb{N}\ , \quad x \longmapsto x$$

is PR, since $\mathrm{id}_{\mathbb{N}} = \mathsf{proj}_0^1$.

6.1.6 Example (Predecessor)**.** The *predecessor function* $\mathsf{pred} : \mathbb{N} \to \mathbb{N}$, defined by

$$\mathsf{pred}(x) := \begin{cases} 0 & \text{if } x = 0; \\ x - 1 & \text{if } x > 0 \end{cases}$$

is PR. Indeed, pred can be defined via primitive recursion:

$$\mathsf{pred}(0) = \mathsf{zero}\ ; \qquad \mathsf{pred}(y+1) = \mathsf{proj}_0^2\big(\, y,\, \mathsf{pred}(y)\,\big)\ .$$

Note that the last expression $\mathsf{proj}_0^2\big(\, y,\, \mathsf{pred}(y)\,\big)$ is equal to y; we need this complicated expression to match the format (6.1) of primitive recursion.

Projections allow us to use "any argument, in any order."

6.1.7 Lemma. *Let $i_0, \dots, i_{m-1} \in n = \{0, \dots, n-1\}$, and $f : \mathbb{N}^m \to \mathbb{N}$ be a PR function. Then the function*

$$\boldsymbol{\lambda}(x_0, \dots, x_{n-1}).\ f(x_{i_0}, x_{i_1}, \dots, x_{i_{m-1}}) \quad : \quad \mathbb{N}^n \longrightarrow \mathbb{N}$$

is PR.

Proof. Let $\overrightarrow{x}$ stand for $x_0, \dots, x_{n-1}$. The function is the same as

$$\boldsymbol{\lambda}(x_0, \dots, x_{n-1}).\ f\big(\,\mathsf{proj}^n_{i_0}(\overrightarrow{x}), \dots, \mathsf{proj}^n_{i_{m-1}}(\overrightarrow{x})\,\big)\ ,$$

which is PR by the composition rule. □

Most of the "usual operations" on natural numbers are PR. For the proof you need *programming* in PR functions.

6.1.8 Example (Addition)**.** The function $\mathsf{add} : \mathbb{N}^2 \to \mathbb{N}$, defined by $\mathsf{add}(x, y) := x + y$, is PR. Indeed,

$$\mathsf{add}(x, 0) = x\ ; \qquad \mathsf{add}(x, y+1) = \mathsf{succ}(\mathsf{add}(x, y))\ ; \tag{6.2}$$

it is straightforward that this definition indeed determines a PR function (Exercise 6.1).

6.1.9 Example (Normalized subtraction $\dot{-}$)**.** The function $\mathsf{subtr} : \mathbb{N}^2 \to \mathbb{N}$, defined by

$$\mathsf{subtr}(x, y)\ := \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{otherwise} \end{cases}$$

is PR. We denote this *normalized subtraction operation* by $x \dot{-} y$.

6.1.10 Example. The following functions are all PR.

$$\begin{array}{ll} \mathsf{mult} : \mathbb{N}^2 \longrightarrow \mathbb{N}\ , & \mathsf{mult}(x, y) := x \cdot y\ ; \\ \mathsf{exp} : \mathbb{N}^2 \longrightarrow \mathbb{N}\ , & \mathsf{exp}(x, y) := x^y\ ; \\ \mathsf{fact} : \mathbb{N} \longrightarrow \mathbb{N}\ , & \mathsf{fact}(x) := x! = x \cdot (x-1) \cdot \dots \cdot 2 \cdot 1\ . \end{array}$$

6.1.11 Lemma (Bounded sum/product)**.** *Let $f : \mathbb{N}^{n+1} \to \mathbb{N}$ be a PR function. The functions*

$$\boldsymbol{\lambda}(\overrightarrow{x}, y).\ \sum_{z<y} f(\overrightarrow{x}, z)\ :\ \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$$

$$\boldsymbol{\lambda}(\overrightarrow{x}, y).\ \prod_{z<y} f(\overrightarrow{x}, z)\ :\ \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$$

are PR. Here $\overrightarrow{x}$ is short for $x_0, \dots, x_{n-1}$; given $(\overrightarrow{x}, y)$ as an argument, the functions produces the outputs

$$\sum_{z<y} f(\overrightarrow{x}, z) = f(\overrightarrow{x}, 0) + \dots + f(\overrightarrow{x}, y-1)\ ;$$

$$\prod_{z<y} f(\overrightarrow{x}, z) = f(\overrightarrow{x}, 0) \cdot \dots \cdot f(\overrightarrow{x}, y-1)\ ,$$

respectively.

Let us give a detailed proof for once. We do the sum part.

Proof. Let us temporarily write $F : \mathbb{N}^{n+1} \to \mathbb{N}$ for the former "bounded sum" function. Then we have

$$F(\overrightarrow{x}, 0) = \mathsf{zero} \ , \qquad F(\overrightarrow{x}, y+1) = \mathsf{add}\big(F(\overrightarrow{x}, y), f(\overrightarrow{x}, y) \big) \ . \tag{6.3}$$

Therefore, in view of (6.1), it suffices to show that the functions

$$\boldsymbol{\lambda}\overrightarrow{x}\,.\,\mathsf{zero} \quad \text{and} \quad \boldsymbol{\lambda}(\overrightarrow{x}, y, z).\,\mathsf{add}\big(z, f(\overrightarrow{x}, y)\big)$$

are PR. The former is obvious by the (Composition) rule of Def. 6.1.2; the latter is obvious, too, since

$$\boldsymbol{\lambda}(\overrightarrow{x}, y, z).\, z \text{ is PR by Lem. 6.1.7;} \tag{6.4}$$
$$\boldsymbol{\lambda}(\overrightarrow{x}, y, z).\, f(\overrightarrow{x}, y) \text{ is PR by Lem. 6.1.7;} \tag{6.5}$$
$$\boldsymbol{\lambda}(\overrightarrow{x}, y, z).\, \mathsf{sum}\big(z, f(\overrightarrow{x}, y) \big) \text{ is PR} \tag{6.6}$$
$$\text{by (6.4), (6.5) and the (Composition) rule of Def. 6.1.2.} \tag{6.7}$$

This concludes the proof. □

6.1.3 Primitive Recursive Predicate

6.1.12 Definition (Predicate)**.** A *predicate* (of arity n) is a subset

$$P \subseteq \mathbb{N}^n \ .$$

Note again that a predicate $P \subseteq \mathbb{N}^n$ is a mathematical—or "semantic"—entity; it is different from a *predicate symbol* in Chap. 4.

Recall from Def. 1.2.13 the *characteristic function* $\chi_P : \mathbb{N}^n \to 2$ for a subset $P \subseteq \mathbb{N}^n$. It returns 0 ("true") if the argument belongs to P; it returns 1 ("false") otherwise.

6.1.13 Definition (Primitive recursive predicate)**.** A predicate $P \subseteq \mathbb{N}^n$ is said to be *primitive recursive* (PR) if the composition

$$\mathbb{N}^n \xrightarrow{\chi_P} 2 \hookrightarrow \mathbb{N}$$

is a PR function $\mathbb{N}^n \to \mathbb{N}$. Here the function $2 \hookrightarrow \mathbb{N}$ is the *inclusion* function given by $0 \mapsto 0$ and $1 \mapsto 1$.

6.1.14 Example.

1. The 1-ary predicate $(\_ = 0)$, i.e. $\{0\} \subseteq \mathbb{N}$ is PR. Indeed, we have
$$\chi_{(\_=0)}(x) = 1 \mathbin{\dot{-}} (1 \mathbin{\dot{-}} x) \ ,$$
where $\dot{-}$ is the normalized subtraction operation from Example 6.1.9.

2. The 2-ary predicate $=$—which we can also denote by $(\__1 = \__2)$ or by $\boldsymbol{\lambda}(x_1, x_2).\, x_1 = x_2$—is PR. It is, as a subset,
$$\{(x, y) \mid x = y\} \subseteq \mathbb{N}^2 \ ;$$
its characteristic function is
$$\chi_{=}(x, y) = \chi_{(\_=0)}\big((x \mathbin{\dot{-}} y) + (y \mathbin{\dot{-}} x) \big) \ .$$

3. The 2-ary predicate $\leq$ is PR; its characteristic function is

$$\chi_{\leq}(x, y) = \chi_{(\_=0)}(x \dot{-} y) \ .$$

The class of PR predicates is closed under the Boolean operations.

6.1.15 Lemma. *Let $P, Q \subseteq \mathbb{N}^n$ be PR predicates. Then the predicates*

$$\neg P \ , \quad P \vee Q \ , \quad P \wedge Q \ ,$$

which are, as subsets,

$$\mathbb{N}^n \setminus P \ , \quad P \cup Q \ , \quad P \cap Q \ ,$$

are also PR. □

It is also true under *bounded quantifiers*. Note the arities of the predicates; in particular, y is an argument of the predicates.

6.1.16 Lemma. *Let $P \subseteq \mathbb{N}^{n+1}$ be a PR predicate. Then the $(n+1)$-ary predicates*

$$\boldsymbol{\lambda}(\overrightarrow{x}, y).\ \big(\forall_{z<y}.\, P(\overrightarrow{x}, z)\big) \qquad \boldsymbol{\lambda}(\overrightarrow{x}, y).\ \big(\exists_{z<y}.\, P(\overrightarrow{x}, z)\big)$$

are both PR.

The former predicate is true if and only if all of

$$P(\overrightarrow{x}, 0) \ , \quad P(\overrightarrow{x}, 1) \ , \quad \ldots, \quad P(\overrightarrow{x}, y-1)$$

are true. The latter is true if at least one of these predicates is true.

Proof. For the former predicate, its characteristic function is given by

$$\chi_{(\_=0)}\big(\sum_{z<y} \chi_P(\overrightarrow{x}, z)\big) \ .$$

This is a PR function by Lem. 6.1.11. For the latter:

$$\prod_{z<y} \chi_P(\overrightarrow{x}, z) \ . \qquad \square$$

6.1.17 Lemma. *Let $P \subseteq \mathbb{N}^m$ be a PR predicate; and $f_0, \ldots, f_{m-1} : \mathbb{N}^n \to \mathbb{N}$ be PR functions. Then the predicate*

$$\boldsymbol{\lambda}\overrightarrow{x}.\, P\big(f_0(\overrightarrow{x}), \ldots, f_{m-1}(\overrightarrow{x})\big) \ ,$$

where $\overrightarrow{x}$ is short for $x_0, \ldots, x_{n-1}$, is a PR predicate. □

6.1.18 Lemma (Case distinction)**.** *Let $P_0, \ldots, P_{n-1} \subseteq \mathbb{N}^m$ be PR predicates such that, for any $\overrightarrow{x} \in \mathbb{N}^m$, exactly one out of $P_0(\overrightarrow{x}), \ldots, P_{n-1}(\overrightarrow{x})$ is true. Furthermore, let $g_0, \ldots, g_{n-1} : \mathbb{N}^m \to \mathbb{N}$ be PR functions. Then the function $f : \mathbb{N}^m \to \mathbb{N}$, defined by*

$$f(\overrightarrow{x}) \ := \ \begin{cases} g_0(\overrightarrow{x}) & \text{if } P_0 \text{ is true} \\ g_1(\overrightarrow{x}) & \text{if } P_1 \text{ is true} \\ \ldots & \\ g_{n-1}(\overrightarrow{x}) & \text{if } P_{n-1} \text{ is true,} \end{cases}$$

is a PR function.

Proof.

$$f(\overrightarrow{x}) = \sum_{i<n} \big(1 \dot{-} \chi_{P_i}(\overrightarrow{x})\big) \cdot g_i(\overrightarrow{x}) \qquad \square$$

6.1.19 Example. The functions $\mathsf{max}, \mathsf{min} : \mathbb{N}^2 \to \mathbb{N}$ are PR.

We note that every PR function is total—all the functions have been of the type $\mathbb{N}^m \to \mathbb{N}$. We will next introduce a wider class of *recursive functions*; these are not necessarily total.

We present some further examples that are used later.

6.1.20 Lemma. *Let $P \subseteq \mathbb{N}^{n+1}$ be a PR predicate. The function*

$$\boldsymbol{\lambda}(\overrightarrow{x}, y).\quad \big(\mu_{z<y}.\, P(\overrightarrow{x}, z)\big) \qquad : \qquad \mathbb{N}^{n+1} \longrightarrow \mathbb{N}\ ,$$

is PR. Here the number $\mu_{z<y}.\, P(\overrightarrow{x}, z)$ is defined by:

- *if any of $P(\overrightarrow{x}, 0), P(\overrightarrow{x}, 1), \ldots, P(\overrightarrow{x}, y-1)$ is true, then*

$$\mu_{z<y}.\, P(\overrightarrow{x}, z) \quad := \quad \textit{(the least } z \textit{ such that } P(\overrightarrow{x}, z) \textit{ is true);}$$

- *if none of $P(\overrightarrow{x}, 0), P(\overrightarrow{x}, 1), \ldots, P(\overrightarrow{x}, y-1)$ is true, then*

$$\mu_{z<y}.\, P(\overrightarrow{x}, z) \quad := \quad y\ .$$

The operator $\mu_{z<y}$ is called bounded minimization.

Proof. (Sketch) Its characteristic function can be given by

$$\begin{aligned} &\chi_P(\overrightarrow{x}, 0) \\ &+ \chi_P(\overrightarrow{x}, 0) \cdot \chi_P(\overrightarrow{x}, 1) \\ &+ \cdots \\ &+ \chi_P(\overrightarrow{x}, 0) \cdot \chi_P(\overrightarrow{x}, 1) \cdot \cdots \cdot \chi_P(\overrightarrow{x}, y-1)\ . \end{aligned}$$

$\square$

6.1.21 Example. 1. The function $\mathsf{div} : \mathbb{N}^2 \to \mathbb{N}$, defined by $\mathsf{div}(x, y) := x \div y$, is PR. Indeed:

$$\mathsf{div}(x, y) \ = \ \big(\mu_{z<x+1}.\, x < z \cdot y\big) \dot{-} 1\ .$$

2. The function $\mathsf{rem} : \mathbb{N}^2 \to \mathbb{N}$, where $\mathsf{rem}(x, y)$ is the remainder when x is divided by y, is also PR.

6.1.22 Lemma. *Let $f : \mathbb{N} \to \mathbb{N}$ be a PR function. Let $f^{\#} : \mathbb{N}^2 \to \mathbb{N}$ be the function that applies f multiple times. Concretely,*

$$f^{\#}(x, 0) \ := \ x\ ; \qquad f^{\#}(x, y+1) \ := \ f(f^{\#}(x, y))\ .$$

This function $f^{\#}$ is PR. $\square$

6.1.23 Lemma. *1. The predicate $\mathsf{prime} \subseteq \mathbb{N}$, where $\mathsf{prime}(x)$ is true if and only if x is a prime number, is PR.*

2. The function $\mathrm{pr} : \mathbb{N} \to \mathbb{N}$, mapping x to the x-th smallest prime number, is PR.

Proof. Exercise 6.8. $\square$

6.2 Recursive Function

It is not hard to see that primitive recursive functions do not cover all the functions that are "computable." Consider the class of functions that are represented by some while-programs; they are naturally thought of as computable. However they are not necessarily total functions. For example, the program

```
z := 0;
while (z + 1 != 0) {
  z := z + 1
}
```

does not terminate. In the formalism of (primitive) recursive function, such "while-loops" are introduced in the form of the *minimization*[1] operator μ.[2]

6.2.1 Definition

Compare the next definition with Def. 6.1.2 of PR function; the principal difference is in the minimization operation. Note also that a recursive function is in general a *partial* function $\mathbb{N}^m \rightharpoonup \mathbb{N}$; partiality results from minimization.

6.2.1 Definition (Recursive function)**.** The class of *recursive functions* is defined inductively as follows.

- (Base cases)
  - $\mathsf{zero} : \mathbb{N}^0 \to \mathbb{N}$ is recursive.
  - $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$ is recursive.
  - $\mathsf{proj}_i^n : \mathbb{N}^n \to \mathbb{N}$ is recursive.
- (Composition) Assume $g : \mathbb{N}^m \rightharpoonup \mathbb{N}$ and $g_0, \ldots, g_{m-1} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ are all recursive functions. Then the partial function

  $$\boldsymbol{\lambda}(x_0, \ldots, x_{n-1}).\, g\big(\, g_0(x_0, \ldots, x_{n-1}), \ldots, g_{m-1}(x_0, \ldots, x_{n-1})\,\big) \quad : \quad \mathbb{N}^n \rightharpoonup \mathbb{N}$$

  is a recursive function. Here the value

  $$g\big(\, g_0(x_0, \ldots, x_{n-1}), \ldots, g_{m-1}(x_0, \ldots, x_{n-1})\,\big)$$

  is
  - defined if all of the values

    $$g_0(x_0, \ldots, x_{n-1}),\ \ldots,\ g_{m-1}(x_0, \ldots, x_{n-1}),\ \text{and}$$
    $$g\big(\, g_0(x_0, \ldots, x_{n-1}), \ldots, g_{m-1}(x_0, \ldots, x_{n-1})\,\big)$$

    are defined; and
  - undefined if any of these values is undefined.

[1] Also called: the *least solution* operator.
[2] There are total computable functions that are *not* primitive recursive; see Example 6.2.5.

– (Primitive recursion) Let $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$ be recursive functions. Then the partial function $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$, defined by

$$\begin{aligned} f(\overrightarrow{x}, 0) &:= g(\overrightarrow{x}) \ , \\ f(\overrightarrow{x}, y+1) &:= h(\overrightarrow{x}, y, f(\overrightarrow{x}, y)) \end{aligned} \tag{6.8}$$

is a recursive function.

– (Minimization) Let $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ be a recursive function. The function

$$\boldsymbol{\lambda}\overrightarrow{x}. \quad \big(\mu_y.\, f(\overrightarrow{x}, y) = 0\big) \qquad : \quad \mathbb{N}^n \rightharpoonup \mathbb{N}$$

is a recursive function. Here the *least solution* $\mu_y.\, f(\overrightarrow{x}, y) = 0$ is

- defined to be z, in case
  - $f(\overrightarrow{x}, 0), f(\overrightarrow{x}, 1), \ldots, f(\overrightarrow{x}, z)$ are all defined,
  - $f(\overrightarrow{x}, 0), f(\overrightarrow{x}, 1), \ldots, f(\overrightarrow{x}, z-1)$ are all $\neq 0$, and
  - $f(\overrightarrow{x}, z) = 0$;
- undefined otherwise.

The minimization operation looks at the values of $f(\overrightarrow{x}, 0), f(\overrightarrow{x}, 1), \ldots$ one by one, checks if they are equal to 0, and returns the least index z for which $f(\overrightarrow{x}, z)$ is 0. The number $\mu_y.\, f(\overrightarrow{x}, y) = 0$ is undefined if:

– $f(\overrightarrow{x}, 0), f(\overrightarrow{x}, 1), \ldots$ are all defined and $\neq 0$, or

– there does exist the smallest z such that $f(\overrightarrow{x}, z) = 0$, but before reaching this z, there is some $y < z$ with which $f(\overrightarrow{x}, y)$ is undefined (note that f is in general a partial function).

We can think of the fact that $f(\overrightarrow{x}, y)$ is undefined as *non-termination* or "taking infinitely long for computation."

6.2.2 Example. The partial function

$$\mu_y.\, (y + 1 = 0)$$

is a 0-ary recursive function $\mathbb{N}^0 \rightharpoonup \mathbb{N}$. It is as a function $\emptyset \to \mathbb{N}$ where $\emptyset \subseteq \mathbb{N}^0$— i.e. its value is not defined for the unique input.

We also note that the partial function

$$0 \cdot \big(\mu_y.\, (y + 1 = 0)\big)$$

is a 0-ary recursive function $\mathbb{N}^0 \rightharpoonup \mathbb{N}$. By the (Composition) part of Def. 6.2.1, its value is not defined. ("Undefined times zero is undefined")

6.2.3 Definition (Total recursive function)**.** A *total recursive function* is a recursive function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ that is defined for any $\overrightarrow{x} \in \mathbb{N}^n$.

The following is the precise definition of two partial functions being "equal."

6.2.4 Definition (Kleene equality)**.** Let $f, g : \mathbb{N}^m \rightharpoonup \mathbb{N}$ be partial functions. The *Kleene equality* $f \doteq g$ is defined as follows.

$$f \doteq g \quad \overset{\text{def}}{\iff} \quad \text{for each } x_0, \ldots, x_{m-1} \in \mathbb{N},$$
$$\begin{cases} \text{neither } f(x_0, \ldots, x_{m-1}) \text{ nor } g(x_0, \ldots, x_{m-1}) \text{ is defined, or} \\ \text{both of them are defined and } f(x_0, \ldots, x_{m-1}) = g(x_0, \ldots, x_{m-1}). \end{cases}$$

That is: $f \doteq g$ means f and g are the same in

– whether they are defined, and

– their values (when defined).

Here is a famous example of a total recursive function that is *not* primitive recursive.

6.2.5 Example (The Ackermann function)**.** Consider the following function $A : \mathbb{N}^2 \to \mathbb{N}$.[3]

$$A(x,y) := \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } x > 0 \text{ and } y = 0 \\ A(x-1, A(x,y-1)) & \text{if } x > 0 \text{ and } y > 0 \end{cases} \tag{6.9}$$

This function A is well-defined (which is not trivial—see Exercise 6.9). It is recursive; while showing that directly (i.e. giving a μ-expression that computes A) is hard, a while-program for A is easy to come up with, and we can use the equivalence between recursive functions and while-programs (Cor. 7.1.9). It is not primitive recursive; it grows faster than any PR function (Exercise 6.10). Note that the definition (6.9) is *not* primitive recursion on x, y (cf. Exercise 6.11).

6.2.2 Recursive Predicate

6.2.6 Definition. A predicate $P \subseteq \mathbb{N}^n$ is said to be *recursive* if its characteristic function $\chi_P : \mathbb{N}^n \to \mathbb{N}$ is a total recursive function.

A recursive predicate is also called a *decidable predicate*.

Note the totality requirement on χ_P. A characteristic function, by its definition, must be always total (Def. 1.2.13).

6.2.7 Lemma. *Let $P, Q \subseteq \mathbb{N}^n$ be recursive predicates. Then the predicates*

$$\neg P \ , \quad P \vee Q \ , \quad P \wedge Q \ ,$$

which are, as subsets,

$$\mathbb{N}^n \setminus P \ , \quad P \cup Q \ , \quad P \cap Q \ ,$$

are also recursive. □

6.2.8 Lemma (Case distinction)**.** *Let $P_0, \ldots, P_{n-1} \subseteq \mathbb{N}^m$ be recursive predicates such that, for any $\overrightarrow{x} \in \mathbb{N}^m$, exactly one out of $P_0(\overrightarrow{x}), \ldots, P_{n-1}(\overrightarrow{x})$ is true. Furthermore, let $g_0, \ldots, g_{n-1} : \mathbb{N}^m \rightharpoonup \mathbb{N}$ be recursive functions. Then the partial function $f : \mathbb{N}^m \rightharpoonup \mathbb{N}$, defined by*

$$f(\overrightarrow{x}) := \begin{cases} g_0(\overrightarrow{x}) & \text{if } P_0(\overrightarrow{x}) \text{ is true} \\ g_1(\overrightarrow{x}) & \text{if } P_1(\overrightarrow{x}) \text{ is true} \\ \ldots & \\ g_{n-1}(\overrightarrow{x}) & \text{if } P_{n-1}(\overrightarrow{x}) \text{ is true,} \end{cases} \tag{6.10}$$

is a recursive function.

[3] There are many variations that are called an "Ackermann function"; the function A here is one of the simplest.

We defer the proof to pp. 116 in Chap. 8, since it calls for a machinery of universal recursive function.

6.2.9 Remark. Note that the proof of Lem. 6.2.8 can *not* be done in the same way as in Lem. 6.1.18. Consider, as an example, the partial function:

$$f(x) := \begin{cases} 0 & \text{if } x = 0, \\ \mu y.\ y + 1 = 0 & \text{if } x > 0, \end{cases}$$

where $\mu y.\ y+1 = 0$ is the "undefined value" in Example 6.2.2. By the definition, the value $f(0)$ is indeed defined and is 0. However, if we are to do the same proof as the one for Lem. 6.1.18, we need to have

$$f(x) \quad = \quad (1 \dot{-} \chi_{(x=0)}) \cdot 0 \ + \ (1 \dot{-} \chi_{\neg(x=0)}) \cdot (\mu y.\ y + 1 = 0) \ .$$

This value is not defined even for $x = 0$, since the second "irrelevant" summand

$$(1 \dot{-} \chi_{\neg(x=0)}) \cdot (\mu y.\ y + 1 = 0)$$

evaluates to "undefined."

Exercises

6.1. Give a detailed proof that the definition (6.2) indeed determines a PR function.

6.2. Prove that the functions subtr, mult, exp, and fact (Examples 6.1.9–6.1.10) are all PR.

6.3. Formulate a result similar to Lem. 6.1.7 for PR predicates, and prove it.

6.4. Prove Lem. 6.1.15.

6.5. Prove Lem. 6.1.17.

6.6. Prove that max and min (Example 6.1.19) are both PR.

6.7. Fill in the details of the proof of Lem. 6.1.20.

6.8. Prove Lem. 6.1.23. Hint: for 1., use a bounded quantifier (Lem. 6.1.16). For 2., use a bounded least solution operator (Lem. 6.1.20); note that, given the x-th prime $\mathrm{pr}(x)$, the $(x+1)$-st prime is not bigger than the prime number $\mathrm{pr}(x)! + 1$.

6.9. Prove that the function A in Example 6.2.5 is well-defined. Hint: use the lexicographic order on $\mathbb{N}^2$.

6.10. Consider the function A in Example 6.2.5.

1. Let $f : \mathbb{N}^m \to \mathbb{N}$ be any PR function. Show that there exists a natural number $z \in \mathbb{N}$ such that

   $$f(x_0, \ldots, x_{m-1}) < A(z, x_0 + \cdots + x_{m-1}) \qquad \text{for any } x_0, \ldots, x_{m-1} \in \mathbb{N}.$$

   Note: z is the same for all $x_0, \ldots, x_{m-1}$. Hint: use induction along the definition of PR function.

2. Conclude that $A : \mathbb{N}^2 \to \mathbb{N}$ is not PR. Hint: this is not very easy! You use a diagonal argument.

6.11. Consider the function A in Example 6.2.5. While the function $A : \mathbb{N}^2 \to \mathbb{N}$ itself is not PR, when we fix its first argument, the function

$$A(x, \_) \ : \ \mathbb{N} \longrightarrow \mathbb{N} \ , \qquad y \longmapsto A(x, y)$$

is PR, for any $x \in \mathbb{N}$. Prove this.

Chapter 7

Recursive Function and While Program

In this chapter we introduce another formalism for computation—namely *while program*—and show that it is equivalent to the formalism of recursive function. This allows us to prove some useful *structural* properties of recursive functions.

As already noted, it is more common to use Turing machines instead of while programs for the second, more operational, formalism. Our choice here follows [11, 13].

7.1 While Program

For the rigorous development of the theory, it is desirable to define precisely what a while program is and what function it computes. However, it incurs a lot of uninspiring details and in the current notes we favor intuitions over rigor and precision. Thus we will be informal when speaking about while programs.

The set of *while programs* is the core of the imperative languages like C. An example is:

$i := 1; \qquad j := x_0;$
`while` $(j \neq 0)$ {
  $i := i \times j$;
  $j := j \dot{-} 1$
};
`return` i

which is easily seen to compute the factorial $x_0! = x_0 \cdot (x_0 - 1) \cdot \cdots \cdot 2 \cdot 1$.

In while programs:

- every variable carries a natural number as its value;
- we can use input values (which we always denote by $x_0, \ldots, x_{n-1}$);
- we can use arithmetic operations $+, \dot{-}, \times$;
- we can use conditionals by $=, \neq, <$ and their Boolean combinations;
- we can use if-branches `if` ... `then` ... `else` ...;
- we can use while-loops `while` ... {...}; and

– the output value is designated by the `return` construct.

Then it is straightforward to see that any recursive function can be computed by some while program.

7.1.1 Theorem. *For any recursive function $f : \mathbb{N}^m \rightharpoonup \mathbb{N}$, there is a while program p that computes $f(x_0, \ldots, x_{m-1})$.*

Proof. (Sketch) For zero, succ and proj^n_j, we write

$$\texttt{return}\quad 0\quad , \qquad \texttt{return}\quad x_0 + 1\quad , \qquad \texttt{return}\quad x_i\quad ,$$

respectively.

For the composition

$$\boldsymbol{\lambda}(x_0, \ldots, x_{n-1}).\quad g\big(\, g_0(x_0, \ldots, x_{n-1}), \ldots, g_{m-1}(x_0, \ldots, x_{n-1})\,\big)\ ,$$

assume that the program

$$\boxed{\mathsf{p}_i}$$
$$\texttt{return}\quad r_i$$

computes the function g_i, for each $i \in [0, m-1]$. Furthermore, assume that the program

$$\boxed{\mathsf{q}}$$
$$\texttt{return}\quad r$$

computes the function g. Then the program

$$\boxed{\mathsf{p}_0}$$
$$\vdots$$
$$\boxed{\mathsf{p}_{m-1}}$$
$$\boxed{\mathsf{q}[r_0/x_0, \ldots, r_{m-1}/x_{m-1}]}$$
$$\texttt{return}\quad r[r_0/x_0, \ldots, r_{m-1}/x_{m-1}]$$

computes the desired composition. Here $[r_0/x_0, \ldots, r_{m-1}/x_{m-1}]$ represents suitable substitution.

For primitive recursion (6.8), we write[1]

$$\begin{array}{l} r := g(\overrightarrow{x});\quad j := 0; \\ \texttt{while}\quad j \neq y\quad \{ \\ \quad r := h(\overrightarrow{x}, j, r) \\ \quad j := j + 1 \\ \}; \\ \texttt{return}\quad r \end{array}$$

Finally, for minimization

$$\boldsymbol{\lambda}\overrightarrow{x}.\quad \big(\, \mu_y.\ f(\overrightarrow{x}, y) = 0\,\big)\ ,$$

[1] Here we are even more informal, writing recursive functions explicitly in a while program. It is straightforward to replace such explicit occurrences of recursive functions with the while programs that compute them.

we can write the following while program.

$$
\begin{array}{l}
r := 0; \\
\texttt{while} \quad f(\overrightarrow{x}, r) \neq 0 \quad \{ \\
\quad r := r + 1 \\
\}; \\
\texttt{return} \quad r
\end{array}
$$

□

Not so easy is the converse, that is, the fact that the function computed by a while program is a recursive function. Towards this goal let us first consider only "normalized" while programs.

7.1.2 Definition (Normalized while program)**.** A while program is *normalized* if it is of the following form:

$$
\begin{array}{ll}
w := e(x_0, \ldots, x_{n-1}); & \text{(* encode input into a single variable } w\text{*)} \\
\texttt{while} \quad q(w) \neq 0 \quad \{ & \\
\quad w := g(w) & \text{(* update } w \text{ *)} \\
\}; & \\
y := h(w); & \text{(* prepare an output *)} \\
\texttt{return} \quad y &
\end{array}
\tag{7.1}
$$

where e, g, h, q are primitive recursive functions (here we have extended the syntax of while programs to allow explicit PR functions).

7.1.3 Lemma. *Let e, g, h, q be primitive recursive functions. Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be the partial function computed by the normalized program (7.1). Then f is recursive.*

Note that $f(\overrightarrow{x})$ is undefined if and only if the program does not terminate.

Proof. (Sketch)

$$
f(\overrightarrow{x}) = h\left(g^{\#}\left(e(\overrightarrow{x}), \quad \mu_z.\left(q\Big(g^{\#}\big(e(\overrightarrow{x}), z \big) \Big) = 0 \right) \right) \right) \ ,
$$

where $g^{\#}$ is as in Lem. 6.1.22. Note here that $\mu_z.(q(g^{\#}(e(\overrightarrow{x}), z)) = 0)$ is the number of loop executions after which $q(w) = 0$ is true for the first time. □

It remains to show that each while program can be transformed into an equivalent (i.e. computing the same function) while program that is normalized. Towards this goal we need two "tricks":

- Computable (or *effective*) encoding a sequence of natural numbers into a natural number (a trick called the *Gödel numbering*); and
- simplification of control structure via *program counters*.

7.1.1 The Gödel Numbering of Sequences

Our aim now is to encode

$$\text{a sequence } (x_0, \ldots, x_{m-1}) \in \mathbb{N}^m \text{ of natural numbers}$$

of an arbitrary length $m \in \mathbb{N}$, into

$$\text{a single natural number } n \in \mathbb{N}.$$

This is obviously possible if we forget about the computability (or effectivity) of the encoding, since we can construct a bijection

$$\mathbb{N}^* = \coprod_{m \in \mathbb{N}} \mathbb{N}^m \quad \cong \quad \mathbb{N}$$

(they have the same cardinality $\aleph_0$). The point here is that one can also do this effectively, that is, using PR functions.

7.1.4 Definition (The Gödel numbering of sequences)**.** Define $G : \mathbb{N}^* \to \mathbb{N}$ by:

$$G(x_0, \ldots, x_{m-1}) := \prod_{i \in [0, m-1]} \big(\operatorname{pr}(i)\big)^{x_i + 1} \ ,$$

where $\operatorname{pr} : \mathbb{N} \to \mathbb{N}$ is the PR function from Lem. 6.1.23.

For example, $G(1, 4, 0, 2) = 2^{1+1} \cdot 3^{4+1} \cdot 5^{0+1} \cdot 7^{2+1} = 1666980$. The number $G(x_0, \ldots, x_{m-1})$ is called the *Gödel number* of the sequence $x_0, \ldots, x_{m-1}$.

We have the following property. Note that there is no such notion like "a PR function $\mathbb{N}^* \to \mathbb{N}$"—the length of the input must be fixed.

7.1.5 Lemma. *For each m, the restriction $G_m : \mathbb{N}^m \to \mathbb{N}$ of the above G is primitive recursive.*

Proof. Obvious from Lem. 6.1.23. □

Here are the "inverses" of G.

7.1.6 Lemma. *There exist PR functions*

$$\begin{array}{rcll} |\_| & : & \mathbb{N} \longrightarrow \mathbb{N} & \text{and} \\ \boldsymbol{\lambda}(x, y).\ (x)_y & : & \mathbb{N}^2 \longrightarrow \mathbb{N} & \end{array}$$

such that

- $\big|G(x_0, \ldots, x_{m-1})\big| = m$, *and*
- $\big(G(x_0, \ldots, x_{m-1})\big)_i = x_i$ *for each* $i \in [0, m-1]$.

Proof. To calculate $|x|$, find the smallest i such that the remainder of $x \div \operatorname{pr}(i)$ is not 0. This can be done with a bounded least solution operator since i never exceeds x. The second is easy. □

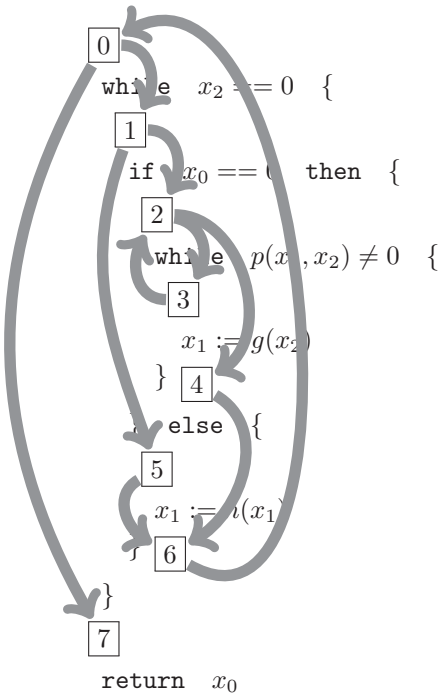
7.1.7 Notation. In what follows, $G(1, 4, 0, 2)$ will often be simply written as $\langle 1, 4, 0, 2 \rangle$.

7.1.2 Normalizing While Programs

We can transform any while program into an equivalent normalized program (Def. 7.1.2). The transformation is explained by an example. Consider the following program, where p, g, h are PR functions.

```
while  x_2 == 0  {
  if  x_0 == 0  {
    while  p(x_1, x_2) ≠ 0  {
      x_1 := g(x_2)
    }
  }  else  {
    x_1 := h(x_1)
  }
}
return  x_0
```
(7.2)

We first put a "marker" for each stage of execution of the program.



Then it is not hard to see that the following program is equivalent to the original one (7.2). Here pc stands for "program counter"; and the `cases` construct

is the obvious abbreviation of `if`...`then`...`else`....

```
while  pc ≠ 7  {
  cases  {
     pc == 0  &&  x_2 == 0  :      pc := 1;
     pc == 0  &&  x_2 ≠ 0   :      pc := 7;
     ...
     pc == 3    :                  ( x_1 := g(x_2);   pc := 2 );
     ...
  }
}
return  x_0
```
(7.3)

Now we "bundle up" the variables using the Gödel numbering (Def. 7.1.4, Lem. 7.1.6).

```
w := G(0, x_0, x_1, x_2)      (* The first 0 is for pc *)
while  (w)_0 ≠ 7  {
  cases  {
     (w)_0 == 0  &&  (w)_3 == 0  :      (w)_0 := 1;
     (w)_0 == 1  &&  (w)_3 ≠ 0   :      (w)_0 := 7;
     ...
     (w)_0 == 3    :                    ( (w)_1 := g((w)_3);   (w)_0 := 7 );
     ...
  }
}
return  (w)_1
```
(7.4)

This program is normalized; note in particular that the `cases`$\{\dots\}$ part is PR due to Lem. 6.1.18.

Using Lem. 7.1.3, we obtain:

7.1.8 Theorem. *The partial function computed by a while program is recursive.* □

Combined with Thm. 7.1.1, we derive:

7.1.9 Corollary. *A partial function is computed by a while program if and only if it is recursive.* □

7.1.3 Kleene's Normal Form Theorem

In the above proof we normalized while programs. This construction can be translated into recursive functions and we obtain the following result.

7.1.10 Theorem (Kleene's normal form)**.** *For any recursive function $f : \mathbb{N}^m \rightharpoonup \mathbb{N}$, there exist PR functions $q : \mathbb{N}^{m+1} \to \mathbb{N}$ and $k : \mathbb{N}^{m+1} \to \mathbb{N}$ such that*

$$f(\vec{x}) \quad = \quad k\big(\vec{x},\ \mu_y.\, q(\vec{x}, y) = 0\big)\ . \tag{7.5}$$

The theorem states that any recursive function can be defined using the minimization operator μ only once.

Proof. Given f, there is a while program that computes it (Thm. 7.1.1). The while program can be normalized; and the partial function computed by this normalized program is, by Lem. 7.1.3, of the form (7.5). □

7.2 Church's Thesis

In Cor. 7.1.9 we saw the coincidence of the power of two computing formalisms: while programs and recursive functions. In the early 20th century results were obtained that many other natural formalisms have exactly the same power, too—Turing machines, λ-calculus, etc. This led to the following "thesis":

> A partial function is *effectively calculable* if and only if it is a recursive function.

This is called *Church's thesis* (also called the *Church-Turing thesis*).

It is emphasized that there is no formal/mathematical definition of the word "effectively calculable" in the thesis. It is an informal notion—it is much like what we called "what machines can do." Therefore the truth of Church's thesis cannot be verified or falsified; in particular, it is (conceptually) impossible to prove Church's thesis! Nevertheless, Church's thesis is nearly universally accepted—that is, it is considered to be sensible to define "effective calculability" by recursive functions.

7.2.1 Remark. *Quantum computation* is a new paradigm of computation that employs quantum devices. It is commonly believed that it solves (at least) some computational problems much faster. For example, famous Shor's algorithm solves the integer factorization problem in polynomial time—whereas the best known classical algorithm is subexponential.[2]

While quantum computation is believed to have advantage in speed, it is also commonly believed that it is still bound by Church's thesis—i.e. what is computed by a quantum computer is a recursive function.

[2]It is however an open problem whether there is a classical polynomial time algorithm for the integer factorization problem. In fact, there is no known separation result between BQP—the most common definition of "quantum polynomial time"—and other related classical complexity classes like P, PSPACE or NP.

Chapter 8

Further on Recursive Functions/Predicates

8.1 Universal Recursive Function

An important feature of recursive functions—as well as other formalisms like Turing machines and while programs—is that *they can interpret themselves*. This is like in your implementation course, where you would write an interpreter of OCaml in OCaml. This fact also provides us with a useful vehicle for proving many theoretical results (later in this chapter); most notably we use it for proving that the halting problem is not computable—i.e. there is a function that is *not* recursive.

We first define a universal recursive function. Note that it is a totally different question whether a universal recursive function indeed exists. We will not present an existence proof, which is via a concrete construction.

8.1.1 Definition (Universal recursive function)**.** A partial function $\mathsf{comp} : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ is said to be a *universal recursive function* if 1) it is recursive, and 2) for any $m \in \mathbb{N}$ and any recursive function $f : \mathbb{N}^m \rightharpoonup \mathbb{N}$, there exists $p_f \in \mathbb{N}$ (called a *code* of f) such that

$$\boldsymbol{\lambda}(x_0, \ldots, x_{m-1}).\quad \mathsf{comp}\big(\, p_f, \langle x_0, \ldots, x_{m-1}\rangle \,\big) \qquad \doteq \qquad f \ .$$

Recall that $\langle x_0, \ldots, x_{m-1}\rangle = G(x_0, \ldots, x_{m-1}) \in \mathbb{N}$ is the Gödel number of the sequence of natural numbers $(x_0, \ldots, x_{m-1})$ (Def. 7.1.4, Notation 7.1.7).

Therefore comp is an *interpreter* that interprets a code p_f of a recursive function f. The function comp—while it is of a fixed arity 2—interprets a recursive function of an arbitrary arity m. This is possible since we feed it with the input $\langle x_0, \ldots, x_{m-1}\rangle$, i.e. the sequence $(x_0, \ldots, x_{m-1})$ encoded into a single natural number.

8.1.2 Theorem. *There is a universal recursive function* $\mathsf{comp} : \mathbb{N}^2 \rightharpoonup \mathbb{N}$.

Proof. One can explicitly write down the definition of comp. This concrete construction is much like implementing an interpreter of OCaml in OCaml—but here it is done in recursive functions instead of in OCaml. See e.g. [13, 11]. □

8.1.3 Remark. In the rest of the current notes, we fix one universal recursive function comp.

As an application, we can finally prove Lem. 6.2.8 on case distinction.

Proof. (Of Lem. 6.2.8, pp. 103) Let $p_i \in \mathbb{N}$ be a code of the recursive function g_i, for each $i \in [0, n-1]$. Then the function

$$c \quad := \quad \boldsymbol{\lambda}\overrightarrow{x}. \quad \Big[\, p_0 \cdot \big(1 \dot{-} \chi_{P_0}(\overrightarrow{x})\big) + \cdots + p_{n-1} \cdot \big(1 \dot{-} \chi_{P_{n-1}}(\overrightarrow{x})\big) \,\Big]$$

from $\mathbb{N}^m$ to $\mathbb{N}$ is obviously a (total) recursive function. Then it is easy to see that the function

$$\boldsymbol{\lambda}\overrightarrow{x}. \quad \mathsf{comp}\big(\, c(\overrightarrow{x}), \langle \overrightarrow{x} \rangle \,\big)$$

is equal to f in (6.10). □

8.1.4 Remark. Note that, given a recursive function $f : \mathbb{N}^m \rightharpoonup \mathbb{N}$, Def. 8.1.1 does not require its code p_f to be unique. That is, there can be two different natural numbers $p, p' \in \mathbb{N}$ such that

$$\boldsymbol{\lambda}\overrightarrow{x}.\, \mathsf{comp}(p, \langle \overrightarrow{x} \rangle) \quad \doteq \quad \boldsymbol{\lambda}\overrightarrow{x}.\, \mathsf{comp}(p', \langle \overrightarrow{x} \rangle) \quad \doteq \quad f \ .$$

Exercise 8.3 shows that this is indeed *necessarily* the case.

Here is a related remark.

8.1.5 Remark. Given any $p, m \in \mathbb{N}$, p is a code of some m-ary recursive function; namely

$$\boldsymbol{\lambda}(x_0, \ldots, x_{m-1}).\quad \mathsf{comp}(p, \langle x_0, \ldots, x_{m-1} \rangle) \quad : \quad \mathbb{N}^m \rightharpoonup \mathbb{N} \ .$$

8.2 The Halting Problem is Undecidable

A universal recursive function comp allows us to make *self-reference* in the world of recursive functions. As in the liar's paradox ("what I say is a lie"), self-reference with a negative twist is a source of paradoxes; this way we obtain a famous result, namely that the halting problem is undecidable.

First we present the precise statement.

8.2.1 Theorem. *Let* $\mathsf{halt} \subseteq \mathbb{N}^2$ *be a binary predicate defined by*

$$\mathsf{halt}(p, x) \textit{ is true} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \mathsf{comp}(p, x) \textit{ is defined.}$$

The predicate halt *is not recursive.*

The predicate $\mathsf{halt}(p, x)$ is true if and only if the recursive function represented by p is defined (i.e. its computation *halts*) for the input coded by x. Recall that a recursive predicate is also called a *decidable* predicate (Def. 6.2.6)—this is in the sense that there is an "effective method" to decide if the predicate holds or not (cf. §7.2). The theorem claims *undecidability* of the halting problem.

We are set out to prove Thm. 8.2.1. This is via the following key lemma, which is proved by a diagonal argument (i.e. "negative self-reference").

8.2.2 Lemma. *Let a total function* $\mathsf{comp}^+ : \mathbb{N}^2 \to \mathbb{N}$ *be defined by:*

$$\mathsf{comp}^+(p,x) = \begin{cases} y & \text{if } \mathsf{comp}(p,x) \text{ is defined and its value is } y \\ 0 & \text{if } \mathsf{comp}(p,x) \text{ is undefined} \end{cases}$$

This function comp^+ *is not recursive.*

The function comp^+ is comp with "padding" so that it becomes total.

Proof. Assume comp^+ is recursive. Let $\mathsf{diag} : \mathbb{N} \to \mathbb{N}$ defined by

$$\mathsf{diag}(x) := \mathsf{comp}^+(x, \langle x \rangle) + 1 \ ; \tag{8.1}$$

since comp^+ is total and recursive, diag is a total recursive function. Hence we can take a code p_0 of the function diag. For this we have

$$\mathsf{diag} \quad \doteq \quad \boldsymbol{\lambda} x.\, \mathsf{comp}(p_0, \langle x \rangle) \tag{8.2}$$

and, since diag is total, we have

$$\mathsf{diag}(x) \quad = \quad \mathsf{comp}(p_0, \langle x \rangle) \qquad \text{for each } x \in \mathbb{N}.$$

Then

$$\mathsf{diag}(p_0) \overset{(8.1)}{=} \mathsf{comp}^+(p_0, \langle p_0 \rangle) + 1 \overset{(8.2)}{=} \mathsf{diag}(p_0) + 1 \ .$$

This is a contradiction. □

We can indeed present the above proof in a diagonal manner. See Fig. 8.1.

Proof. (Of Thm. 8.2.1) The function comp^+ can be defined by

$$\mathsf{comp}^+(p,x) = \begin{cases} \mathsf{comp}(p,x) & \text{if } \mathsf{halt}(p,x) \text{ is true,} \\ 0 & \text{if } \mathsf{halt}(p,x) \text{ is not true.} \end{cases}$$

If halt is a recursive predicate, comp^+ is a recursive function by Lem. 6.2.8. This contradicts Lem. 8.2.2. □

An intuitive understanding of this undecidability result is: there is no algorithm that

- (as input) takes a program and its input values, and
- (as output) answers if the program terminates or not.

Other well-known undecidable predicates are: "p is a code of a total function"; and "p is a code of the given function f." (Both are unary predicate on $p \in \mathbb{N}$) Later we will prove a general result (Thm. 8.3.3) that shows undecidability of many predicates.

8.3 Recursion Theorem

The main topic of this section is *recursion theorem*. In it the word "recursion" has little to do with the phrase "recursive function"; it is the result that allows us to make a *recursive call* of programs (i.e. recursive functions).

First we need the following result, which is also famous. It is also called the *parameter theorem*.

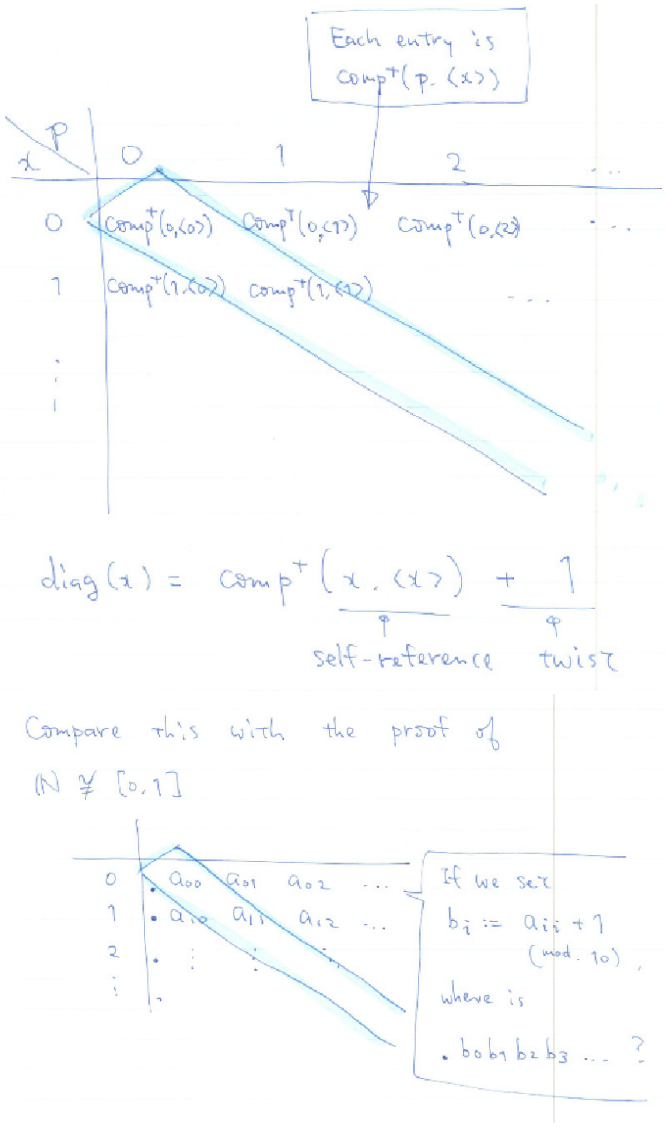


Figure 8.1: A diagonal argument for the halting problem (In the top figure, p and x must be swapped)

8.3.1 Theorem (s-m-n Theorem)**.** *Let $m, n \in \mathbb{N}$ be natural numbers. There is a primitive recursive function $\mathcal{S}_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ that satisfies the following: for each $p \in \mathbb{N}$, $\overrightarrow{x} \in \mathbb{N}^n$ and $\overrightarrow{y} \in \mathbb{N}^m$,*

$$\boldsymbol{\lambda}\overrightarrow{x}.\, \mathsf{comp}(\mathcal{S}_n^m(p, \overrightarrow{y}), \langle \overrightarrow{x} \rangle) \quad \doteq \quad \boldsymbol{\lambda}\overrightarrow{x}.\, \mathsf{comp}(p, \langle \overrightarrow{x}, \overrightarrow{y} \rangle).\ .$$

Proof. (Sketch) By direct manipulation of codes. Specifically, we consider the following manipulation.

- Taking $p \in \mathbb{N}$ and $\overrightarrow{y} \in \mathbb{N}^m$ as input,
- first we build up a while program p whose code is p,
- next fix the last part of the input of p to $\overrightarrow{y}$, and
- finally return a code of the resulting program.

This operation is the function $\mathcal{S}_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$. It is a syntactic manipulation for which we do not need minimization operations (or while loops); therefore we can construct $\mathcal{S}_n^m$ as a PR function. □

The name "s-m-n theorem" comes from the notation $\mathcal{S}_n^m$. The statement can be intuitively understood as the foundation of *partial evaluation*. This is used in the proof of the following remarkable result.

8.3.2 Theorem (Recursion theorem)**.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a total recursive function, and $k \in \mathbb{N}$ be an arbitrary natural number. Then there exists $r \in \mathbb{N}$ such that r and $f(r)$ represent the same k-ary recursive function. The latter means:*

$$\boldsymbol{\lambda}\overrightarrow{x}.\, \mathsf{comp}(r, \langle \overrightarrow{x} \rangle) \ \doteq\ \boldsymbol{\lambda}\overrightarrow{x}.\, \mathsf{comp}(f(r), \langle \overrightarrow{x} \rangle) \tag{8.3}$$

where $\doteq$ is the Kleene equality (Def. 6.2.4).

Proof. Consider a partial function g defined by

$$g(\overrightarrow{x}, v) := \mathsf{comp}(f(\mathcal{S}_k^1(v, v)),\ \langle \overrightarrow{x} \rangle). \tag{8.4}$$

Here v can be thought of as a code of a $(k+1)$-ary partial recursive function, whose last argument is the code of a "subroutine." The value $\mathcal{S}_k^1(v, v)$ is then a code of a k-ary function that makes a recursive call, calling itself as a subroutine.

This function g in (8.4) is recursive; take its code p. Then the number $\mathcal{S}_k^1(p, p)$ is what we want as r. Indeed,

$$\begin{aligned}
&\mathsf{comp}(\mathcal{S}_k^1(p, p), \langle \overrightarrow{x} \rangle) \\
&= \mathsf{comp}(p, \langle \overrightarrow{x}, p \rangle) && \text{by the property of } \mathcal{S}_k^1 \\
&= g(\overrightarrow{x}, p) && \text{since } p \text{ is a code of } g \\
&= \mathsf{comp}(f(\mathcal{S}_k^1(p, p)), \langle \overrightarrow{x} \rangle) && \text{by def. of } g.
\end{aligned}$$

□

Some intuitions. We think of $f : \mathbb{N} \to \mathbb{N}$ as a function that takes a code and returns a code. It is like a code of a recursive function with a subroutine left open: it takes a code of a subroutine s and returns the code of the whole program $f(s)$. The equality (8.3) hints that r is the recursive call $f(f(f(\dots)))$.

As an application we present the following result. It is a useful tool for proving undecidability of many predicates.

8.3.3 Theorem (Rice's theorem)**.** *Let k be an arbitrary natural number; and $g : \mathbb{N} \rightharpoonup \mathbb{N}$. If g satisfies the following conditions, then g is not recursive.*

1. *("Totality") For any $p \in \mathbb{N}$, the value $g(p)$ is defined and $g(p) \in \{0, 1\}$.*
2. *("No constant") g is not constant: there exists two natural numbers $p_0, p_1 \in \mathbb{N}$ such that $g(p_0) = 0$ and $g(p_1) = 1$.*
3. *("Reflects equivalence of codes") If p, q are codes for the same recursive function $\mathbb{N}^k \rightharpoonup \mathbb{N}$ (in Kleene's sense, Def. 6.2.4), then $g(p) = g(q)$.*

Proof. We use the recursion theorem (Thm. 8.3.2).

Assume that g is recursive. Define a total function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(x) = \begin{cases} p_1 & \text{if } g(x) = 0, \\ p_0 & \text{if } g(x) = 1. \end{cases}$$

Then f is a total recursive function by Lem. 6.2.8; note the totality of g. By the recursion theorem, there exists a code r of a k-input recursive function such that r and $f(r)$ code the same recursive function. Now:

- if $g(r) = 0$, then $f(r) = p_1$ so $g(f(r)) = 1$;
- if $g(r) = 1$, then $f(r) = p_0$ so $g(f(r)) = 0$.

Therefore $g(r) \neq g(f(r))$ in either case. This contradicts the condition 3. □

8.4 Recursively Enumerable Predicate

Now we go a bit further beyond what is recursive/computable/recursive, by introducing a class of predicates called *recursively enumerable (RE)*. These predicates are in general undecidable; however they play an important role later in incompleteness.

8.4.1 Definition (Recursively enumerable predicate)**.** A predicate $P \subseteq \mathbb{N}^m$ is said to be *recursively enumerable (RE)* if there exists a recursive predicate $Q \subseteq \mathbb{N}^{m+1}$ such that, for any $(x_0, \ldots, x_{m-1}) \in \mathbb{N}^m$,

$$P(x_0, \ldots, x_{m-1}) \text{ holds} \quad \Longleftrightarrow \quad Q(x_0, \ldots, x_{m-1}, y) \text{ holds for some } y \in \mathbb{N} \ .$$

To *enumerate* a collection is to list the elements of the collection one by one. The following characterization will help us building this intuition.

8.4.2 Theorem. *Let $P \subseteq \mathbb{N}^m$ be a predicate. The following are equivalent.*

1. *P is* semi-decidable*: there is a recursive function $f : \mathbb{N}^m \rightharpoonup \mathbb{N}$ such that*

$$f(\overrightarrow{x}) = \begin{cases} 0 & \text{if } P(\overrightarrow{x}) \text{ holds} \\ \text{undefined} & \text{if } P(\overrightarrow{x}) \text{ does not hold} \end{cases}$$

2. *P is recursively enumerable.*

3. *There is a recursive function* $g : \mathbb{N}^m \rightharpoonup \mathbb{N}$ *such that*

$$P = \mathrm{dom}(g) = \{ \overrightarrow{x} \in \mathbb{N}^m \mid \textit{the value } g(\overrightarrow{x}) \textit{ is defined} \} \ .$$

4. *there exists a PR predicate* $Q \subseteq \mathbb{N}^{m+1}$ *such that, for for any* $(x_0, \ldots, x_{m-1}) \in \mathbb{N}^m$,

$$P(x_0, \ldots, x_{m-1}) \textit{ holds} \qquad \Longleftrightarrow \qquad Q(x_0, \ldots, x_{m-1}, y) \textit{ holds for some } y \in \mathbb{N} \ .$$

Furthermore, when $m = 1$, *the above conditions are equivalent to:*

5. $P \subseteq \mathbb{N}$ *is either empty, or there is a PR function* $h : \mathbb{N} \to \mathbb{N}$ *such that*

$$P = \mathrm{image}(h) = \{ h(x) \mid x \in \mathbb{N} \} \ .$$

In Cond. 1., the function f might look like a characteristic function χ_P but it is not, because a characteristic function must be total (Def. 1.2.13; see also Def. 6.2.6). Cond. 4. says that, in Def. 8.4.1, we could in fact restrict to PR predicates.

Proof. We present the proof only for the case of $m = 1$.

[1.⇒ 3.] Take f in Cond. 1. as g in Cond. 3.

[3.⇒ 5.] By Kleene's normal form theorem (Thm. 7.1.10), the recursive function g in Cond. 3 can be written as

$$g(x) \quad = \quad i\Big(\, x, \ \mu_y.\big(\, j(x,y) = 0 \, \big) \, \Big)$$

using PR functions i, j. Assume P is nonempty; choose $a \in P$ (a "dummy value"). Using the Gödel numbering of sequences (Def. 7.1.4, Notation 7.1.7), we can enumerate all x such that there exists y with $j(x, y) = 0$:

$$h(x) := \begin{cases} (x)_1 & \text{if } |x| = 2 \text{ and } j\big(\, (x)_1, (x)_2 \, \big) = 0; \\ a & \text{otherwise} \end{cases}$$

Then, since j, $|\_|$, $(\_)_i$ are all PR, h is PR. Moreover we have $\mathrm{image}(h) = P$.

[5.⇒ 4.] If P is empty, then $Q = \emptyset$ obviously satisfies Cond. 4. Otherwise, define Q by

$$Q(x, y) \text{ holds} \qquad \Longleftrightarrow \qquad x = h(y) \ .$$

Then

$$\begin{aligned} & x \in \mathrm{image}(h) && (8.5) \\ \Longleftrightarrow \quad & x = h(y) \quad \text{for some } y \in \mathbb{N} && (8.6) \\ \Longleftrightarrow \quad & Q(x, y) \quad \text{holds for some } y \in \mathbb{N}. && (8.7) \end{aligned}$$

[4.⇒ 2.] Obvious (if a predicate is PR then it is recursive)

[2.⇒ 1.] Let $Q \subseteq \mathbb{N}^2$ be a recursive predicate such that

$$P(x) \text{ holds} \qquad \Longleftrightarrow \qquad Q(x, y) \text{ holds for some } y \in \mathbb{N} \ .$$

Its characteristic function $\chi_Q : \mathbb{N}^2 \to \mathbb{N}$ is a total recursive function (Def. 6.2.6); use it in the following definition of f in Cond. 1.:

$$f(x) \ := \ 0 \cdot \big(\, \mu_y. \, \chi_Q(x, y) = 0 \, \big) \ . \qquad \square$$

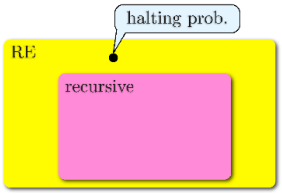
By Cond. 5., we can use some PR function h to enumerate the elements of P:

$$P \;=\; \{\, h(0), h(1), h(2), \dots \,\} \;.$$

Thm. 8.4.2 leads to the following informal description of the difference between recursive predicates and RE ones.

- If a predicate $P \subseteq \mathbb{N}$ is recursive: there is a "machine" that tells, for the problem of whether a given $x \in \mathbb{N}$ belongs to P, "yes" or "no" after some finite time.
- If a predicate $P \subseteq \mathbb{N}$ is RE: there is a "machine" that tells "yes" (after some finite time) if a given $x \in \mathbb{N}$ is in P. If x is not in P the machine might not say anything. In this case we can only know that x is not in P after infinitely long time!

It is straightforward to see that the class of recursive predicates is included in that of RE predicates (Exercise 8.4). The following proposition *separates* the two classes—i.e. the inclusion is proper—by showing that the halting problem is RE.



(8.8)

8.4.3 Proposition. *The predicate* $\mathsf{halt} \subseteq \mathbb{N}^2$ *is RE.*

Proof. Obvious from the definition of halt (Thm. 8.2.1), Cond. 3. of Thm. 8.4.2, and the fact that comp is recursive. □

Note that when P is RE, it negation $\mathbb{N}^m \setminus P$ is not necessarily RE. In fact we have the following important result; it "attacks from both sides."

8.4.4 Theorem (Negation theorem)**.** *Let $P \subseteq \mathbb{N}^m$ be a predicate. The following are equivalent.*

1. *P is recursive.*
2. *P and $\mathbb{N}^m \setminus P$ are both RE.*

Proof. [1 to 2] The negation of a recursive predicate is obviously recursive too; and a recursive predicate is RE.

[2 to 1] We have recursive predicates $Q, R \subseteq \mathbb{N}^{m+1}$ such that, for any $\overrightarrow{x} \in \mathbb{N}^m$,

$$P(\overrightarrow{x}) \text{ holds} \iff Q(\overrightarrow{x}, y) \text{ holds for some } y \in \mathbb{N} \;; \tag{8.9}$$

$$P(\overrightarrow{x}) \text{ does not hold} \iff R(\overrightarrow{x}, y) \text{ holds for some } y \in \mathbb{N} \;. \tag{8.10}$$

Now the predicate

$$\boldsymbol{\lambda}(\overrightarrow{x}, y).\quad \big(\, Q(\overrightarrow{x}, y) \vee R(\overrightarrow{x}, y) \,\big)$$

is again recursive (Lem. 6.2.7); thus the partial function $g : \mathbb{N}^m \rightharpoonup \mathbb{N}$, defined by

$$\begin{aligned} g(\overrightarrow{x}) &:= \mu_y.\ \ Q(\overrightarrow{x}, y) \vee R(\overrightarrow{x}, y) \\ &= \mu_y.\ \ \chi_{Q \vee R}\,(\overrightarrow{x}, y) = 0 \end{aligned}$$

is a recursive function. A crucial fact here is that g is total: for each $\overrightarrow{x} \in \mathbb{N}^m$, the predicate P either holds or does not hold; thus exactly one of (8.9–8.10) is true.

We use this function in the following predicate $S \subseteq \mathbb{N}^m$:

$$S(\overrightarrow{x}) \text{ holds} \qquad \overset{\text{def}}{\Longleftrightarrow} \qquad \boldsymbol{\lambda}\overrightarrow{x}.\ \ Q\big(\overrightarrow{x}, g(\overrightarrow{x})\big) \text{ holds} \ ; \tag{8.11}$$

our goal is to show that this predicate S is recursive, and that it coincides with P.

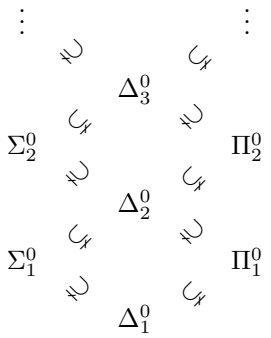
It is clear that S is recursive, since both Q and g are recursive. To see that $S = P$, let $\overrightarrow{x} \in \mathbb{N}^m$ be any tuple.

- Assume $P(\overrightarrow{x})$ holds. Then by (8.9–8.10),
  - there exists $y \in \mathbb{N}$ such that $Q(\overrightarrow{x}, y)$ holds, but
  - there is no $y \in \mathbb{N}$ such that $R(\overrightarrow{x}, y)$ holds.

  Thus the value $g(\overrightarrow{x})$ is such that $Q\big(\overrightarrow{x}, g(\overrightarrow{x})\big)$ is true; this means $S(\overrightarrow{x})$ is true.
- Assume $P(\overrightarrow{x})$ does not hold. Then by (8.9), $Q(\overrightarrow{x}, y)$ is false for any $y \in \mathbb{N}$. Thus in particular $Q\big(\overrightarrow{x}, g(\overrightarrow{x})\big)$ is false; hence $S(\overrightarrow{x})$ is false.

This concludes the proof. □

8.4.5 Remark. What we have seen now is the first two layers of the *arithmetical hierarchy*: above recursive predicates (the class Δ_1^0) and RE predicates (the class Σ_1^0), we further have the following ladder of classes of predicates. The upper, the more complex.



This is how one starts *recursion theory*; it is concerned with the "complexity classes" above recursive predicates. Note that the "complexity classes" that we hear about more often—like P, NP, EXPTIME, PSPACE, etc.—are all recursive, thus below Δ_1^0.

We have observed the separation between RE and recursive predicates (see (8.8)). The separation between PR and recursive predicates holds too. The following (technical) proof is an exercise of the diagonal method. You can skip it at your first read.

8.4.6 Proposition. *There is a recursive predicate that is not primitive recursive.*

Proof. We shall construct a predicate $Q \subseteq \mathbb{N}$ that is recursive but is not PR. The proof uses the diagonal method.

Let $\mathsf{PRP} \subseteq \mathbb{N}$ be a unary predicate defined by

$$\mathsf{PRP} \;:=\; \left\{ p \;\middle|\; \begin{array}{l} p \text{ is the encoding of the definition of the characteristic} \\ \text{function } \chi_P \text{ of some unary PR predicate } P \end{array} \right\} .$$

Then we can show that $\mathsf{PRP} \subseteq \mathbb{N}$ is a recursive predicate. Its proof relies on the concrete construction of the universal recursive function comp.

(Note here that PRP is a proper subset of the set

$$\mathsf{PRP}' \;:=\; \left\{ p \;\middle|\; \begin{array}{l} p \text{ is a code of the characteristic function } \chi_P \\ \text{of some unary PR predicate } P \end{array} \right\} .$$

PRP' is bigger than PRP because it is possible that a function defined using minimization happens to be PR. The set PRP' is more complex than PRP: for example PRP' is not recursive by Rice's theorem.)

We can "enumerate" the recursive predicate $\mathsf{PRP} \subseteq \mathbb{N}$, that is, there exists a recursive function $\mathsf{enumPRP} : \mathbb{N} \to \mathbb{N}$ such that

$$\mathsf{PRP} = \text{image}(\mathsf{enumPRP}) \ .$$

Indeed, such a function can be explicitly given using the minimization operator μ; or alternatively, one can use Thm. 8.4.2, Cond. 5. (PRP is recursive hence is RE).

Now consider the following function $f : \mathbb{N}^2 \to \mathbb{N}$.

$$f(x, y) \;:=\; \mathsf{comp}\big(\, \mathsf{enumPRP}(x), y \,\big)$$

Obviously f is recursive. Moreover f is total: for each x we have $\mathsf{enumPRP}(x) \in \text{image}(\mathsf{enumPRP}) = \mathsf{PRP}$, hence $\mathsf{enumPRP}(x)$ is a code of a total function χ_P for some PR predicate P. Therefore the predicate $Q \subseteq \mathbb{N}$, defined by

$$Q(x) \text{ holds} \qquad \overset{\text{def}}{\iff} \qquad f(x, x) = 0 \tag{8.12}$$

is a recursive predicate.

We claim by contradiction that Q is not PR. Assume Q is PR. Then so is $\neg Q$; therefore there exists $x_0 \in \mathbb{N}$ such that $\mathsf{enumPRP}(x_0)$ is a code of $\chi_{\neg Q}$. Now

$$\begin{array}{lll} (\neg Q)(x_0) \text{ holds} & & \\ \iff & \mathsf{comp}\big(\, \mathsf{enumPRP}(x_0), x_0 \,\big) = 0 & \text{since } \mathsf{enumPRP}(x_0) \text{ is a code of } \chi_{\neg Q} \\ \iff & Q(x_0) \text{ holds} & \text{by (8.12),} \end{array}$$

which is a contradiction. □

Exercises

8.1. Use Thm. 8.3.3 to prove that the predicate $\mathsf{total}_1 \subseteq \mathbb{N}$, defined by

$$\mathsf{total}_1(p) \quad \text{is true} \quad \overset{\text{def}}{\iff} \quad \boldsymbol{\lambda} x.\, \mathsf{comp}(p, \langle x \rangle) \text{ is a total recursive function}$$

is undecidable.

8.2. Use Thm. 8.3.3 to prove that the predicate $\mathsf{equal}_1 \subseteq \mathbb{N}^2$, defined by

$$\mathsf{equal}_1(p, q) \quad \text{is true} \quad \overset{\text{def}}{\iff} \quad \boldsymbol{\lambda} x.\, \mathsf{comp}(p, \langle x \rangle) \doteq \boldsymbol{\lambda} x.\, \mathsf{comp}(q, \langle x \rangle)$$

is not recursive.

Hint: fix q and consider a unary function $\boldsymbol{\lambda} p.\, \mathsf{equal}_1(p, q)$.

8.3. Use Exercise 8.2 to prove the following: there necessarily exist two different natural numbers p, q that are codes of the same unary recursive function, that is,

$$\boldsymbol{\lambda} x.\, \mathsf{comp}(p, \langle x \rangle) \doteq \boldsymbol{\lambda} x.\, \mathsf{comp}(q, \langle x \rangle) \qquad \text{and} \qquad p \neq q \ .$$

8.4. Show that every recursive predicate is recursively enumerable.

8.5. Let $P \subseteq \mathbb{N}^m$ be a recursive predicate, and $f_0, \ldots, f_{m-1} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be recursive functions. Prove that the predicate

$$\boldsymbol{\lambda} \overrightarrow{x}. \quad P\big(\, f_0(\overrightarrow{x}), \ldots, f_{m-1}(\overrightarrow{x}) \,\big) \ ,$$

defined by

$$P\big(\, f_0(\overrightarrow{x}), \ldots, f_{m-1}(\overrightarrow{x}) \,\big) \text{ holds}$$
$$\overset{\text{def}}{\iff} \quad \text{the values } f_0(\overrightarrow{x}), \ldots, f_{m-1}(\overrightarrow{x}) \text{ are all defined and } P\big(\, f_0(\overrightarrow{x}), \ldots, f_{m-1}(\overrightarrow{x}) \,\big) \text{ is true,} \tag{8.13}$$

is *not necessarily* recursive. (Note that f_i is not necessarily total. Hint: comp is recursive but halt is not recursive)

8.6. Let $P \subseteq \mathbb{N}^m$ be a recursive predicate, and $f_0, \ldots, f_{m-1} : \mathbb{N}^n \to \mathbb{N}$ be *total* recursive functions. Prove that the predicate

$$\boldsymbol{\lambda} \overrightarrow{x}. \quad P\big(\, f_0(\overrightarrow{x}), \ldots, f_{m-1}(\overrightarrow{x}) \,\big) \ ,$$

defined by defined by (8.13), is recursive.

8.7. Let $P \subseteq \mathbb{N}^m$ be an RE predicate, and $f_0, \ldots, f_{m-1} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be recursive functions. Prove that the predicate

$$\boldsymbol{\lambda} \overrightarrow{x}. \quad P\big(\, f_0(\overrightarrow{x}), \ldots, f_{m-1}(\overrightarrow{x}) \,\big) \ ,$$

defined by (8.13), is RE. (Hint: rather hard if you use Def. 8.4.1 directly. Choose a convenient equivalent condition from Thm 8.4.2.)

8.8. (From [13, §6.5]) We are interested in the condition

– $f(\overrightarrow{x}, 0), f(\overrightarrow{x}, 1), \ldots, f(\overrightarrow{x}, z)$ are all defined

in the definition of minimization (Def. 6.2.1). Its intuition was explained right after Def. 6.2.1; here we observe that, by dropping the condition, we actually get more than recursive functions.

1. Show that the following function $g : \mathbb{N}^3 \rightharpoonup \mathbb{N}$ is recursive.

$$g(p,x,y) = \begin{cases} 0 & \text{if } y > 0, \text{ or } \mathsf{comp}(p,x) \text{ is defined} \\ \text{undefined} & \text{if } y = 0, \text{ and } \mathsf{comp}(p,x) \text{ is undefined} \end{cases}$$

2. We define a partial function $f : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ with a relaxed minimization operator μ':

$$f(p,x) \;:=\; \mu'_y.\,\big(g(p,x,y) = 0\big) \;,$$

where the right-hand side denotes the smallest $y \in \mathbb{N}$ such that $g(p,x,y) = 0$ holds (we do not require that $g(p,x,0), g(p,x,1), \ldots, g(p,x,y-1)$ are all defined).

Show that this partial function f is not recursive.

Chapter 9

Gödel's Incompleteness Theorem

9.1 Theory in Predicate Logic

We first extend the predicate logic part of what is in Chap. 5. This will be needed in later sections.

9.1.1 Definition (Validity under a theory)**.** Let Φ be a theory. We say a formula A is *valid under* Φ and write $\Phi \models A$, if for any model $\mathbb{S}$ of Φ we have

$$\mathbb{S} \models A \ .$$

We will be using the following fact.

9.1.2 Lemma. *For a closed formula A, a structure $\mathbb{S}$ and any valuations J, J' over $\mathbb{S}$, we have*

$$[\![A]\!]_{\mathbb{S},J} = [\![A]\!]_{\mathbb{S},J'} \ . \tag{9.1}$$

It follows that

$$\mathbb{S} \models A \quad \textit{or} \quad \mathbb{S} \models \neg A \ .$$

Proof. The first part is immediate from Lem. 2.5.5. For the second part: assume $\mathbb{S} \not\models A$. Then by Def. 4.3.4, there is a valuation J such that $[\![A]\!]_{\mathbb{S},J} = \mathrm{ff}$. Therefore we have $[\![\neg A]\!]_{\mathbb{S},J} = \mathrm{tt}$; moreover by (9.1) we have $[\![\neg A]\!]_{\mathbb{S},J'} = \mathrm{tt}$ for any J'. Therefore $\mathbb{S} \models \neg A$. □

The following is a consequence of the strong completeness result, Thm. 5.2.8.

9.1.3 Theorem (Strong completeness, revisited)**.** *Let Φ be a theory in predicate logic; assume that Φ consists only of closed formulas. Then for any formula A, we have*

$$\Phi \vdash A \quad \Longleftrightarrow \quad \Phi \models A \ .$$

Recall that $\Phi \vdash A$ means A is derivable using the axioms from Φ (Def. 5.2.2); $\Phi \models A$ is from Def. 9.1.1. The condition that any $B \in \Phi$ must be closed is not a real burden—when we have a formula (i.e. an axiom) $B' \in \Phi$ that is not closed, it is most of the time its universal closure $\forall \overrightarrow{x}.B'$ that is really intended.

Proof. We show that the following are all equivalent.

1. $\Phi \vdash A$
2. $\Phi \cup \{\neg A\}$ is inconsistent
3. $\Phi \cup \{\neg A\}$ is unsatisfiable
4. $\Phi \models A$

[1.⇒ 2.] Let Π be a proof of A under Φ. Then the following proof is under $\Phi \cup \{\neg A\}$.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ \Pi \\ \Rightarrow A\end{array} \quad \dfrac{}{\Rightarrow \neg A}\ (\textsc{Axiom})}{\Rightarrow A \wedge \neg A}\ (\wedge\text{-R}) \qquad \dfrac{\dfrac{}{A \Rightarrow A}\ (\textsc{Init})}{A, \neg A \Rightarrow}\ (\neg\text{-L})}{\dfrac{}{A \wedge \neg A \Rightarrow}\ (\wedge\text{-L})}\ (\textsc{Cut})$$

[2.⇒ 1.] Let Π' be a proof of $\Rightarrow$ under $\Phi \cup \{\neg A\}$. We apply the following operations to Π' and obtain Π''.

- We first add a formula A to the right-hand side of all the sequents in Π'.
- At each leaf of Π' where the (Axiom) rule is used to derive $\Rightarrow \neg A$, i.e.

$$\dfrac{}{\Rightarrow \neg A}\ (\textsc{Axiom})\ ,$$

we now have

$$\dfrac{}{\Rightarrow \neg A, A}\ (\textsc{Axiom})\ .$$

We replace this with

$$\dfrac{\dfrac{}{A \Rightarrow A}\ (\textsc{Init})}{\Rightarrow \neg A, A}\ (\neg\text{-R})\ .$$

The resulting Π'' is a proof under Φ and its root is the sequent $\Rightarrow A$.

[2.⇔ 3.] By Thm. 5.2.8.

[3.⇒ 4.] Assume $\mathbb{S}$ is a model of Φ (Def. 5.3.4). We argue by contradiction: assume that there is a valuation J over $\mathbb{S}$ such that $[\![A]\!]_{\mathbb{S},J} = \mathrm{ff}$. Then $\mathbb{S}$ and J make all the formulas in $\Phi \cup \{\neg A\}$ true; this contradicts Cond. 3.

[4.⇒ 3.] By contradiction. Assume $\mathbb{S}$ and J make all the formulas in $\Phi \cup \{\neg A\}$ true, that is,

$$\begin{aligned} &[\![B]\!]_{\mathbb{S},J} = \mathrm{tt} \qquad \text{for each } B \in \Phi; \\ &[\![A]\!]_{\mathbb{S},J} = \mathrm{ff}\ . \end{aligned}$$

Since each $B \in \Phi$ is a closed formula, we have

$$[\![B]\!]_{\mathbb{S},J} = \mathrm{tt} \qquad \text{for each } J.$$

This means that $\mathbb{S}$ is a model of Φ. At the same time we have $\mathbb{S} \not\models A$ and this contradicts Cond. 4. □

9.2 Introduction to Incompleteness

We have seen, in Thm. 4.4.1 & 4.4.3, that predicate LK is sound and complete. This means: derivable formulas are exactly those formulas that are valid under any structure $\mathbb{S}$.

$$\vdash A \quad \underset{\text{completeness}}{\overset{\text{soundness}}{\rightleftharpoons}} \quad \models A \quad \overset{\text{by def.}}{\Longleftrightarrow} \quad \text{for any } \mathbb{S},\ \mathbb{S} \models A \tag{9.2}$$

In Thm. 9.1.3 we observed a stronger result:

$$\Phi \vdash A \quad \underset{\text{strong completeness}}{\overset{\text{soundness}}{\rightleftharpoons}} \quad \Phi \models A \quad \overset{\text{by def.}}{\Longleftrightarrow} \quad \text{for any } \mathbb{S} \in \mathrm{Mod}(\Phi),\ \mathbb{S} \models A \tag{9.3}$$

Now let's say we want to do "usual mathematics" within this formal framework of predicate logic. Let us first focus on a minimal fragment of "mathematics," i.e. *arithmetic* over natural numbers. Therefore we choose the following symbols:

$$\mathbf{FnSymb}_{\mathsf{a}} = \{0, \mathsf{s}, +, \cdot\ \} \ , \qquad \mathbf{PdSymb}_{\mathsf{a}} = \{=, <\} \ , \tag{9.4}$$

where each symbol comes with a suitable arity.

In this case, however, the scheme (9.3) would not be the most interesting one. We have a clear idea of what a model should be—namely the set $\mathbb{N}$ of natural numbers, on which the function/predicate symbols are interpreted in a natural way—and the other (strange, "nonstandard") models are of less interest. That is, we are more interested in the following scheme.

$$\Phi \vdash A \quad \overset{??}{\Longleftrightarrow} \quad \mathbb{N} \models A \tag{9.5}$$

Now the question is to find a suitable set Φ of axioms (i.e. a theory, Def. 5.2.1) such that (9.5) holds. Unfortunately, what Gödel's incompleteness theorem states is that this is impossible—however cleverly we choose Φ—as long as Φ is a "reasonable" set of axioms.

In fact, if any set of formulas is allowed as Φ, then the choice

$$\Phi_{\mathsf{ArithTruth}} \ = \ \{A \mid \mathbb{N} \models A\} \tag{9.6}$$

makes (9.5) hold. However this is cheating: given a formula A, how can we know if A belongs to $\Phi_{\mathsf{ArithTruth}}$ or not? It is very much questionable if the derivation system using $\Phi_{\mathsf{ArithTruth}}$ as an axiom set can be possibly called a "syntactic machinery."

It is at this very point that we use the theory of computability that we learned in the earlier chapters. Namely, a reasonable theory Φ must be such that, given a formula A, whether A belongs to Φ is "effectively decidable," that is, *recursive*. Such a theory is formally called a *recursively axiomatized theory*.

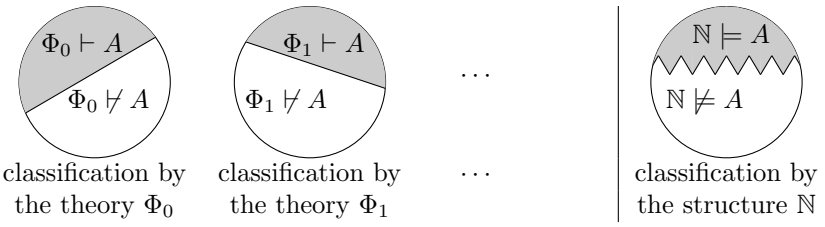


Figure 9.1: Classifications of different granularities

Then Gödel's incompleteness theorem can be understood in the following way. The way recursively axiomatized theories classify closed formulas,[1] i.e.

$$\begin{aligned}&\{\text{closed formulas}\} = \\ &\quad \{A \mid A \text{ is closed and } \Phi \vdash A\} \amalg \{A \mid A \text{ is closed and } \Phi \nvdash A\}\end{aligned} \tag{9.7}$$

is *necessarily simpler* than the *arithmetic truth*:

$$\begin{aligned}&\{\text{closed formulas}\} = \\ &\quad \{A \mid A \text{ is closed and } \mathbb{N} \models A\} \quad \amalg \quad \{A \mid A \text{ is closed and } \mathbb{N} \not\models A\} \ .\end{aligned}$$

See Fig. 9.1.

9.2.1 Remark. The above story differs from the common (and historical) path the incompleteness results are introduced. There completeness is a purely syntactic property (like in Def. 9.3.1); and Gödel's incompleteness theorem states that

> a complete, recursively axiomatized, consistent extension of Peano's arithmetic is impossible.

Notice that all these properties (like consistency in Def. 5.2.4) are syntactic; in contrast our version of incompleteness (Thm. 9.4.4) involves a model $\mathbb{N}$, a semantical and infinitary entity.

In case you are skeptical about the existence or legitimacy of a monster like $\mathbb{N}$, the statement of Thm. 9.4.4 is of no use. Still we made our choice because this way a great deal of technical subtleties (like the notion of ω-*inconsistency*) can be saved. An interested reader is referred e.g. to [12, 6].

9.3 Complexity of Theories

Note that the notion of "completeness" that we have talked about in Chap. 2–5 is concerned with a derivation system (like: "LK is complete"). Although we use the same word, the following notion is something totally different.

9.3.1 Definition (Complete theory)**.** A theory Φ is said to be *complete* if, for any closed formula A, we have exactly one of

$$\Phi \vdash A \qquad \text{or} \qquad \Phi \vdash \neg A \ .$$

[1] The reason we restrict our attention to *closed* formulas is Lem. 9.1.2.

In particular, a complete theory is consistent (Def. 5.2.4. See also Exercise 9.1).

In the definition of complete theory we are only concerned with *closed* formulas. This is because of Lem. 9.1.2.

A single structure determines a complete theory.

9.3.2 Lemma. *Let $\mathbb{S}$ be an arbitrary structure. The theory*

$$\Phi_{\mathbb{S}} \;:=\; \{A \mid \; \mathbb{S} \models A\}$$

is complete.

Proof. Let A be an arbitrary closed formula. If $\mathbb{S} \models A$ then obviously $\Phi_{\mathbb{S}} \vdash A$ (using the (Axiom) rule).

Otherwise, by Lem. 9.1.2, we have $\mathbb{S} \models \neg A$ therefore $\neg A \in \Phi_{\mathbb{S}}$. Therefore we have $\Phi_{\mathbb{S}} \vdash \neg A$ (again using the (Axiom) rule). □

To formally define the notion of recursively axiomatized theory, we need to encode formulas into natural numbers. Recall that $\langle x_0, \dots, x_{m-1}\rangle = G(x_0, \dots, x_{m-1}) \in \mathbb{N}$ is the Gödel number of the sequence of natural numbers $(x_0, \dots, x_{m-1})$ (Def. 7.1.4, Notation 7.1.7).

9.3.3 Definition (The Gödel numbering of predicate logic)**.** We fix an encoding $\ulcorner \_ \urcorner$ of all terms and formulas in predicate logic, into natural numbers. The precise definition of $\ulcorner \_ \urcorner$ does not matter; it can be defined in the following way, for example.

$\ulcorner x_0 \urcorner := \langle 0, 0\rangle$, $\ulcorner x_1 \urcorner := \langle 0, 1\rangle$, $\ulcorner x_2 \urcorner := \langle 0, 2\rangle$, $\dots$
$\ulcorner 0 \urcorner := \langle 1, 0\rangle$, $\ulcorner \mathsf{s}(t) \urcorner := \langle 2, \ulcorner t \urcorner\rangle$, $\ulcorner s + t \urcorner := \langle 3, \ulcorner s \urcorner, \ulcorner t \urcorner\rangle$, $\ulcorner s \cdot t \urcorner := \langle 4, \ulcorner s \urcorner, \ulcorner t \urcorner\rangle$
$\ulcorner t = u \urcorner := \langle 5, \ulcorner t \urcorner, \ulcorner u \urcorner\rangle$, $\ulcorner t < u \urcorner := \langle 6, \ulcorner t \urcorner, \ulcorner u \urcorner\rangle$
$\ulcorner A \wedge B \urcorner := \langle 7, \ulcorner A \urcorner, \ulcorner B \urcorner\rangle$, $\ulcorner A \vee B \urcorner := \langle 8, \ulcorner A \urcorner, \ulcorner B \urcorner\rangle$, $\cdots$
$\ulcorner \forall x_i.\, A \urcorner := \langle 11, i, \ulcorner A \urcorner\rangle$, $\ulcorner \exists x_i.\, A \urcorner := \langle 12, i, \ulcorner A \urcorner\rangle$

Here $\mathbf{Var} = \{x_0, x_1, \dots\}$ is an enumeration of variables.

The natural number $\ulcorner A \urcorner$ is called the *Gödel number* of the formula A.

It is obvious that the concrete choice of Gödel numbers in the above determines an injective mapping

$$\ulcorner \_ \urcorner \;:\; \mathbf{Terms} \amalg \mathbf{Fml} \quad \hookrightarrow \quad \mathbb{N} \;;$$

that is, different terms/formulas never get mapped to the same number. We use this fact, as well as the following fact, in what follows.

9.3.4 Lemma. *1. The predicate $\mathsf{fml} \subseteq \mathbb{N}$, defined by*

$$\mathsf{fml}(x) \textit{ holds} \qquad \overset{\text{def}}{\iff} \qquad x = \ulcorner A \urcorner \textit{ for some formula } A,$$

is recursive.

2. *The total functions*

$$\mathsf{neg}(x) := \begin{cases} \ulcorner \neg A \urcorner & \text{if } x = \ulcorner A \urcorner \\ 0 & \text{if } \mathsf{fml}(x) \text{ does not hold,} \end{cases}$$

$$\mathsf{univClosure}(x) := \begin{cases} \ulcorner \forall \overrightarrow{x}\,.\, A \urcorner & \text{if } x = \ulcorner A \urcorner \\ 0 & \text{if } \mathsf{fml}(x) \text{ does not hold,} \end{cases}$$

$$\mathsf{subst}(x,y,z) := \begin{cases} \ulcorner A[t/u] \urcorner & \text{if } x = \ulcorner A \urcorner,\ y = \ulcorner t \urcorner \text{ and } z = \ulcorner u \urcorner \\ & \text{for some } A \in \mathbf{Fml}, t \in \mathbf{Terms} \text{ and } u \in \mathbf{Var}; \\ 0 & \text{otherwise,} \end{cases}$$

are all recursive. Here $\forall \overrightarrow{x}\,.\, A$ *is the universal closure of* A *(Exercise 5.14).*

3. *The number* $0 \in \mathbb{N}$ *is not the Gödel number of any formula, i.e.*

$$\ulcorner A \urcorner \neq 0 \qquad \text{for any } A \in \mathbf{Fml}.$$

□

That is: it is effectively calculable whether a given natural number x is a Gödel number of some formula or not.

9.3.5 Definition (Recursively axiomatized theory)**.** A theory Φ is said to be *recursively axiomatized* if the predicate $\mathsf{axiom}_\Phi \subseteq \mathbb{N}$, defined by

$$\mathsf{axiom}_\Phi(x) \text{ holds} \quad \overset{\text{def}}{\iff} \quad x = \ulcorner A \urcorner \text{ for some } A \in \Phi, \tag{9.8}$$

is recursive.

Note that, when $\mathsf{axiom}_\Phi(x)$ does not hold, it is either:

– x is not a Gödel number of a formula, or

– $x = \ulcorner B \urcorner$ with $B \notin \Phi$.

9.3.6 Definition (Deductive closure)**.** Let Φ be a theory. Its *deductive closure* $\overline{\Phi}$ is the theory

$$\overline{\Phi} := \{A \mid \Phi \vdash A\}\ ,$$

where $\Phi \vdash A$ means derivability under the axioms from Φ (Def. 5.2.2).

Therefore $\overline{\Phi}$ is the set of formulas that are derivable from the axioms in Φ.

The following result is fundamental, setting an upper bound to the "complexity" of deductive systems.

9.3.7 Theorem. *Let* Φ *be a recursively axiomatized theory. Then the predicate* thm_Φ*, defined by*

$$\mathsf{thm}_\Phi(x) \textit{ holds} \quad \overset{\text{def}}{\iff} \quad x = \ulcorner A \urcorner \textit{ for some formula } A \textit{ and } \Phi \vdash A, \tag{9.9}$$

is recursively enumerable (RE).

Proof. (Sketch)

– We assign Gödel numbers also to proofs. This can be done in a straightforward manner: for example, we can define the Gödel number of the proof

$$\Pi \quad = \quad \left(\dfrac{\begin{array}{cc} \vdots\ \Pi' & \vdots\ \Pi'' \\ \Gamma \Rightarrow \Delta, C & \Gamma \Rightarrow \Delta, D \end{array}}{\Gamma \Rightarrow \Delta, C \wedge D}\ (\wedge\text{-R}) \right)$$

(where $\Gamma \equiv A_0, \ldots, A_{m-1}$ and $\Delta \equiv B_0, \ldots, B_{n-1}$) to be

$$\ulcorner \Pi \urcorner := \langle 2, \langle \ulcorner A_0 \urcorner, \ldots, \ulcorner A_{m-1} \urcorner \rangle, \langle \ulcorner B_0 \urcorner, \ldots, \ulcorner B_{n-1} \urcorner, \ulcorner C \wedge D \urcorner \rangle, \ulcorner \Pi' \urcorner, \ulcorner \Pi'' \urcorner \rangle \ .$$

Here the first component "2" is the label for the $(\wedge\text{-R})$ rule.

– Much like the function comp (in fact much easier), we can show that the predicate $\mathsf{prf}_\Phi \subseteq \mathbb{N}^2$, defined by

$$\mathsf{prf}_\Phi(x, y) \text{ holds} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \begin{pmatrix} x = \ulcorner A \urcorner;\ y = \ulcorner \Pi \urcorner;\ \Pi \text{ is a proof in LK with} \\ \text{axioms from } \Phi; \text{ and the root of } \Pi \text{ is } \Rightarrow A, \end{pmatrix}$$

is recursive. Here it is crucial that Φ is recursive axiomatized.

– Then we have

$$\mathsf{thm}_\Phi(x) \text{ holds} \qquad \Longleftrightarrow \qquad \mathsf{prf}_\Phi(x, y) \text{ holds for some } y.$$

This proves that thm_Φ is RE. □

Here is an intuitive reading of the above result. To check if a given formula A is derivable under Φ (i.e. if $\Phi \vdash A$), we can "proof search," by checking if any of

$$\mathsf{prf}_\Phi(\ulcorner A \urcorner, 0),\ \mathsf{prf}_\Phi(\ulcorner A \urcorner, 1),\ \mathsf{prf}_\Phi(\ulcorner A \urcorner, 2),\ \ldots$$

is true, in the one-by-one manner. If $\Phi \vdash A$ then this proof search procedure stops eventually so we know $\Phi \vdash A$; if $\Phi \nvdash A$ then we wait forever.

9.3.8 Remark. Thm. 9.3.7 does not prohibit thm_Φ from being recursive. It is indeed recursive for some theories Φ: a notable class of examples is given by Thm. 9.3.9; another important example is the theory of real closed fields (see [6]) which has an important application (namely *quantifier elimination*) in automated theorem proving.

However there are also theories Φ for which thm_Φ is not recursive (but is RE). In fact, for a slightly complicated language (i.e. the choice of **FnSymb** and **PdSymb**), the predicate $\mathsf{thm}_\emptyset$ is not recursive.

9.3.9 Theorem. *Let Φ be a theory that is recursively axiomatized and complete (Def. 9.3.1). Then the predicate thm_Φ in (9.9) is recursive.*

Proof. The strategy is as follows. We know that the predicate thm_Φ is RE; we shall show that $\mathbb{N} \setminus \mathsf{thm}_\Phi$ is also RE. Then by Thm. 8.4.4 we conclude that thm_Φ is recursive.

Thus we are set out to show that $\mathbb{N} \setminus \mathsf{thm}_\Phi$ is RE. We have the following equivalences, for any $x \in \mathbb{N}$.

$$\mathsf{thm}_\Phi(x) \text{ does not hold} \tag{9.10}$$
$$\iff \quad \mathsf{fml}(x) \text{ does not hold, or } x = \ulcorner A \urcorner \text{ with } \Phi \nvdash A \tag{9.11}$$
$$\overset{(*)}{\iff} \quad \mathsf{fml}(x) \text{ does not hold, or } x = \ulcorner A \urcorner \text{ with } \Phi \nvdash \forall \overrightarrow{x}.\, A \tag{9.12}$$
$$\overset{(\dagger)}{\iff} \quad \mathsf{fml}(x) \text{ does not hold, or } x = \ulcorner A \urcorner \text{ with } \Phi \vdash \neg \forall \overrightarrow{x}.\, A \tag{9.13}$$
$$\overset{(\ddagger)}{\iff} \quad \mathsf{fml}(x) \text{ does not hold, or } \mathsf{thm}_\Phi(\mathsf{neg}(\mathsf{univClosure}(x))) \text{ holds.} \tag{9.14}$$

Here fml is from Lem. 9.3.4; $\forall \overrightarrow{x}.\, A$ is the universal closure of the formula A (Exercise 5.14); and the functions neg and $\mathsf{univClosure}$ are from Lem. 9.3.4. The equivalence $(*)$ is due to Exercise 9.2; $(\dagger)$ is because Φ is complete (Def. 9.3.1; note that $\forall \overrightarrow{x}.\, A$ is a closed formula). Now we have

- the predicate fml is recursive (Lem. 9.3.4); and
- the predicate
$$\boldsymbol{\lambda} x. \quad \mathsf{thm}_\Phi(\mathsf{neg}(\mathsf{univClosure}(x)))$$
is RE, because thm_Φ is RE (Thm. 9.3.7) and neg, $\mathsf{univClosure}$ are recursive (Lem. 9.3.4, Exercise 8.7).

Thus the predicate (9.14) is RE, from which we conclude that $\mathbb{N} \setminus \mathsf{thm}_\Phi$ is RE. □

This theorem (Thm. 9.3.9) is what is meant by "the classification (9.7) is necessarily simple." More precisely: if a recursively axiomatized theory Φ were to satisfy (9.5), then Φ is complete by Lem. 9.3.2; but then Thm. 9.3.9 yields that thm_Φ—i.e. if a given formula is derivable from Φ or not—is recursive.

9.4 The Arithmetic Truth is Undecidable

In this section we prove that the arithmetic truth is beyond the complexity of any recursively axiomatized theory. That is, the predicate $\mathsf{arithTruth} \subseteq \mathbb{N}$, defined by

$$\mathsf{arithTruth}(x) \text{ holds} \quad \overset{\text{def}}{\iff} \quad x = \ulcorner A \urcorner \text{ for some formula } A \text{ and } \mathbb{N} \models A, \tag{9.15}$$

is undecidable. This will conclude the proof of Gödel's incompleteness theorem. The undecidability proof is, as before, by the diagonal method (i.e. self-reference plus a negative twist).

We fix sets of function and predicate symbols to be the ones $\mathbf{FnSymb_a}, \mathbf{PdSymb_a}$ for arithmetic (from (9.4)).

9.4.1 Definition (Numeral)**.** For each $x \in \mathbb{N}$, we define the *numeral* k_x to be the following term.

$$k_x :\equiv s(s(\ldots s(0))) \quad \text{with } x\text{-many } s\text{'s.}$$

The following fact is the key to our diagonal proof. It means: arguments on recursive functions can be "simulated" in predicate logic (using $\mathbf{FnSymb}_{\mathsf{a}}$ and $\mathbf{PdSymb}_{\mathsf{a}}$). We do not present its proof (which is again like "programming in first-order logic"); an interested reader is referred to textbooks like [6].

9.4.2 Theorem (Representability)**.** *Any recursive predicate $P \subseteq \mathbb{N}$ can be* represented *by a predicate formula over $\mathbf{FnSymb}_{\mathsf{a}}$ and $\mathbf{PdSymb}_{\mathsf{a}}$. This means: for any given $u \in \mathbf{Var}$, there exists a formula A with $\mathrm{FV}(A) = \{u\}$ such that, for all $x \in \mathbb{N}$, we have*

$$P(x) \text{ holds} \quad \Longleftrightarrow \quad \mathbb{N} \models A[k_x/u] \ . \qquad \square$$

9.4.3 Theorem. *The predicate* $\mathsf{arithTruth}$ *is not recursive.*

Proof. By contradiction. Consider the following predicate ($\subseteq \mathbb{N}^2$):

$$\boldsymbol{\lambda} x.\boldsymbol{\lambda} y.\ \mathsf{arithTruth}(\mathsf{subst}(x, \ulcorner k_y \urcorner, \ulcorner u \urcorner))$$

where subst is the recursive function from Lem. 9.3.4—recall that we have $\mathsf{subst}(\ulcorner A \urcorner, \ulcorner k_y \urcorner, \ulcorner u \urcorner) = \ulcorner A[k_y/u] \urcorner$.

Now we take its diagonal, and twist that. Specifically, consider the following predicate ($\subseteq \mathbb{N}$).

$$\boldsymbol{\lambda} x.\ \neg\mathsf{arithTruth}(\mathsf{subst}(x, \ulcorner k_x \urcorner, \ulcorner u \urcorner)) \tag{9.16}$$

If $\mathsf{arithTruth}$ is recursive, so is this predicate, due to Exercise 8.6 and the fact that subst is *total* and recursive.

By Thm. 9.4.2, there is a formula A with $\mathrm{FV}(A) = \{u\}$ that represents the predicate (9.16). That is, for all $x \in \mathbb{N}$,

$$\begin{array}{lll}
\neg\mathsf{arithTruth}(\mathsf{subst}(x, \ulcorner k_x \urcorner, \ulcorner u \urcorner)) & & \\
\Longleftrightarrow & \mathbb{N} \models A[k_x/u] & \text{since } A \text{ represents the predicate} \\
\Longleftrightarrow & \mathsf{arithTruth}(\ulcorner A[k_x/u] \urcorner) & \text{by def. (9.15) of } \mathsf{arithTruth} \\
\Longleftrightarrow & \mathsf{arithTruth}(\mathsf{subst}(\ulcorner A \urcorner, \ulcorner k_x \urcorner, \ulcorner u \urcorner)) \ . &
\end{array}$$

Take $x = \ulcorner A \urcorner$; then

$$\begin{array}{ll}
 & \neg\mathsf{arithTruth}(\mathsf{subst}(\ulcorner A \urcorner, \ulcorner k_{\ulcorner A \urcorner} \urcorner, \ulcorner u \urcorner)) \\
\Longleftrightarrow & \mathsf{arithTruth}(\mathsf{subst}(\ulcorner A \urcorner, \ulcorner k_{\ulcorner A \urcorner} \urcorner, \ulcorner u \urcorner)) \ .
\end{array}$$

This is a contradiction. $\square$

Finally we answer (negatively) the question (9.5).

9.4.4 Theorem (Incompleteness)**.** *There is no recursively axiomatized theory Φ that characterizes the arithmetic truth, that is,*

$$\Phi \vdash A \qquad \Longleftrightarrow \qquad \mathbb{N} \models A \ . \tag{9.17}$$

Proof. By contradiction; assume there is such Φ. Then by Def. 9.3.1, Φ is complete (see also Lem. 9.3.2). This and the assumption that Φ is recursively axiomatized yield that the predicate thm_Φ is recursive (Thm. 9.3.9). However this contradicts Thm. 9.4.3, since (9.17) immediately yields $\mathsf{thm}_\Phi = \mathsf{arithTruth}$. $\square$

Exercises

9.1. Show that a complete theory Φ (Def. 9.3.1) is necessarily consistent (Def. 5.2.4). (Hint: use the (WEAKENING) rules)

9.2. Let A be a formula, and Φ be a theory. Show that, for its universal closure

$$\forall x_0.\, \forall x_1.\, \ldots \forall x_{m-1}.\, A$$

(where $\mathrm{FV}(A) = \{x_0, \ldots, x_{m-1}\}$, cf. Exercise 5.14), we have

$$\Phi \vdash A \quad \Longleftrightarrow \quad \Phi \vdash \forall x_0.\, \forall x_1.\, \ldots \forall x_{m-1}.\, A \ .$$

(Hint: use the ($\forall$-R) rule)

Bibliography

[1] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Math. Textbooks. Cambridge Univ. Press, 1990. 7

[2] H. Enderton. *Computability Theory: An Introduction to Recursion Theory*. Elsevier Science, 2010. 4

[3] J.Y. Girard, Y. Lafont and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989. Available online. 72

[4] B. Jacobs. *Categorical Logic and Type Theory*. North Holland, Amsterdam, 1999. 65

[5] R.C. Nelson. Course note on resolution. `www.cs.rochester.edu/~nelson/courses/csc_173/proplogic/resolution.html`, 1999. 88

[6] J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967. 4, 26, 68, 69, 130, 133, 135

[7] R.I. Soare. Computability and recursion. *Bulletin of Symbolic Logic*, 2:284–321, 1996. 93

[8] G. Takeuti. *Proof Theory*. North-Holland, 2nd edn., 1987. 68, 71

[9] 菊池 誠. **不完全性定理**. 共立出版, 2014. 4

[10] 戸次 大介. **数理論理学**. 東京大学出版会, 2012. 4

[11] 高橋 正子. **計算論——計算可能性とラムダ計算**. 近代科学社, 1991. 4, 107, 115

[12] 鹿島 亮. **不完全性定理と算術の体系**, vol. 3 of **ゲーデルと** *20* **世紀の論理学**, chap. 第一不完全性定理と第二不完全性定理. 東京大学出版会, 2007. 4, 130

[13] 鹿島 亮. *C* **言語による計算の理論**. サイエンス社, 2008. 4, 107, 115, 125

[14] 鹿島 亮. **数理論理学**. 朝倉書店, 2009. 71

[15] 小野 寛晰. **情報科学における論理**. 日本評論社, 1994. 4, 74, 84, 85

[16] 松阪 和夫. **集合・位相入門**. 岩波書店, 1968. 7

[17] 照井 一成. **コンピュータは数学者になれるのか? 数学基礎論から証明とプログラムの理論へ**. 青土社, 2015. 4

[18] 新井 敏康. **数学基礎論**. 岩波書店, 2011. 4

[19] 田中 一之. **数の体系と超準モデル**. 裳華房, 2002. 44, 69

[20] 萩谷 昌己 and 西崎 真也. **論理と計算のしくみ**. 岩波書店, 2007. 4

Index