Theory of
COMPUTATION

GEORGE TOURLAKIS

Theory of Computation

Theory of Computation

George Tourlakis

York University
Toronto, Canada

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

To my parents

CONTENTS

Preface

At the intuitive level, any practicing mathematician or computer scientist —indeed any student of these two fields of study— will have no difficulty at all to recognize a *computation* or an *algorithm*, as soon as they see one, the latter defining, *in a finite manner*, computations for any given input. It is also an expectation that students of computer science (and, increasingly nowadays, of mathematics) will acquire the skill to devise algorithms (normally expressed as computer programs) that solve a variety of problems.

But how does one tackle the questions "is there an algorithm that solves such and such a problem for all possible inputs?" —a question with a potentially "no" answer— and also "is there an algorithm that solves such and such a problem via computations that take no more *steps* than some (fixed) polynomial function of the input length?" —this, too, being a question with a, potentially, "no" answer.

Typical (and tangible, indeed "interesting" and practically important) examples that fit the above questions, respectively, are

- "is there an algorithm which can determine whether or not a given computer program (the latter written in, say, the C-language) is *correct*?"[1]

---

[1] A "correct" program produces, *for every input*, precisely the output that is expected by an *a priori* specification.

and

- "is there an algorithm that will determine whether or not any given Boolean formula is a tautology, doing so via computations that take no more *steps* than some (fixed) polynomial function of the input length?"

For the first question we have a definitive "no" answer,[2] while for the second one we simply do not know, at the present state of knowledge and understanding of what "computing" means.[3]

But what do we mean when we say that "there is *no algorithm* that solves a given problem" —with or without restrictions on the algorithm's computation lengths? This appears to be a much harder statement to validate than "there *is* an algorithm that solves such and such a problem" —for the latter, all we have to do is *to produce such an algorithm* and a proof that it works as claimed. By contrast, the former statement implies a, mathematically speaking, *provably failed search* over the entire set of *all algorithms*, while we were looking for one that solves our problem.

One evidently needs a precise definition of the concept of algorithm that is neither experiential, nor technology-dependent in order to assert that we encountered such a failed "search". This directly calls for a *mathematical theory* whose objects of study include *algorithms* (and, correspondingly, *computations*) in order to construct such sets of (all) algorithms within the theory and be able to reason about the membership problem of such sets. This theory we call the *theory of computation*. It contains tools which, *in principle*, can "search"[4] the set of all algorithms to see whether a problem is solvable by one; or, more ambitiously, to see if it can be solved by an algorithm whose computations are "efficient" —under some suitable definition of efficiency.

The theory of computation is the *metatheory* of computing. In the field of computing one computes: that is, develops programs and large scale software that are well-

---

[2]There is some interesting "small print" here! As long as the concept of *algorithm* is identified with that of, say, the Shepherdson-Sturgis "machines" of this volume —or for that matter with Turing machines— then the answer is definitely a "no": There is a simple mathematical proof that we will see later on, that no Shepherdson-Sturgis machine (nor a Turing machine) exists that solves the problem. Now, such an identification has been advocated by Alonzo Church as part of his famous belief known as "Church's Thesis". If one accepts this identification, then the result about the non-existence of a Shepherdson-Sturgis machine that solves the problem is tantamount to the non-existence of an *algorithm* that does so. However, Church's "thesis" is empirical, rather than provable, and is not without detractors; cf. Kalmár (1957). Suffice it to say that *this* statement is mathematically valid: *No program, written in any programming language, which is equivalent in expressive power to that of our Shepherdson-Sturgis machines, exists that solves the problem.*

[3]There is substantial evidence that the answer, if discovered, will likely be "no".

[4]The quotes are necessary since it is not precisely a *search* that one performs. For example, the unsolvability —by any algorithm— of the *program correctness problem* is based on a so-called *reduction* technique that we will learn in this volume. A reduction basically establishes that a problem A is solvable by algorithmic means if we *assume* that we have a "black-box" algorithmic solution —that we may "call" just as we call a built-in function— of another problem, B. We say that "A is reduced (or reducible) to B". If we now *know* (say via a previous mathematical proof of the fact) that A cannot be algorithmically solved, then nor can B! We will, as a starting point, show the unsolvability by algorithmic means, certainly not by any Shepherdson-Sturgis machine, of a certain "prototype" problem, known as *the halting problem*, "$x \in K$?". This will be done by a technique akin to Cantor's *diagonalization*. After this, many reduction arguments are effected by showing that K is reducible to a problem A. This renders A unsolvable!

documented, correct, efficient, reliable and easily maintainable. In the (meta)theory of computing one tackles the fundamental questions of the *limitations of computing*, limitations that are intrinsic rather than technology-dependent.[5] These limitations may rule out outright the existence of algorithmic solutions for some problems, while for others they rule out efficient solutions.

Our approach is anchored on the concrete (and assumed) practical knowledge about general computer programming attained by the reader in a first year programming course, as well as the knowledge of discrete mathematics at the same level. The next natural step then is to develop the metatheory of *general computing*, building on the computing experience that we have assumed the reader attained. This will be our chapter on computability, that is, the most general *meta*theory of computing. We develop this metatheory via the programming formalism known as Shepherdson-Sturgis *Unbounded Register Machines* (URM) —which is a straightforward abstraction of modern high level programming languages. Within that chapter we will also explore a restriction of the URM programming language, that of the *loop programs* of A. Meyer and D. Ritchie. We will learn that while these loop programs can only compute a very small subset of "all the computable functions", nevertheless are *significantly more than adequate* for programming solutions of any "practical", computationally solvable, problem. For example, even restricting the nesting of loop instructions to *as low as two*, we can compute —in principle— enormously large functions, which with input x can produce outputs such as

$$2^{2^{\cdot^{\cdot^{\cdot^{x}}}}} \left.\right\} 10^{350000} \text{ 2's} \tag{1}$$

The qualification above, "in principle", stems from the enormity of the output displayed in (1) —even for the input $x = 0$— that renders the above function way beyond "practical".

The chapter —after spending considerable care in developing the technique of *reductions*— concludes by demonstrating the intimate connection between the *unsolvability phenomenon* of computing on one hand, and the *unprovability phenomenon* of proving within first-order logic (cf. Gödel (1931)) on the other, when the latter is called upon to reason about "rich" theories such as (Peano's) arithmetic —that is, the theory of natural numbers, equipped with: the standard operations (plus, times); relations (less than); as well as with the principle of mathematical induction.

What to include and what not to include in an introductory book on the theory of computation is a challenge that, to some extend, is resolved by the preferences of the author. But I should like to think that the choices of topics made in this volume are more rational than simply being manifestations of "preference".

The overarching goal is to develop for the reader a "first-order" grounding in the fundamentals, that is, the theoretical limitations of computing in its various models of computation, from the most general model —the URM— down to the finite automaton.

---

[5]However this metatheory is called by most people "theory". Hence the title of this volume.

We view the technique of *reductions* as fundamental in the analysis of limitations of computing, and we spend a good deal of space on this topic, a variant of which (polynomial-time reductions) the student of computer science will encounter in Subsection 5.1.2 and will re-encounter in later studies as well, for example, in a course on algorithms and complexity. On the other hand, we do not hesitate to omit combinatorial topics such as "Post's correspondence problem", which only leads to specialized results (e.g., the algorithmic unsolvability of detecting ambiguity in context free languages) that we feel embody a less fundamental technical interest. Our emphasis is on laying the foundational tools and concepts that allow us to carry out a mathematical analysis of, and acquire a thorough understanding of, theoretical limitations of computing in both their absolute manifestation (uncomputability) and also in their relative manifestation (complexity and "intractability").

Consistent with our stated goal and emphasis, we purposely give short shrift to the area of so-called "positive" results, apart from a few familiarization examples of "programming" with URMs, loop programs, FA, NFA, and PDA. This is not a course about writing algorithms, but mostly about what algorithms *cannot do at all* and about what *they have a lot of trouble doing*. For example, results of Chapter 5 immediately imply that, in general, FORTRAN-like programs that allow nesting of the loop instruction equal to just *three* have highly *impractical* run times; certainly as high as[6]

$$\left. 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \right\} x \text{ 2's}$$

Thus, we leave out "applications" such as lexical scanners via finite automata; automata-minimization; parsing of context free languages using LL, LR, recursive-descend, and other parsers; and defer them to a later course on *compiler writing tools* —these topics do not belong here. We would rather concentrate on what is *foundationally* important and omit what is not.

Another challenge is where to *start* building this metatheory. What should be our abstraction of a computer program? It should be a straightforward observation that since this metatheory, or "theory" as we nickname it, *abstracts* computing practices —in order to analyze and study said abstractions mathematically— the student must have encountered in the first instance *the concrete counterparts* of these *abstractions* for the latter to make any sense.

It is hardly the case that, prior to the second year of study, students have "programmed" scanners or parsers. Rather, students have programmed solutions for less specialized problems, using a *high level general purpose* language such as C/C++, Java, possibly Pascal, etc. They never programmed an automaton, a push-down automaton, or anything like a Turing machine (unless they have taken up machine language in the first year).

Yet the overwhelming majority of the literature develops the "theory of computation", in a manner of speaking, backwards —invariably starting with the theory of

---

[6]See 5.2.0.47 and 5.2.0.49. L_3 programs have run times bounded by Ackermann's $A_3^k(x)$, for some $k > 0$.

finite automata, as if automata is precisely what the reader was programming in his[7] first university course on programming. We apply this principle: Before the student studies the (meta)theory, he must have attained a good grasp of the *practice* that this theory attempts to dissect and discuss. Thus, it is natural to start our story with the (meta)theory of *general purpose computer programs*.

Because of these considerations, our first chapter is on URMs and computability. The choice of URMs as an abstraction of general-purpose computing —a relative latecomer (cf. Shepherdson and Sturgis (1963)) in the search for a good answer to "what would be a good technology-independent model of computation?"— also connects well with the experience of the student who will come to this course to learn what makes things tick in programming, and why some things do not tick at all. He most likely learned his programming via a high level language like C or Java rather than through *machine language*. The ubiquitous Turing machine (Turing (1936, 1937)) is more like machine language, indeed, is rather even less user-friendly.[8] It offers no advantage at this level of exposition, and rather presents an obscure and hard-to-use (and hard to "arithmetize"[9]) model of computation that one *need not* use as the *basis* of computability. On the other hand it lends itself well to certain studies in complexity theory and is an eminently usable tool in the proof of Cook's theorem (cf. Subsection 5.1.3). So we will not totally avoid the Turing machine!

We turn to the formulaic topics of a book on *Automata and Languages* — Chapter 3— only after we become familiar, to some extent, with the (general) computability theory, including the special computability theory of more "practical" functions, the primitive recursive functions. *Automata* are introduced as a very restricted *programming formalism*, and their limitations (in expressivity) and their associated languages are studied.

It is often said, with justification, that a course in theory of computation has as side-effect the firming up of the student's grasp of (discrete) mathematical techniques and mathematical reasoning, as well as the ability to apply such techniques in computer science and beyond. Of course, it cannot be emphasized enough that the student of a theory of computation course must be equipped already with the knowledge expected to be acquired by the successful completion of a one-semester course on discrete mathematics. This required background knowledge is often encapsulated, retold, and aspects of it are emphasized, in the space of a few pages at the front-end of a book like this. This is the ubiquitous "Chapter 0" of many books on the subject. In the case of the present book I would like, most of all, to retell two stories, *logic* and *induction*, that I often found being insufficiently developed in the student's "toolbox", notwithstanding earlier courses he may have taken. Thus, in Subsection 1.1.1 we develop the notational and how-to parts of elementary predicate logic in the space of some 20 pages, paying special attention to correctness of exposition. Section 1.4 presents the induction principle on the natural numbers in two steps: One, how

---

[7]Pronouns such as "he", "his", "him" are, *by definition*, gender-neutral in this volume and are used solely for textual convenience.
[8]Machine language can manipulate *numbers*, whereas a Turing machine can only manipulate *digits*!
[9]This verb will make sense later.

to use its various forms, and a proof of their equivalence to the least (positive) integer principle. Two, we argue, at the intuitive level, why induction *must* be a *valid principle* after all![10] We also go over concepts about sets and related notation, as well as relations and functions, very quickly since they do not need much retelling. We will also introduce quickly and in an elementary fashion a topic likely not encountered by the reader in the typical "discrete math" course: the distinction between two infinities —*countable* and *uncountable*— so that we can have an excuse to introduce the reader to Cantor's ingenious (and simple) *diagonalization* argument, that recurs in one or another shape and form, over and over, in the computability and complexity part of the theory of computation.

On intuitive arguments; "formalization" and why a course in theory cannot be taught exclusively *by hand-waving*: The main reason that compels us to teach (meta)theory in a computer science curriculum is not so much to prevent the innocent from trying to program a solution for the halting problem (cf. 2.5.0.16), just as we do not teach courses in geometry just to prevent circle-squaring "research". Rather, formal mathematical methods used in a course in the theory of computation, more so than the results themselves, are transferable skills that the student becomes endowed with, which equip him to model and mathematically analyze concrete phenomena that occur in computation, and through a mathematical process of reasoning to be able to recognize, understand, and correlate such phenomena. These formal methods, skills and results, put the "science" keyword into *computer science*.

Intuition, obtained through experience, is invaluable, of course, and we often argue intuitively *before* we offer a proof of a fact. But: one *cannot* have "proof-by-intuition".

We have included in this volume a good amount of *complexity theory* that will likely be mostly skipped whenever the book is called upon to serve a second year course on the theory of computation. There are a few "high level complexity" results already in Section 2.7 using diagonalization (cf. 2.7.1.9 and 2.7.1.11). Later, quite a bit is developed in Chapter 5, including the concept of $\mathcal{N}\mathcal{P}$-completeness and Cook's theorem; an account of Cobham's class of *feasibly computable functions* (mostly delegated to the Exercises section, 5.3); and some elements of the hierarchy theory of the primitive recursive functions culminating in the rather startling fact that we cannot algorithmically solve the *correctness problem* of FORTRAN-like programs even if we restrict the *nesting of loops* to just *two levels*. FORTRAN-like languages have as abstract counterpart the *loop programs* of Meyer and Ritchie (1967) that we study in the chapters on computability (2nd) and complexity (5th).

Were I to use this book in a second year course in the theory of computation I would skim quickly over the mathematical "prerequisites" chapter, and then cover 2.1–2.7, parts of 2.10, certainly Gödel's incompleteness theorem and its relation to *uncomputability*: 2.11 —but not 2.11.1. I would then cover only as much as time permits from Chapter 3 on finite automata; certainly the pumping lemma, consistent

---

[10]In so doing I will be sure to let the student know that I am not squaring the circle: Induction is not a provable principle of the arithmetic of Peano, it is an axiom. However, this will not stop us from arguing its *plausibility*, i.e., why it is a *reasonable*, "natural" axiom.

with my view that this is a "course" about what *cannot* be done, or cannot be done "easily", rather than a toolbox for how to *do* things. The latter is deferred to a course and book on algorithms.

In a more advanced course where one can proceed faster, I would want also to cover the sections on creative sets and the recursion theorem, and also as much complexity theory as possible from Chapter 5, starting with the material leading to Cook's theorem.

The reader will forgive the many footnotes, which some will assess as bad style! There is always a story within a story, the ". . . and another thing . . . ", that is best delegated to footnotes.

The style of exposition that I prefer is informal and conversational and is expected to serve well not only the readers who who have the guidance of an instructor, but also those readers who wish to learn the elements of the theory of computation on their own. I use several devices to promote understanding, such as frequent "pauses" that anticipate questions and encourage the reader to rethink an issue that might be misunderstood if read but not studied and reflected upon. Additionally, I have included numerous remarks, examples and embedded exercises (the latter in addition to the end-of-chapter exercises) that reflect on a preceding definition or theorem. All pauses are delimited by "**Pause.**" and ◄

The stylized "winding road ahead" warning, that I first saw in Bourbaki's books (Bourbaki (1966)) and have used in my other books, delimits a passage that is too *important* to skim over.

On the other hand, I am using to delimit passages that I could not resist including, but, frankly, can be skipped (unless you are curious).

There are over 200 end-of-chapter exercises and 41 embedded ones. Many have hints and thus I refrained from (subjectively) flagging them for level of difficulty. After all, as one of my mentors, Alan Borodin, used to say to us (when I was a graduate student at the University of Toronto), "attempt all exercises; but definitely do the ones you cannot do".

GEORGE TOURLAKIS

Toronto
November 2011

CHAPTER 1

MATHEMATICAL FOUNDATIONS

In this chapter we will briefly review tools, methods and *notation* from mathematics and logic, which we will directly apply throughout the remaining of this volume.

1.1 SETS AND LOGIC; NAÏVELY

The most elementary elements from "set theory" and logic are a good starting point for our review. The quotes are necessary since the term *set theory* as it is understood today applies to the *axiomatic* version, which is a vast field of knowledge, methods, tools and research [cf. Shoenfield (1967); Tourlakis (2003b)]—and this is not what we outline here. Rather, we present the standard notation and the elementary operations on sets, on one hand, and take a brief look at infinity and the *diagonal method* of Cantor's, on the other. Diagonalization is a tool of significant importance in computability. The tiny fragment of concepts from set theory that you will find in this section (and then see them applied throughout this volume) are framed within Cantor's original "naïve set theory", good expositions of which (but far exceeding our needs) can be found in Halmos (1960) and Kamke (1950).

We will be forced to interweave our exposition of concepts from set theory with concepts—and notation—from elementary logic, since all mathematics is based on

logical deductions, and the vast majority of the literature, from the most elementary to the most advanced, employs logical notation; e.g., symbols such as "∀" and "∃".

The term "set" is *not defined*,[11] in either the modern or in the naïve Cantorian version of the theory. Expositions of the latter, however, often ask the reader to think of a *set* as just a synonym for the words "class",[12] "collection", or "aggregate". Intuitively, a set is a "container" along with its contents—its *elements* or *members*. Taken together, contents and container, are viewed as a *single mathematical object*. In mathematics one deals only with sets that contain mathematical objects (so we are not interested in sets of mice or fish).

Since a set is itself an *object*, a set may contain sets as elements.

All the reasoning that one does in order to develop set theory—even that of the naïve variety—or any part of mathematics, including all our reasoning in this book, utilizes mathematical logic. Logic is the mathematics of reasoning and its "objects" of study are predominantly mathematical "statements" or "assertions"—technically known as *formulae*[13]—and mathematical proofs. Logic can be applied to mathematics either experientially and informally—learned via practice as it were—or formally. The predominance of mathematical writings apply logic informally as a vehicle toward reaching their objectives.[14] Examples of writings where logic is formally applied to mathematics are the volumes that Bourbaki wrote, starting here [Bourbaki (1966)]. More recent examples at the undergraduate and graduate levels are Gries and Schneider (1994) and Tourlakis (2003b) respectively.

In this volume we apply logic informally. An overview is provided in the next subsection.

1.1.1 A Detour via Logic

As is customary in mathematics, we utilize *letters*, upper or lower case, usually from near the end of the alphabet (u, v, y, x, z, S, T, V) to *denote*, that is, to *name* mathematical objects—in particular, *sets*.

By abuse of language we say that u, v, y, x, z, S, T, V are (rather than *denote* or *name*) objects. These letters function just like the variables in algebra do; they are *object-variables*.

---

[11]The reader who has taken Euclidean geometry in high school will be familiar with this parallel: The terms "point", "line", and "plane" are not defined either, but we get to know them intimately through their properties that we develop through mathematical proofs, starting from Euclid's axioms.

[12]In axiomatic set theory a "class" is a kind of collection that may be so "large" that it technically fails to be a set. The axioms force sets to be "small" classes.

[13]More accurately, a "statement" and a formula are two different things. However, the latter mathematically "encodes" the former.

[14]Despite the dangers this entails, as Gödel's incompleteness theorems exposed [Gödel (1931)], modern mathematicians are confident that their subject and tools have matured enough, to the point that one can safely apply logic, once again, post-Gödel, informally. For example, Kunen states in his article on set-theoretic *combinatorics*, Kunen (1978), "A knowledge of [formal] logic is neither necessary, nor even desirable".

As is the case in algebra, the variables x, y, z are not the only objects set theory studies. It also studies numbers such as $0, 1, -7$ and π, matrices such as $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ and objects that are the results of function applications such as 7^{23000}, x^{y^z} and 2^x.

Unlike axiomatic set theory, which introduces its objects via formal constructions, naïve set theory allows us to use, "off the shelf", all the mathematical objects such as the above, as well as, of course, objects that are sets such as $\{2, 3, \{1\}\}$ and $A \cup B$.[15]

Logicians like to call mathematical objects *terms*. We utilize in this book the generic names t and s (with primes or subscripts, whenever we need more than two such names) to refer to arbitrary terms that we do not want to be specific about.

1.1.1.1 Definition. The *simplest* possible relations of set theory are of just *two forms*: $t \in s$—read "t is a member of s" or "t belongs to s"—and $t = s$, read "t is equal to s", where, as we indicated above, t and s are any terms whatsoever.

These relations are the *atomic formulae* (of set theory). The qualifier "atomic" refers to two facts:

- These two types cannot be expressed (simulated) in terms of simpler relations by using the notation and tools of logic.

- Using these two relations as building blocks we can *construct* every possible formula of set theory as we will explain shortly. □

1.1.1.2 Example. $x \in y$, $u = v$, $z \in S$ and $3 \in z$ and $2^z = y^3$ are atomic formulae.

$\mathbb{N}$, the set of all *natural numbers* (i.e., all the numbers that we obtain by starting at 0 and repeatedly adding 1: $0, 1, 2, 3, 4, \ldots$), is an important constant in naïve set theory.

By "$\mathbb{N} \ldots$ *is* an important constant" we mean, of course, via the habitual abuse of language exercised by mathematicians, the accurate "$\mathbb{N} \ldots$ *denotes* (or *names*) an important constant".

Here is an example that uses $\mathbb{N}$ in an atomic formula: $-7 \in \mathbb{N}$. Incidentally, this formula makes (i.e., encodes) a false statement; we say the *formula is false*.

One may form this basic formula as well, $\mathbb{N} = \bigcup_{i=0}^{\infty} \{i\}$, where the meaning of the symbols "$\{\ldots\}$" and "$\bigcup_{i=0}^{\infty}$" will be introduced later in this section.

Yet another example is $\{1\} \in \{2, 1\}$—a false statement (formula) as we will be able to determine soon. □

Logic (and mathematics) contain much more complex formulae than those of the atomic variety. The added complexity is achieved by repeatedly "gluing" atomic formulae together employing as glue the logical, or *Boolean, connectives*

$$\neg, \wedge, \vee, \rightarrow, \equiv$$

---

[15]Notation for objects such as $\{\ldots\}$ and $x \cup y$ will be reviewed shortly.

and the *quantifiers*

$$\forall, \exists$$

As we have noted already, unlike the case of naïve set theory—where we take for granted the *a priori* presence of *all* objects of mathematics, such as 3, −7, $\mathbb{N}$ and $x^{y^{z^u}}$ — axiomatic set theory needs no *a priori* existence of any objects. Starting just with the relations $x \in y$ and $x = y$ it uses powerful rules, which can be used to build not only all formulae of set theory, but also all the objects of mathematics that we are familiar with, such as the above-mentioned and many others.

What about arithmetic? The arithmetical objects of "pure" (Peano) arithmetic are the variables, constants, and outputs of functions applied on objects that we have already built. What are its formulae? If we are thinking of pure arithmetic, which is studied outside set theory, then we may choose as atomic formulae all those that can be built from the three start-up relations $z = x + y$, $z = x \times y$ and $z = x^y$: new atomic formulae result by substituting arbitrary (arithmetical) objects for variables. Note that the equality relation is obtained from $z = x + y$ by substituting 0 for y.

All formulae of arithmetic can be built, starting from the atomic ones, as explained in the general Definition 1.1.1.3 below. This assertion is revisited in Subsection 2.11.1.

Gödel showed in Gödel (1931) that the atomic formula $z = x^y$ is, well, *not atomic*: It can be simulated (built) within pure arithmetic starting just with $z = x + y$ and $z = x \times y$.

The "practicing mathematician" prefers to work within an "impure" arithmetic, where he has access to sets and their notations, operations, and properties. In particular, this impure arithmetic employs set variables and, more generally, set objects in addition to number variables and number objects.

Throughout this volume a formula (whether specific to set theory or to any other area in mathematics, such as arithmetic—pure or impure) will be *denoted* by an upper case calligraphic letter, such as $\mathscr{A}, \mathscr{B}, \mathscr{F}, \mathscr{G}$.

We now indicate how formulae are put together using brackets, connectives, and quantifiers, employing atomic formulae as basic building blocks. The definition below is generic, thus unified: it applies to the structure of all formulae of mathematics. The choice of atomic formulae (which presupposes an *a priori* choice of mathematical symbols, such as $0, +, \in$) and of types of variables is what determines whether we build set theory formulae, pure or impure arithmetic formulae, or "other".

1.1.1.3 Definition. A set theory formula is one of:

(1) An atomic formula (1.1.1.1).

(2) $(\neg\mathscr{A})$, where $\mathscr{A}$ is known to be[16] a formula.

---

[16]I.e., to stand for one. Thus, the expression "$(\neg\mathscr{A})$" is constructed by writing "(", followed by writing "$\neg$", followed by *writing in full* whatever $\mathscr{A}$ names, and finally writing ")".

(3) $(\mathscr{A} \wedge \mathscr{B})$, where $\mathscr{A}$ and $\mathscr{B}$ are known to be formulae.

(4) $(\mathscr{A} \vee \mathscr{B})$, where $\mathscr{A}$ and $\mathscr{B}$ are known to be formulae.

(5) $(\mathscr{A} \rightarrow \mathscr{B})$, where $\mathscr{A}$ and $\mathscr{B}$ are known to be formulae.

(6) $(\mathscr{A} \equiv \mathscr{B})$, where $\mathscr{A}$ and $\mathscr{B}$ are known to be formulae.

(7) $((\forall x)\mathscr{A})$, where $\mathscr{A}$ is known to be a formula and x is any variable.

(8) $((\exists x)\mathscr{A})$, where $\mathscr{A}$ is known to be a formula and x is any variable. We say in the last two cases that "$\mathscr{A}$ is the *scope* of Qx, where Q is $\forall$ or $\exists$".

We call $\forall$ the *universal* and $\exists$ the *existential* quantifiers. We will extend the terminology "quantifier" to apply to the compound symbols $(\forall x)$ or $(\exists)$. $\qquad \square$

1.1.1.4 Definition. (Immediate Predecessors) Let $\mathscr{F}$ be a formula. By 1.1.1.3 it has one of the forms (1)–(8). If it is of type (1), then it has no *immediate predecessors*— i.e., it was not built using connectives or quantifiers from simper formulae. If it has the forms (2)–(8), then in each case its immediate predecessors are the formulae $\mathscr{A}$ and $\mathscr{B}$ [the latter enters in cases (3)–(6)] that were used to build it. We use the acronym *ip* for *immediate predecessors*. $\qquad \square$

The presence of brackets guarantees that the decomposition or deconstruction of a formula into its immediate predecessors is unique. This fact can be proved, but it is beyond our aims so we will not do so here [see Bourbaki (1966); Enderton (1972); Tourlakis (2008, 2003a)]. Logicians refer to it as the *unique readability* of a formula.

1.1.1.5 Example. Here are some formulae:
$x \in y, 3 = z, z = x^w$—by (1),
$(\neg x = y)$—by (1), followed by an application of (2); we usually write this more simply as "$x \neq y$",
$(x \in y \vee z = x^w)$—by (1), followed by an application of (4),
$((\forall x)z = x^w)$—by (1), followed by an application of (7),
$(x = 0 \rightarrow x = 0)$—by (1), followed by an application of (5), and
$(x = 0 \rightarrow ((\forall x)x = 0))$—by (1), followed by an application of (7) to obtain $((\forall x)x = 0)$, and then by an application of (5).

The reader should check that we inserted brackets precisely as prescribed by Definition 1.1.1.3. $\qquad \square$

1.1.1.6 Remark. (Building a formula) If $\mathscr{F}$ is (stands for, that is) a formula we can deconstruct it according to Definition 1.1.1.3 using a natural process.

Initialize: Write down $\mathscr{F}$. Flag it *pending*.
Repeat this process *until it cannot be carried further*:
$\left\{\vphantom{\begin{array}{c}a\\a\end{array}}\right.$

Write down, *above* whatever you have written so far, the *ip* of all *pending* formulae (if they have ip); and *remove* the flag "pending" from the latter. *Add* the flag to the ones you have just written.

$$\left.\begin{array}{c}\\\\\end{array}\right\}$$

The process is terminating since we write *shorter and shorter formulae* at every step (*and* remove the flags); we cannot do this forever!

Clearly, if we now review *from top to bottom* the sequence that we wrote, we realize that it traces forward the process of constructing $\mathscr{F}$ by repeated application of Definition 1.1.1.3. This top-down view of our "deconstruction" is a *formula-construction sequence* for $\mathscr{F}$.

For example, applying the process to the last formula of the preceding example we get:

$$x = 0$$
$$x = 0$$
$$((\forall x)x = 0)$$
$$(x = 0 \rightarrow ((\forall x)x = 0))$$

where one copy of $x = 0$ was contributed by the bottom formula and the other (at the top) by $((\forall x)x = 0)$.

Going forward we can discard copies that we do not need. Thus a valid formula construction is also this one:

$$x = 0$$
$$((\forall x)x = 0)$$
$$(x = 0 \rightarrow ((\forall x)x = 0))$$

Indeed, we validate the first formula in the sequence via (1) of 1.1.3; the second using the first and (7); and the last one using the first two and (5).   □

A term such as x^2 has x as its only input variable. An atomic formula such as $z \in \mathbb{N}$ has z as its only input variable, while the (atomic) formula $x + y = y^w$ has x, y and w as input variables. Whenever we want to draw attention to the input variables—say, x, u, S and z—of a term t or a formula $\mathscr{A}$ we will write $t(x, u, S, z)$ or $\mathscr{A}(x, u, S, z)$, respectively. This is entirely analogous to writing "$f(x, z) = x^2 + \sin z$" in order to name the expression (term) $x^2 + \sin z$ as a function $f(x, z)$ of the two listed variables.

1.1.1.7 Definition. (Input Variables—in Terms) All the variables that occur in a term—other than an x that occurs in a term of the form $\{x : \ldots\}$ (which is a set object that will be introduced shortly)—are input variables.   □

1.1.1.8 Example. Thus, the term x has x as its only input variable; while the term 3 has no input variables. $x^{2^{z^y}}$ has x, z, y as its input variables. We will soon introduce terms (set objects) such as $\{x : x = 0\}$. This object, which the reader may recognize as a fancy way to simply write $\{0\}$, has no input variables.   □

1.1.1.9 Definition. (Input Variables—in Formulae) A variable occurrence[17] in an atomic formula $t \in s$ or $t = s$ is an *input occurrence* precisely if it is an input occurrence in one of the terms t and s. Thus, "$0 \in \{x : x = 0\}$" has no input variables while "$x = 0$" has one.

Formation rules (2)–(6) in Definition 1.1.1.3 "*preserve*" the input occurrences of variables in the constituent formulae $\mathscr{A}$ and $\mathscr{B}$ that we join together using one of $\neg, \wedge, \vee, \rightarrow, \equiv$ as glue. On the other hand, each quantifier $(\forall z)$ or $(\exists z)$ *forces* each occurrence of a variable as described below to become non-input:

- The occurrence z in the quantifier

- Any occurrence of z in the scope of said $(\forall z)$ or $(\exists z)$

Thus, if we start with $\mathscr{A}(x, y, z)$, of inputs x, y, z, the new formula $((Qy)\mathscr{A}(x, y, z))$, where Q stands here for one of $\forall, \exists$, has only x and z as input variables.   □

We have carefully referred to *occurrences*, rather than *variables*, in the above definition. A *variable can* be both input and non-input. An *occurrence cannot* be both. For example, in $(x = 0 \rightarrow (\forall x)x = 0)$ the first x-occurrence is input; the last two are non-input. The *variable* x is both.

Thus "x is an input/non-input variable" (of a formula) means that there are occurrences of x that are input/non-input.

The standard name utilized in the literature for input variables is *free variables*. Non-input variable occurrences are technically called *bound occurrences*, but are also called *apparent occurrences*, since even though they are visible, they are not allowed—indeed it makes no sense—to receive arguments (input). This is analogous to the "Σ-notation" for sums: $\sum_{i=1}^{3} i$ means $1 + 2 + 3$. While we can "see" the variable i, it is not really there![18] It cannot accept inputs. For example, "$\sum_{2=1}^{3} 2$" is total nonsense.

The jargon input/non-input *is deliberately chosen: We may substitute terms only in those variable occurrences that are free (input).*

If $\mathscr{F}$ is some formula and $x, y, z, \ldots$ is the *complete* list of variables that occur in it, we can draw attention to this fact by writing $\mathscr{F}(x, y, z, \ldots)$. If $x, y, z, \ldots$ is a list of variables such that *some*[19] among them occur in $\mathscr{F}$, then we indicate this by $\mathscr{F}[x, y, z, \ldots]$.

In the context of $\mathscr{F}[x, y, z, \ldots]$ [or $\mathscr{F}(x, y, z, \ldots)$], $\mathscr{F}[t_1, t_2, t_3, \ldots]$ [correspondingly $\mathscr{F}(t_1, t_2, t_3, \ldots)$] stands for the formula obtained from $\mathscr{F}$ by replacing each *original* occurrence of $x, y, z, \ldots$ in $\mathscr{F}$ by the terms $t_1, t_2, t_3, \ldots$ respectively.

Some people call this operation *simultaneous* or *parallel* substitution. Thus, if $\mathscr{F}[x, y]$ names "$x = y$", whereas t_1 is $y + 1$, and t_2 is 5, then $\mathscr{F}[t_1, t_2]$ is "$y + 1 = 5$" and *not* "$5 + 1 = 5$". The latter result would have been obtained if we first substituted

[17]For example, in $x = x$ the variable x has two occurrences.
[18]A fact demonstrated strongly by the explicit form of the sum, $1 + 2 + 3$.
[19]"Some" includes "none" and "all" as special cases.

t_1 in x to obtain $y + 1 = y$, and *then* substituted t_2 in y to obtain $5 + 1 = 5$. If we are to do this "simultaneous substitution" right, then we must not substitute t_2 into the y to the left of "$=$"; this y is *not* "original".

Observe also that if x does *not* occur in $\mathscr{F}[x]$, then $\mathscr{F}[t]$ is just the *original* $\mathscr{F}$.

Before we turn to the *meaning* (and naming) of the connectives and quantifiers, let us agree that we can get away with much fewer brackets than Definition 1.1.1.3 prescribes. The procedure to do so is to *agree* on connective and quantifier "priorities" so that we know, in the absence of brackets, which of the two connectives/quantifiers is supposed to "win" if they both compete to apply on the same part in a formula.

By analogy, a high school student learns the convention that "$\times$ has a higher priority than $+$", thus $2 + 3 \times 4$ means $2 + (3 \times 4)$—that is, $\times$ rather than $+$ claims the part "3".

Our convention is this: The connective $\neg$ as well as the quantifiers $\forall$ and $\exists$ have the highest priority, equal among the three. In order of decreasing priority, the remaining *binary* connectives[20] are listed as $\wedge, \vee, \rightarrow, \equiv$. If two binary connectives compete to glue with a subformula, then the higher-priority one wins. For example, assuming that $\mathscr{A}$ has already in place all the brackets that are prescribed by Definition 1.1.1.3, then $\ldots \rightarrow \mathscr{A} \vee \cdots$ means $\ldots \rightarrow \left(\mathscr{A} \vee \cdots\right.$, while $\ldots \neg \mathscr{A} \wedge \cdots$ means $\ldots \left(\neg \mathscr{A}\right) \wedge \cdots$.

If *two instances of the same binary connective* compete to glue with a subformula, then *the one to the right* wins. For example, assuming that $\mathscr{A}$ has all the brackets prescribed by Definition 1.1.1.3 in place, then $\ldots \rightarrow \mathscr{A} \rightarrow \cdots$ means $\ldots \rightarrow \left(\mathscr{A} \rightarrow \right.$ $\cdots$.

Similarly, if any of $\neg, \forall, \exists$ compete for a part of a formula, again *the one to the right* wins. E.g., $\ldots \neg(\forall x)(\exists y)\mathscr{A} \cdots$ means $\ldots \left(\neg\left((\forall x)\left((\exists y)\mathscr{A}\right)\right)\right) \cdots$, where once again we assumed that $\mathscr{A}$ has all the brackets prescribed by Definition 1.1.1.3 already in place.

How do we "compute" the truth or falsehood of a formula? To begin with, to succeed in this we must realize that just as a function gives, in general, different outputs for different inputs, in the same way the "output" of a formula, its *truth-value*, can only be computed, in general, if we "freeze" the input variables. For each such frozen instance of the input side, we can compute the output side: true or false.

But where do the inputs come from? For areas of study like calculus or arithmetic the answers are easy: From the set of real numbers—denoted by $\mathbb{R}$—and the set of natural numbers respectively.

For set theory it sounds easy too: From the set of all sets!

If it were not for the unfortunate fact that "the set of all sets" does not exist, or, to put it differently, it is a *non-set class* due to its enormity, we could have left it

---

[20]"Binary" since they each glue *two* subformulae.

at that. To avoid paradoxes such as "a set that is *not* a set"—cf. Section 1.3 on diagonalization for an insight into why some collections cannot be sets—we will want to take our inputs from a (comfortably large) *set* in any given set-theoretic discussion: the so-called *reference set* or *domain*.

1.1.1.10 Remark. The mathematician's intuitive understanding of the statement "$\mathscr{F}$ is *true* (resp. *false*)" is that "$\mathscr{F}$ is true (resp. false) for *all* the possible values of the free (input) variables of $\mathscr{F}$".

Thus, if we are working in arithmetic, "$2n + 1$ is odd" means the same thing as "it is true that, for all $n \in \mathbb{N}$, $2n + 1$ is odd". "$2^n > n$" again omits an implied prefix "it is true that, for all $n \in \mathbb{N}$". An example of a false statement *with* input variables is "$2n$ is odd". □

1.1.1.11 Definition. An *instance* of a formula $\mathscr{F}$, in symbols $\mathscr{F}'$, is a formula obtained from $\mathscr{F}$ by replacing *each* of its variables by some value from the relevant reference set.

Clearly, $\mathscr{F}'$ is variable-free—a so-called *closed formula* or *sentence*—and therefore it has a well-defined truth-value: exactly one of *true* or *false*.

Sometimes we use more explicit notation: An instance of $\mathscr{G}(x, y, z, \ldots)$ or of $\mathscr{G}[x, y, z, \ldots]$ is $\mathscr{G}(i, j, k, \ldots)$ or $\mathscr{G}[i, j, k, \ldots]$, respectively, where $i, j, k, \ldots$ are objects (constants) from the reference set.

$\mathscr{F}'$ and $\mathscr{G}'$ are *consistent* or *common* instances of $\mathscr{F}$ and $\mathscr{G}$ if *every* free variable that appears in both of the latter receives the same value in both instances. □

1.1.1.12 Example. Let $\mathscr{A}$ stand for "$x(x + 1)$ is even", $\mathscr{B}$ stand for "$2x + 1$ is even" and $\mathscr{C}$ stand for "x is even", where x is a variable over $\mathbb{N}$. Then,

$$\mathscr{A} \quad \text{is true,}$$
$$\mathscr{B} \quad \text{is false, and}$$
$$\mathscr{C} \quad \text{is neither true, nor false.}$$

The lesson from this is that if the truth-value of a formula depends on variables, then *not true* is *not* necessarily the same as *false*. □

We will not be concerned with how the truth-value of atomic formulae is "computed"; one can think of them as entities analogous to "built-in functions" of computer programming: *Somehow, the way to compute their true/false output is a matter that a hidden procedure (alternatively, our math knowledge and sophistication) can do for us.*

Our purpose here rather is to describe how the *connectives and quantifiers* behave toward determining the truth-value of a complex formula.

In view of Remark 1.1.1.10, the road toward the semantics of $\mathscr{A} \vee \mathscr{B}$, $((\forall x)\mathscr{A})$, etc., passes through the semantics of arbitrary instances of these; namely, we need to only define the meaning of $\mathscr{A}' \vee \mathscr{B}'$, $((\forall x)\mathscr{A})'$, etc., respectively.

1.1.1.13 Definition. (Computing with Connectives and Quantifiers) Let $\mathscr{A}$ and $\mathscr{B}$ be any formulae, and $\mathscr{A}'$ and $\mathscr{B}'$ be arbitrary *common* instances (1.1.1.11).

(1) $\neg\mathscr{A}'$—pronounced "not $\mathscr{A}'$"—is true iff[21] $\mathscr{A}'$ is false.

(2) $\mathscr{A}' \vee \mathscr{B}'$—pronounced "$\mathscr{A}'$ or $\mathscr{B}'$"—is true iff either $\mathscr{A}'$ is true or $\mathscr{B}'$ is true, or *both* (so-called *inclusive or*).

(3) $\mathscr{A}' \wedge \mathscr{B}'$—pronounced "$\mathscr{A}'$ and $\mathscr{B}'$"—is true iff $\mathscr{A}'$ is true *and* $\mathscr{B}'$ is true.

(4) $\mathscr{A}' \to \mathscr{B}'$—pronounced "if $\mathscr{A}'$, then $\mathscr{B}'$"—is true iff either $\mathscr{A}'$ is false *or* $\mathscr{B}'$ is true, or *both*.[22]

(5) $\mathscr{A}' \equiv \mathscr{B}'$—pronounced "$\mathscr{A}'$ iff $\mathscr{B}'$"—is true just in case[23] $\mathscr{A}'$ and $\mathscr{B}'$ are both true or both false.

(6) The instance $(\forall x)\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$—which is pronounced "for all x, $\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$ (holds)"[24]— is true iff, for *all* possible values k of x from the domain, $\mathscr{A}(i_1, \ldots, i_m, k, j_1, \ldots, j_n)$ is true.

(7) The instance $(\exists x)\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$ —which is pronounced "for some x, $\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$ (holds)"—is true iff, for *some* value k of x from the domain, $\mathscr{A}(i_1, \ldots, i_m, k, j_1, \ldots, j_n)$ is true. □

1.1.1.14 Remark. (Truth Tables) The content of the preceding definition—cases (1)–(5)—is normally captured more visually in table form (we have removed the primes for readability):

$\mathscr{A}$	$\mathscr{B}$	$\neg$	$\mathscr{A}$	$\mathscr{A} \vee \mathscr{B}$	$\mathscr{A} \wedge \mathscr{B}$	$\mathscr{A} \to \mathscr{B}$	$\mathscr{A} \equiv \mathscr{A}$
f	f	t		f	f	t	t
f	t	t		t	f	t	f
t	f	f		t	f	f	f
t	t	f		t	t	t	t

We read the table as follows: First, the symbols **t** and **f** stand for the values "true" and "false" respectively. Second, the two columns to the left of the vertical line $\|$ give all possible pairs of values (outputs) of $\mathscr{A}$ and $\mathscr{B}$. Third, below $\neg\mathscr{A}$, $\mathscr{A} \vee \mathscr{B}$, etc., we list the computed truth-values (of the formulae of the first row) that correspond to the assumed $\mathscr{A}$ and $\mathscr{B}$ values.

The odd alignment under $\neg\mathscr{A}$ is consistent with all the others: It emphasizes the placement of the "result" under the "operator"—here $\neg$—that causes it. □

---

[21] *if and only if*

[22] Other approaches to "implication" are possible. For example, the *Intuitionists* have a different understanding for $\to$ than that of the majority of mathematicians, who adopt the classical definition above.

[23] A synonym of "iff".

[24] The verb "holds" means "is true".

1.1.1.15 Remark. According to Remark 1.1.1.10,

$$(\forall x)\mathscr{A}(y_1, \ldots, y_m, x, z_1, \ldots, z_n) \text{ is true} \tag{†}$$

means precisely this:

> *For every choice of the i_l and j_r from the reference set,*
> $(\forall x)\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$ is true $\qquad\qquad$ (*)

By 6 of Definition 1.1.1.13, (*) means

> For every choice of the i_l and j_r from the reference set,
> *and*
> for *all* possible values k of x from the domain,

$$\mathscr{A}(i_1, \ldots, i_m, k, j_1, \ldots, j_n) \text{ is true}$$

The above, and hence also (†), translate via Remark 1.1.1.10 as

$$\mathscr{A}(y_1, \ldots, y_m, x, z_1, \ldots, z_n) \text{ is true} \tag{‡}$$

Iterating this observation yields that (‡) is an equivalent statement to the one we obtain by quantifying universally any—in particular, *all*—of the variables $y_1, \ldots, y_m$, $x, z_1, \ldots, z_n$ of $\mathscr{A}$. That is,

> Adding or removing a "$(\forall x)$" at the leftmost end of the formula makes no difference to the latter's meaning.

Hm. This begs the question: Then what do we need the universal quantifier for? □

1.1.1.16 Example. We note easily that, say, with $\mathbb{R}$ (the reals) as our domain, $x = 0 \rightarrow x = 0$ is true (cf. 4 in 1.1.1.13). However, $x = 0 \rightarrow (\forall x)x = 0$ is not, since its instance $0 = 0 \rightarrow (\forall x)x = 0$ is false: to the left of $\rightarrow$ we have true, while to the right we have false.

Thus, adding or removing a "$(\forall x)$" to *parts* of a formula *can* make a difference in meaning! The universal quantifier is useful after all. □

Carrying around the predicate "is true" all the time is annoying. We will adopt immediately the mathematician's jargon: *Simply stating "$\mathscr{A}(x, y, \ldots)$" is synonymous* to "$\mathscr{A}(x, y, \ldots)$ is true" or "$\mathscr{A}(x, y, \ldots)$ holds".

1.1.1.17 Example. Let $\mathbb{N}$ be our domain. Then,

$$(\exists x)y < x \tag{1}$$

is true. It says that "for every y, an x exists[25] such that $y < x$". According to 1.1.1.15 there is an *implied* $(\forall y)$ at the beginning of the formula.

---

[25]That is, "for every *value of* y, a *value of* x exists". The mathematician is used to the sloppy language that omits "*value of*". It is clear that he does *not* refer to the variables themselves, but rather refers to their values.

Note that there is *no* single value of x that makes (1) true (because $\mathbb{N}$ has no upper bound). For each value n of y, $n + 1$ is an appropriate value of x (so are $n + 2$, $2n + 1$, etc.)

How can we write down the (false) statement that *one* value of x works for all y?

$$(\exists x)(\forall y)y < x$$

A final remark: How do we write that "there exists a *unique* x such that $\mathscr{A}$ is true" (where $\mathscr{A}$ is any formula)?

$$(\exists x)\Big(\mathscr{A}[x] \wedge \neg(\exists z)(\mathscr{A}[z] \wedge x \neq z)\Big)$$

Fortunately, there is a short form for the above ($\exists!$ reads "for some unique")

$$(\exists! x)\mathscr{A} \qquad\qquad \square$$

1.1.1.18 Example. The reader probably already knows that there is redundancy in the chosen set of connectives, that is, some of them can be simulated by the others.

For example, it is immediate by comparing (4), (1), and (2) in 1.1.1.13, that $\mathscr{A} \rightarrow \mathscr{B}$ is the same as (has the same meaning as) $\neg\mathscr{A} \vee \mathscr{B}$. Similarly, $\mathscr{A} \equiv \mathscr{B}$ is the same as $(\mathscr{A} \rightarrow \mathscr{B}) \wedge (\mathscr{B} \rightarrow \mathscr{A})$.

Even $\mathscr{A} \wedge \mathscr{B}$ can be expressed via $\neg$ and $\vee$ as $\neg(\neg\mathscr{A} \vee \mathscr{B})$. This is easiest to see, perhaps, via a truth-table:

$\mathscr{A}$	$\mathscr{B}$	$\neg$	(	$\neg$	$\mathscr{A}$	$\vee$	$\neg$	$\mathscr{B}$)	$\mathscr{A} \wedge \mathscr{B}$
f	f	(4)f		(1)t		(3)t	(2)t		f
f	t	(4)f		(1)t		(3)t	(2)f		f
t	f	(4)f		(1)f		(3)t	(2)t		f
t	t	(4)t		(1)f		(3)f	(2)f		t

The numbers such as "(1)t" in the first row indicate order of evaluation using the operator at the top of the column. Comparison of the column labeled (4) with the last column shows that each of $\mathscr{A} \wedge \mathscr{B}$ and $\neg(\neg\mathscr{A} \vee \mathscr{B})$ yield the same output for any given value-pair of $\mathscr{A}$ and $\mathscr{B}$. Thus we proved the equivalence of the $\mathscr{A} \wedge \mathscr{B}$ and $\neg(\neg\mathscr{A} \vee \mathscr{B})$. This result is known as "de Morgan's Law". $\qquad \square$

1.1.1.19 Exercise. Prove that $\mathscr{A} \vee \mathscr{B}$ can be expressed as $\neg(\neg\mathscr{A} \wedge \mathscr{B})$. This is the "other" (technically *dual* of) de Morgan's Law. $\qquad \square$

1.1.1.20 Example. We know [(7) of Definition 1.1.1.13] that $(\exists x)\mathscr{A}(z_1, \ldots, z_m, x, y_1, \ldots, y_n)$ means

$$\text{For every choice of } i_1, \ldots, i_m, j_1, \ldots, j_n, \qquad (1.1)$$

$$\text{there is a } k \text{ such that} \qquad (1.2)$$

$$\mathscr{A}(i_1, \ldots, i_m, k, j_1, \ldots, j_n) \qquad (1.3)$$

Now, the *negation* of "there is a k such that $\mathscr{A}(i_1,\ldots,i_m,k,j_1,\ldots,j_n)$" is

$$\text{"no } k \text{ makes } \mathscr{A}(i_1,\ldots,i_m,k,j_1,\ldots,j_n) \text{ true"} \qquad (1)$$

that is,

$$\text{all } k \text{ make } \neg\mathscr{A}(i_1,\ldots,i_m,k,j_1,\ldots,j_n) \text{ true} \qquad (2)$$

(2) says the same thing as

$$(\forall x)\neg\mathscr{A}(i_1,\ldots,i_m,x,j_1,\ldots,j_n) \qquad (3)$$

(1.2)–(1.3), (1), and (3) yield (via 1.1.1.13)

$$(\exists x)\mathscr{A}(i_1,\ldots,i_m,k,j_1,\ldots,j_n) \equiv \neg(\forall x)\neg\mathscr{A}(i_1,\ldots,i_m,x,j_1,\ldots,j_n)$$

By (1.1) and 1.1.1.10, we have

$$(\exists x)\mathscr{A}(z_1,\ldots,z_m,x,y_1,\ldots,y_n) \equiv \neg(\forall x)\neg\mathscr{A}(z_1,\ldots,z_m,x,y_1,\ldots,y_n)$$

for short

$$(\exists x)\mathscr{A} \equiv \neg(\forall x)\neg\mathscr{A} \qquad \qquad \square$$

1.1.1.21 Exercise. Prove that $(\forall x)\mathscr{A} \equiv \neg(\exists x)\neg\mathscr{A}$. $\qquad \square$

1.1.1.22 Remark. We note that $\mathscr{A} \to \mathscr{B}$ is true, in particular, when *no* instance of $\mathscr{A}$ is true, i.e., when $\mathscr{A}$ is *false*—in all its instances, that is. In this case the so-called *classical* or *material implication* holds "vacuously", *even if there is no connection between $\mathscr{A}$ and $\mathscr{B}$ at all and even if $\mathscr{B}$ is not true!* For example, if $\mathscr{A}$ is $0 \neq 0$ and $\mathscr{B}$ is "in every Euclidean triangle the sum of the angles equals 9π", then $(\mathscr{A} \to \mathscr{B})$ is true. The same holds if $\mathscr{B}$ stands for "n is even". The latter has both true and false instances over $\mathbb{N}$, but that is immaterial. In each chosen instance, $\mathscr{A}' \to \mathscr{B}'$ is true—that is (1.1.1.10), $\mathscr{A} \to \mathscr{B}$ is true.

Equally disturbing is the fact that while both sides of the arrow might be true, though *totally unrelated*, the *implication* will be true, as in $\mathscr{C} \to \mathscr{D}$ where $\mathscr{C}$ is $0 = 0$ and $\mathscr{D}$ is "in every Euclidean triangle the sum of the angles equals 2π".

The Intuitionists, a school of thought founded on the writings of Kronecker, Brouwer and Heyting, do not like this state of affairs. The *intended meaning*, or *intended semantics*, for their $\mathscr{A} \to \mathscr{B}$, connects the hypothesis $\mathscr{A}$ strongly to the conclusion $\mathscr{B}$. The meaning, informally speaking, is that, from a *proof* (intuitionistic proof, of course!) of $\mathscr{A}$, a proof for $\mathscr{B}$ can be *constructed*.

We are not going to say what *is* an intuitionistic "proof", as this is of no concern to us. As a matter of fact, "proof" (for classical logic) will be defined only later (see 1.1.1.34). At present, let the reader think of "proof" as a process used to establish that a formula holds. Nevertheless, the above stated *intentions* regarding (intuitionistic) proofs are a clear enough indication of how strongly $\mathscr{A}$ and $\mathscr{B}$ must be related before the Intuitionist will agree to write $\mathscr{A} \to \mathscr{B}$.

In particular, in intuitionistic logic $\rightarrow$ and $\vee$ do not relate as in 1.1.1.18 above. In fact, in both logics $\mathscr{A} \rightarrow \mathscr{A}$ holds, however, in intuitionistic logic $\mathscr{A} \vee \neg\mathscr{A}$ does *not* hold! Or as we say, the *law of the excluded middle* does *not* hold. Intuitively speaking, the reason behind this phenomenon is that the intuitionistic proofs are so structured that, to establish that a formula $\mathscr{A} \vee \mathscr{B}$ holds, in general you need to construct a proof of one of $\mathscr{A}$ or $\mathscr{B}$.

For example, the following classical proof that there are *irrational* numbers[26] a, b such that a^b is rational is *unacceptable intuitionistically*.

Take $a = b = \sqrt{2}$. If this works, done. If not, that means that $\sqrt{2}^{\sqrt{2}}$ is *irrational*. Well, then take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. Now $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$. A rational. End of proof.

Above, $\mathscr{A}$ is "$\sqrt{2}^{\sqrt{2}}$ is rational" and $\mathscr{B}$ is "$\sqrt{2}^{\sqrt{2}}$ is irrational". We used the (classical) fact that $\mathscr{A} \vee \mathscr{B}$ is true, since one or the other of $\mathscr{A}$ and $\mathscr{B}$ is (classically) true. However, classically, we do not need to know which is which!

A thorough exposition of intuitionistic logic can be found in the advanced book of Schütte ([Schü]). □

1.1.1.23 Example. Suppose that x does not occur free in $\mathscr{A}[x]$.

Pause. Ensure that this is consistent with the notation introduced in Definition 1.1.1.11.◄

Then

$$\mathscr{A}[x] \equiv (\forall x)\mathscr{A}[x] \tag{1}$$

Indeed, let $y, z, w, \ldots$ be the complete list of free variables of $\mathscr{A}$, where x is not one of them. To verify (1) we need to show that for any choice of values $k, l, m, \ldots$ from the domain

$$\mathscr{A}(k, l, m, \ldots) \equiv (\forall x)\mathscr{A}[x, k, l, m, \ldots] \tag{2}$$

that is,

$$\text{both sides of (2) are } \mathbf{t} \text{ or both are } \mathbf{f}. \tag{3}$$

We need to analyze $(\forall x)\mathscr{A}[x, k, l, m, \ldots]$. It says

"for all n in the domain, $\mathscr{A}[n, k, l, m, \ldots]$ is true"

But this is true exactly when $\mathscr{A}(k, l, m, \ldots)$ is, since n does not affect the output: the variable x is non-input in $\mathscr{A}$. □

1.1.1.24 Exercise. Suppose we drop the condition "suppose that x does not occur free in $\mathscr{A}[x]$" above. Does (1) still hold? You must provide the "why"! □

---

[26]That is, not rational. A rational number has the form, by definition, p/q where both p and $q \neq 0$ are integers.

1.1.1.25 Exercise. In view of 1.1.1.15, "$\mathscr{A}[x]$ is true" iff "$(\forall x)\mathscr{A}[x]$ is true". So, is this observation not all that we need to assert (1)? See also the previous exercise. $\square$

Atomic formulae contain no Boolean connectives at all. On the other hand, formulae with a *leading* quantifier, $\forall$ or $\exists$, contain no *explicit* Boolean connectives: any Boolean connectives they may have are "hidden" in the quantifier's scope. Thus, one may view these two categories of formulae as "*atomic, but from a Boolean point of view*", meaning you cannot split them into *simpler* ones by simply *removing a Boolean connective*. For example, if you start with $(\forall x)(x = 0 \lor x = y)$ and remove the $\lor$, you get the nonsensical $(\forall x)(x = 0\ x = y)$. Logicians call these "Boolean atomic formulae" *prime* formulae but also *Boolean variables*.

1.1.1.26 Definition. (Prime Formulae; Tautologies) A *prime formula* or *Boolean variable* is either an atomic formula, or a formula with a leading quantifier.

If a formula $\mathscr{F}$—when viewed as a *Boolean combination* of prime formulae, that is, as a formula built from prime formulae using only the formation rules (2)–(6) of 1.1.1.3—evaluates as **t** according to the truth table 1.1.1.14, for *all possible assumed* truth-values of its Boolean variables, then we call it a *tautology* and write this concisely as $\models_{taut} \mathscr{F}$. $\square$

1.1.1.27 Remark. Every formula is built by appropriately applying Boolean glue on a number of prime formulae: Indeed, in any formula built according to 1.1.1.3 we can identify all its *maximal* prime subformulae—that is prime subformulae not contained in larger prime subformulae.

For example, in $(\forall x)(x = 0 \to (\exists y)x = y) \lor w = u \land u = 2^x$ we may indicate the prime subformulae by "boxing" them as below.

$$(1)$$

Double-boundary boxes enclose maximal prime formulae. The Boolean structure of (1) is

Only maximal prime formulae contribute to the Boolean structure and express the original formula as a Boolean combination of prime formulae glued together by connectives. The non-maximal prime formulae are hidden *inside maximal ones.*

The italicized claim above follows directly from an adaptation of the "deconstruction" in 1.1.1.6:

Just replace the step

> Write down, *above* whatever you have written so far, the *ip* of all *pending* formulae (if they have ip); and *remove* the flag "pending" from the latter.

by

> Write down, *above* whatever you have written so far, the *ip* of all *non-prime pending* formulae (if they have ip); and *remove* the flag "pending" from the latter.

Conversely, a Boolean combination of prime formulae is a formula in the sense of 1.1.1.3: Indeed, a Boolean combination is an expression built by applying (2)–(6) in the context of a formula-construction (cf. 1.1.1.6) with starting points prime formulae, rather than atomic formulae. Since a prime formula *is* a formula, the process leads to a formula. See Exercise 1.8.1 for a rigorous proof (that you will supply, equipped with the induction tool). □

The quantifier "(for all possible) *assumed*" in Definition 1.1.1.26 is significant. It means that we *do not* compute the actual (intrinsic) truth-values of the constituent Boolean variables (in the domain that we have in mind)—*even if they do have such a value*; note that $x = y$ does not.

Rather, we *assume* for each prime formula, *all the—in principle—possible output values*; that is, *both* of **t** and **f**.

For example, for the determination of tautologyhood of a formula, where $x = x$ enters as a Boolean variable, we *assume* two possible output values, **t** *and* **f** even though we *know* that its *intrinsic* value is **t**.

In particular, $x = x$ is *not* a tautology.

Pause. Ensure that this last observation fits with 1.1.26!◄

We indicate this fact by writing $\not\models_{taut} x = x$.

Assuming all possible truth-values of a prime formula (rather than attempting to compute "the" value) is tantamount to allowing the *Boolean structure* and Boolean structure alone—that is how the connectives have glued the formula together—to determine the truth-value via truth tables (1.1.1.14).

1.1.1.28 Example. Some tautologies: $x = 0 \rightarrow x = 0$, $(\forall x)\mathscr{A} \vee \neg(\forall x)\mathscr{A}$, $x = y \rightarrow x = y \vee z = 2^{2^w}$.
Some non-tautologies: $x = 0 \rightarrow x = 5$, $(\forall x)x = x \vee (\forall y)y = y$, $x = y \rightarrow x = w \vee z = 2^{2^w}$. For example, looking at the value assumptions below—or value *assignments*, as is the accepted term,
$$x = y := \mathbf{t}$$
$$x = w := \mathbf{f}, \text{ and}$$
$$z = 2^{2^w} := \mathbf{f},$$
we see that $x = y \rightarrow x = w \vee z = 2^{2^w}$ evaluates as **f**, thus it is indeed not a tautology. Incidentally, I used ":=" to denote (truth) value assignment. □

1.1.1.29 Definition. (Tautological Implication) We say that $\mathscr{A}_1, \ldots, \mathscr{A}_n$ *tautologically imply* $\mathscr{B}$ iff $\models_{taut} \mathscr{A}_1 \rightarrow \mathscr{A}_2 \rightarrow \ldots \rightarrow \mathscr{A}_n \rightarrow \mathscr{B}$.

We write $\mathscr{A}_1, \ldots, \mathscr{A}_n \models_{taut} \mathscr{B}$ in this case. $\mathscr{B}$, we say, is (the result of) a tautological implication from $\mathscr{A}_1, \ldots, \mathscr{A}_n$. □

1.1.1.30 Remark. We note that a tautological implication *preserves truth*, from left to right. See exercise below. □

1.1.1.31 Exercise. Let $p_1, \ldots, p_m$ be all the Boolean variables that appear in the formulae $\mathscr{A}_1, \ldots, \mathscr{A}_n, \mathscr{B}$.

Show that $\mathscr{A}_1, \ldots, \mathscr{A}_n \models_{taut} \mathscr{B}$ iff every set of values assigned to the Boolean variables that makes *all* the $\mathscr{A}_i$ **t**, also makes $\mathscr{B}$ **t**. □

1.1.1.32 Example. All of the following are correct: $x = 0 \models_{taut} x = 0, x < y, y < z \models_{taut} x < y \wedge y < z, x = y \models_{taut} \mathscr{A} \vee \neg\mathscr{A}$ (no matter what $\mathscr{A}$ stands for). $\mathscr{A} \models_{taut} \mathscr{B} \rightarrow \mathscr{A}$ is also correct, since $\mathscr{A} \rightarrow \mathscr{B} \rightarrow \mathscr{A}$, that is, $\mathscr{A} \rightarrow (\mathscr{B} \rightarrow \mathscr{A})$ can be readily verified as a tautology using either 4 of 1.1.1.13 or the truth table (1.1.1.14). A shortcut is to just consider the two assumed values, **t** and **f**, for $\mathscr{A}$. The second makes the formula true outright, while the first makes the bracketed subformula **t**, hence the whole formula **t**.

This is incorrect: $x < y, y < z \models_{taut} x < z$ since choosing

$x < y := \mathbf{t}$

$y < z := \mathbf{t}$, and

$x < z := \mathbf{f}$

we see that $x < y \rightarrow y < z \rightarrow x < z$ evaluates as **f**, so it is not a tautology. □

1.1.1.33 Remark. (Capture of a Free Variable) Let $\mathscr{A}(x)$ stand for $(\exists y)y \neq x$—recall that $y \neq x$ is short for $\neg y = x$. It states (i.e., codifies the statement) "for any value of x there is a value of y that is different". Assuming that our domain is $\mathbb{N}$, this is clearly a true statement. So is $\mathscr{A}(z)$, obtained by substituting z for x, or, in programming jargon, "calling" $\mathscr{A}(x)$ with argument z.

What about $\mathscr{A}(y)$? This is $(\exists y)y \neq y$ which is evidently false: "there is a value of y which is different from itself"!

This is unfortunate because, intuitively, what $\mathscr{A}(x)$ says should have nothing to do with the name of the input variable!

What happened here is that when we substituted y for x, the free y was *captured*—i.e., *became bound*—by a lurking quantifier, $(\exists y)$: y got into the latter's scope.

We should never allow such substitutions since, as this example shows, they may change the intended meaning of the resulting formula. In general, a substitution into $\mathscr{F}[x]$ that results into $\mathscr{F}[t]$ should not *be allowed, if the term t contains a free variable y that will become bound (captured) after the substitution.*

Is there a workaround? Yes!

Consider an instance $(\exists x)\mathscr{F}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$ of

$$(\exists x)\mathscr{F}(z_1, \ldots, z_m, x, w_1, \ldots, w_n) \tag{1}$$

and a *consistent* instance (cf. 1.1.1.11) $(\exists u)\mathscr{F}(i_1, \ldots, i_m, u, j_1, \ldots, j_n)$ of

$$(\exists u)\mathscr{F}(z_1, \ldots, z_m, u, w_1, \ldots, w_n) \tag{2}$$

where u is a new variable. The two instances are equivalent, since if $x = k$ works for the first, then $u = k$ works for the second, and vice versa. The instance being arbitrary, we get the equivalence of (1) and (2), that is

$$(\exists x)\mathscr{F}(z_1, \ldots, z_m, x, w_1, \ldots, w_n) \equiv (\exists u)\mathscr{F}(z_1, \ldots, z_m, u, w_1, \ldots, w_n)$$

In view of 1.1.1.20 and 1.1.1.21, or directly using a similar argument as above, one sees at once that changing the bound variable of a universal quantifier into a brand new variable does not change the meaning either.

This leads to the so-called "*variant theorem*" of logic, that

Changing a bound variable in a formula into a new *(i.e., not already used in the formula) variable does not change the meaning: the original and the new formulae are equivalent.*

But then, given $\mathscr{A}[x]$, we can *always* effect $\mathscr{A}[t]$ with impunity as long as we *rename* out of harm's way, *before the substitution takes place*, all the original bound variables:[27] All we need to do in a successful renaming is to ensure that none of them is the same as a free variable of t. Strictly speaking, in $\mathscr{A}[t]$ we do not have the original formula $\mathscr{A}$, but a *variant* of the original—since we have renamed the latter's bound variables. Nevertheless since the old and the new are equivalent, we will use the same name for both, $\mathscr{A}$. □

We now turn to what a mathematician or computer scientist does with logic: He writes proofs.

1.1.1.34 Definition. (Proofs) A *proof* is a *construction process* that builds a *finite length sequence* of true formulae, *one at a time*. We normally write this sequence vertically on the page, with explanatory annotation. Three simple rules regarding what formula we *may* write at each (construction-) step govern the process. We may write

(1) A formula that we *know* as, or *accept* as, true.

(2) A formula that is a tautological implication of formulae already written in the course of the proof.

(3) $(\forall x)\mathscr{A}(\ldots, x, \ldots)$ provided $\mathscr{A}(\ldots, x, \ldots)$ has already been written in the course of the proof.

Any formula $\mathscr{A}$ that appears in a proof we call a *theorem*. We say that the proof "established" or *proved* the theorem $\mathscr{A}$. □

A theorem follows from certain axioms Γ. Saying just "theorem" does not indicate this dependence, unless what is the relevant set of axioms is clearly understood from the context. If in doubt, or if we are discussing theorems of various theories

---

[27]This is a sufficient and straightforward overkill. In principle, we only *need* to rename those bound variables that are referenced in those quantifiers that will capture a variable in t, if we do nothing.

simultaneously, then we must name the applicable axiom set in each case by saying "Γ-theorem" or "theorem from Γ".

It is clear from what we have developed so far, that steps (2) and (3) *preserve truth* (cf. 1.1.1.15). An application of step (3) is called application of *generalization*, or rule *Gen*, as we will say.

What exactly is going on in step (1)? Well, we know that some formulae are true because we recognize them as such outright, without the help of any complicated process; they are "initial" truths—or *initial theorems*—such as $x = x$, which is true in *all mathematics*, or $x + 1 \neq 0$, which is *true in a specific theory*: arithmetic of the natural numbers. These initial theorems, whether they are universal or theory-specific, are called *axioms*.

1.1.1.35 Remark. All axioms are "atomic theorems", that is, they are obtained by an application of (1)—they are not "results" of the application of (2) or (3) on previous theorems written in the course of a proof.

As indicated in the ☡-passage above, we have two types of axioms.

(a) Those that are true because of the way the formulae that express them are put together, using connectives and quantifiers. These axioms are *not* specific to any branch of mathematics: *They hold for all mathematics.*

We call such axioms *logical.*[28]

With some ingenuity, a very small set of formulae[29] can be chosen, among the *universally* or *absolutely true* formulae, to serve as logical axioms. Read on!

For example, $x = x$ and $(\forall x)\mathscr{A}[x] \to \mathscr{A}[t]$ are such universal truths, and we will adopt both as logical axioms.

(b) A formula $\mathscr{A}$ is a *nonlogical* axiom in a *mathematical theory* provided it is taken as an important *start-up truth of the theory*—an "atomic theorem"—*not* because of its form, but rather because of *what it says* in connection with the various symbols that are peculiar to the theory.

For example, $x + 1 \neq 0$ is an important start-up truth—a nonlogical axiom—of (Peano) arithmetic over $\mathbb{N}$. There is nothing to render it "universal"; in fact, it is not a true statement if our domain is either the reals, $\mathbb{R}$, or the integers, $\mathbb{Z}$ (all of positive, negative and zero). Another nonlogical axiom, for Euclidean geometry, is "Euclid's 5th postulate", which asserts that *through a point outside a line we can draw precisely one parallel to said line.* Again, the existence of so-called non-Euclidean geometries shows that this is not a universal truth.

When we use the term "theory Γ", we mean somewhat ambiguously, but equivalently, either

---

[28] They express universal truths of *logic*, that is.

[29] Strictly speaking, *formula-forms* or *formula-schemata*, since these formulae will contain, as subformulae, formula-names of unspecified formulae (such as $\mathscr{A}$), arbitrary (object) variables, function names, etc.

- Γ is the set *all its theorems*, that is, the set of all formulae that can be proved starting from the axioms of the theory repeatedly applying the proof-tools (1)–(3) above.

Or

- Γ is identified with the set of *all the postulated nonlogical axioms*. Clearly, if we have the nonlogical axioms, then using the logical axioms that are common to all theories, along with the proof-process (1)–(3) above, we may write down, in principle,[30] all the theorems of the theory.

We prefer the viewpoint of the second bullet, as it gives prominence to our start-up assumptions; the nonlogical axioms.

The terminology "$\mathscr{F}$ *is true in the theory*" simply means that $\mathscr{F}$ is a *theorem of the theory*. Its truth is *relative to the truth of the nonlogical axioms*; it is not absolute or universal. For example, "the sum of the angles of any triangle equals 180°" is true in (is a theorem of) Euclidean geometry. It is not true in (is not a theorem of) either Riemann's or Lobachevski's geometries.

That $\mathscr{F}$ is true in a theory Σ will be denoted symbolically as $\Sigma \vdash \mathscr{F}$.

Note that the logical axioms are not mentioned at the left of "$\vdash$". Thus, if Σ is empty and we have proved $\mathscr{F}$ *only using logical axioms*, then we will write $\vdash \mathscr{F}$.

It is immediate from the foregoing that since a proof is not obliged, in an applications of step (1) (1.1.1.34), to use any nonlogical axiom, that every theory also contains among its theorems all the absolute truths $\mathscr{F}$ for which $\vdash \mathscr{F}$.[31]

It is clear that $\vdash$ is *transitive*, that is, if we have $\Sigma \vdash \mathscr{A}_i$ for $i = 1, \ldots, n$, and also $A_1, \ldots, A_n \vdash \mathscr{B}$, then $\Sigma \vdash \mathscr{B}$. Indeed, we can clearly concatenate the proofs of each A_i—in any order—and then append the proof of $\mathscr{B}$ at the very end. What we get is a proof of $\mathscr{B}$ as required.

Since at each step of writing a proof we look back rather than forward [steps (2) and (3)], it is clear that chopping off the "tail" of a proof at any point leaves us with a proof. This observation entails that when we attempt to prove a formula from given axioms, we just stop as soon as we have written the formula down.   □

1.1.1.36 Exercise. Elaborate on the remark above regarding the transitivity of $\vdash$.   □

1.1.1.37 Exercise. In mathematical practice we are allowed to use in a proof any already proved theorems, in a step of type (1) of 1.1.1.34. Carefully justify this practice in the context of Definition 1.1.1.34.   □

Since rules (2) and Gen on 1.1.1.34 preserve truth, and the logical axioms are universally true, then so is any $\mathscr{F}$ for which we have $\vdash \mathscr{F}$. Logicians call the content of this observation *soundness*.

---

[30]"In principle": The set of theorems of an interesting theory such as arithmetic, Euclidean geometry, or set theory is infinite.
[31]By Gödel's completeness theorem, Gödel (1930), the hedging "for which $\vdash \mathscr{F}$" is redundant.

1.1.1.38 Definition. (Logical Axioms) The usually adopted logical axioms are (forall choices of formulae, variables and terms that appear in them):

(i) All tautologies

(ii) $(\forall x)\mathscr{A}[x] \to \mathscr{A}[t]$ (see conclusion of 1.1.1.33)

(iii) $\mathscr{A}[x] \to (\forall x)\mathscr{A}[x]$, provided x is not free in $\mathscr{A}[x]$

(iv) $(\forall x)(\mathscr{A}[x] \to \mathscr{B}[x]) \to (\forall x)\mathscr{A}[x] \to (\forall x)\mathscr{B}[x]$

(v) $x = x$

(vi) $t = s \to (\mathscr{A}[t] \equiv \mathscr{A}[s])$ (see conclusion of 1.1.1.33). $\qquad\square$

That the above logical axioms are *adequate* to prove *all* universal truths using the proof mechanism of 1.1.1.34 is a result of Gödel's [completeness theorem; Gödel (1930)].

1.1.1.39 Remark. It is easy to verify that all the logical axioms are indeed universal truths. For group (i) and (v) this is trivial. The "truth" expressed (codified) in group (ii) is that "if $\mathscr{A}[x]$ is true *for all objects in its domain*, then it must be true if we take x to be a specific object t". Note that even if t has input variables, then as they vary over the domain they generate objects from the domain. The generated objects are part of "*all objects in its domain*".

The truth of all formulae in group (iii) follows from a trivial modification of the argument in 1.1.1.23.

Group (vi) is Leibniz's characterization of equality: It states that replacing "equals by equals" in an argument slot (x of of $\mathscr{A}[x]$ in our case) produces the same result.

Finally, let us look at group (iv). For simplicity we assume that x is the only variable, so we will show the truth of $(\forall x)(\mathscr{A}(x) \to \mathscr{B}(x)) \to (\forall x)\mathscr{A}(x) \to (\forall x)\mathscr{B}(x)$ in its domain. First off, this means

$$(\forall x)(\mathscr{A}(x) \to \mathscr{B}(x)) \to \Big((\forall x)\mathscr{A}(x) \to (\forall x)\mathscr{B}(x)\Big) \tag{1}$$

Since (1) is an implication, 1.1.1.14 indicates that the only real work for us is if $(\forall x)(\mathscr{A}(x) \to \mathscr{B}(x))$ evaluates as **t**. If this is so, this means

$$\text{For every } k \text{ in the domain, } \mathscr{A}(k) \to \mathscr{B}(k) \text{ is } \mathbf{t} \tag{2}$$

We now try to prove that the right hand side of the leftmost $\to$ must evaluate as **t**. As it too is an implication, we will only consider the real work case where $(\forall x)\mathscr{A}(x)$ is true, that is

$$\text{For every } k \text{ in the domain, } \mathscr{A}(k) \text{ is } \mathbf{t} \tag{3}$$

and try to obtain that

$$(\forall x)\mathscr{B}(x) \tag{4}$$

is true. Now, via 1.1.1.14, (2) and (3) give "For every k in the domain, $\mathscr{B}(k)$ is $\mathbf{t}$", which establishes (4). □

1.1.1.40 Example. Here are some proofs written in extreme pedantry.
  (I) Establish the (universal) truth of $\mathscr{A}[t] \rightarrow (\exists x)\mathscr{A}[x]$. The proof follows:

(1) $(\forall x)\neg\mathscr{A}[x] \rightarrow \neg\mathscr{A}[t]$ ⟨axiom (ii)⟩

(2) $\mathscr{A}[t] \rightarrow \neg(\forall x)\neg\mathscr{A}[x]$ ⟨(1) and rule (2) of proof-writing; 1.1.1.34⟩

(3) $\mathscr{A}[t] \rightarrow (\exists x)\mathscr{A}[x]$ ⟨(2) and rule (2) of 1.1.1.34, using 1.1.1.20⟩

The comments in ⟨...⟩-brackets explain why we wrote the formula to their left. The numbering to the left allows easy reference to previous formulae. The last step is "replacing equivalents by equivalents" in a Boolean combination. The result stays the same since it is as if we "called" $\mathscr{A} \rightarrow \mathscr{X}$ with inputs, first $\neg(\forall x)\neg\mathscr{A}[x]$ and then $(\exists x)\mathscr{A}[x]$. But these inputs have the same (truth) value, so both calls will return the same answer. Since step (2) has written a truth (why?), so has step (2).
  (II) Establish that $t = t$ for any term t. The proof follows:

(1) $x = x$ ⟨axiom (v)⟩

(2) $(\forall x)x = x$ ⟨(1) + Gen⟩

(3) $(\forall x)x = x \rightarrow t = t$ ⟨axiom (ii)⟩

(4) $t = t$ ⟨(2, 3) + tautological implication⟩ □

1.1.1.41 Remark. In the second proof above we used two important tools explicitly. We identify both here so they can be used "off the shelf" in diverse situations in the future.
  The step from (2) to (4) via (3) generalizes to the rule "from (the truth of) $(\forall x)\mathscr{A}[x]$ follows (the truth of) $\mathscr{A}[t]$". This rule is called *specialization* or *Spec*. It follows from an application of this tautological implication, $\mathscr{F}, \mathscr{F} \rightarrow \mathscr{G} \models_{taut} \mathscr{G}$ known as *modus ponens*, for short MP. The reader can easily verify that indeed MP is a tautological implication, so it qualifies as a proof-step of type (2) (1.1.1.34). □

1.1.1.42 Exercise. Give a proof that from the truth of $\mathscr{A}[x]$ follows the truth of $\mathscr{A}[t]$. □

1.1.1.43 Example. We verify that with an assumption (nonlogical!) of the form $\mathscr{A} \rightarrow \mathscr{B}$ we can prove $\mathscr{A} \rightarrow (\forall x)\mathscr{B}$, on the proviso that x is not free in $\mathscr{A}$. That is, we verify, under the stated condition, that $\mathscr{A} \rightarrow \mathscr{B} \vdash \mathscr{A} \rightarrow (\forall x)\mathscr{B}$.

(1) $\mathscr{A} \rightarrow \mathscr{B}$ ⟨hyp⟩

(2) $(\forall x)(\mathscr{A} \rightarrow \mathscr{B})$ ⟨(1) + Gen⟩

(3) $(\forall x)(\mathscr{A} \rightarrow \mathscr{B}) \rightarrow (\forall x)\mathscr{A} \rightarrow (\forall x)\mathscr{B}$ ⟨axiom (iv)⟩

(4) $(\forall x)\mathscr{A} \rightarrow (\forall x)\mathscr{B}$ ⟨(2, 3) + MP⟩

$$(5) \quad \mathscr{A} \to (\forall x)\mathscr{B} \qquad\qquad \langle (*) \rangle$$

In step (1) we said "hyp". The formula is a "hypothesis"—a starting point; not something that we claim as true; a nonlogical axiom. In step (5) I wrote (*) so that I can explain the reasons for the step outside the proof, since the explanation is long, namely: (5) follows from (4) by replacing "equivalents for equivalents"—see 1.1.1.23 and the similar situation in 1.1.1.40. □

1.1.1.44 Exercise. Prove that $\mathscr{A} \to \mathscr{B} \vdash \neg\mathscr{B} \to \neg\mathscr{A}$. The two sides of $\vdash$ are called *contrapositives* of each other. □

1.1.1.45 Exercise. Prove that $\mathscr{A} \to \mathscr{B} \vdash (\exists).\mathscr{A} \to \mathscr{B}$ as long as x is not free in $\mathscr{B}$.
  Hint. Rely on 1.1.1.44 and use 1.1.1.43. □

1.1.1.46 Example. Let us establish the familiar commutativity property of equality as a result of the logical axioms [in particular, of (v) and (vi); cf. p. 21].

  Let $\mathscr{A}[z]$ stand for $z = x$. An instance of Leibniz's axiom is $x = y \to (\mathscr{A}[x] \equiv \mathscr{A}[y])$ i.e.,

$$x = y \to (x = x \equiv y = x) \qquad\qquad (1)$$

We can now embark on a proof:

$$
\begin{array}{lll}
(a) & x = y \to (x = x \equiv y = x) & \langle \text{logical axiom (1)} \rangle \\
(b) & x = x \to x = y \to y = x & \langle \text{tautological implication of (a)} \rangle \\
(c) & x = x & \langle \text{logical axiom} \rangle \\
(d) & x = y \to y = x & \langle (\text{b, c}) + \text{MP} \rangle
\end{array}
$$

Step (b) takes some doing, but is easy. Recall 1.1.1.29 and 1.1.1.31. We need to argue that if line (a) is **t**, then this forces line (b)

$$x = x \to (x = y \to y = x) \qquad\qquad (2)$$

to be **t** as well. By the way, the Boolean variables here are $x = x$, $x = y$ and $y = x$.

  Well, the real work toward seeing that (2) is **t** is when $x = x$ and $x = y$ are **t**. If so, the assumption that line (a) is true forces $x = x \equiv y = x$ to be true (because the part to the left of $\to$ in said line is). Since $x = x$ is assumed **t**,[32] then so must $y = x$, which establishes (2).

  Since the above proof contains no nonlogical axioms, we may write $\vdash x = y \to y = x$.

  The reader will note that this is not a tautology, since $x = y$ and $y = x$ are distinct Boolean variables. □

---

[32]The word "assumed" was inserted for emphasis: $x = x$ in this argument is a Boolean variable. We are not looking for its intrinsic value, rather we are taking turns to consider each of its "possible" values, **f** and **t**. The argument skipped the first value because it trivially makes (2) true.

1.1.1.47 Exercise. Armed with the commutativity of =, prove the transitivity of =. That is, establish the claim $\vdash x = y \rightarrow y = z \rightarrow x = z$. □

1.1.1.48 Example. There are a couple of trivial, but often-used proof tools. They are expressed in the form $\mathscr{X}, \mathscr{Y}, \ldots \vdash \mathscr{A}$, that is, "if I know that $\mathscr{X}, \mathscr{Y}, \ldots$ hold—either because they are assumptions (e.g., could be nonlogical axioms) or are already proved theorems—then I can prove that $\mathscr{A}$ holds as well".

(a) **Proof by cases.** $\mathscr{A} \rightarrow \mathscr{B}, \mathscr{C} \rightarrow \mathscr{B} \vdash \mathscr{A} \vee \mathscr{C} \rightarrow \mathscr{B}$.

It states that to prove that $\mathscr{B}$ follows from a disjunction, it *suffices* to prove separately that $\mathscr{B}$ follows from each *case*—$\mathscr{A}$ and $\mathscr{C}$—of the disjunction.

(b) **Ping pong.** $\mathscr{A} \rightarrow \mathscr{B}, \mathscr{B} \rightarrow \mathscr{A} \vdash \mathscr{A} \equiv \mathscr{B}$. It states that to prove an equivalence, $\mathscr{A} \equiv \mathscr{B}$, it *suffices* to prove *each direction*—$\mathscr{A} \rightarrow \mathscr{B}$ and $\mathscr{B} \rightarrow \mathscr{A}$—separately, since, from the two directions taken as hypotheses jointly, we can prove the equivalence.

Each of (a) and (b) admit immediate proofs: Once we have assumed the hypotheses on each side, a tautological implication yields the conclusion at once [cf. 1.1.1.34, rule (2) applied]. □

A major proof tool of the mathematician and computer scientist is the so-called *deduction theorem*. It is stated without proof here—for us it is its statement that matters.

See Tourlakis (2003a) or Tourlakis (2008) for a proof of the deduction theorem, but be warned that the two versions in these references are different, because the specific *foundations* of logic in these two are different! The difference lies in how generalization is applied, and that affects the statement, and proof, of the deduction theorem. The present volume uses the style of generalization as it is practiced in the first cited reference.

1.1.1.49 Theorem. (Deduction Theorem) *If we can prove $\mathscr{B}$ from assumptions Γ and $\mathscr{A}$, then we can prove $\mathscr{A} \rightarrow \mathscr{B}$ from the assumptions Γ alone,* on a condition. *The condition is that during the proof of $\mathscr{B}$ from hypotheses Γ and $\mathscr{A}$, step (3) of 1.1.1.34 was never applied with a variable that occurs free in $\mathscr{A}$.*

In other words, the free variables of $\mathscr{A}$ during said proof are "frozen"; they behave like constants.

We can say the above symbolically as "if $\Gamma, \mathscr{A} \vdash \mathscr{B}$, then $\Gamma \vdash \mathscr{A} \rightarrow \mathscr{B}$, on a condition, etc."

What is the deduction theorem good for? Well, for one thing, it tells us that instead of proving $\mathscr{A} \rightarrow \mathscr{B}$ it suffices to prove the less complex—since the glue $\rightarrow$ and the part $\mathscr{A}$ are removed—$\mathscr{B}$. For another, we have the added bonus of "knowing more" before starting the proof. While toward proving $\mathscr{A} \rightarrow \mathscr{B}$ we "knew" Γ, toward proving $\mathscr{B}$ we also know $\mathscr{A}$.

Is the restriction on $\mathscr{A}$ too limiting? Not really. In practice, say, we want to prove that $\Gamma \vdash \mathscr{A}(x,y) \to \mathscr{B}(z,y,w)$. We proceed as follows:

> *Fix all the variables* of $\mathscr{A}$, here x and y, to some unspecified values. *Remember not to use either x or y in a rule Gen during the proof!*
>
> Assume now $\mathscr{A}(x,y)$. Proceed to prove $\mathscr{B}(z,y,w)$—recall that y is a "constant".
>
> By the deduction theorem, we have proved $\mathscr{A}(x,y) \to \mathscr{B}(z,y,w)$ from just Γ, because we never performed generalization with the frozen x and y.

In practice we apply less pedantry in the process:

(1) We only say, "let us fix the values all free variables $x, y, \ldots$ in $\mathscr{A}$".

We are, of course, obliged to remember that *this also fixes these same variables in $\mathscr{B}$, and everywhere else, throughout the proof.* Thus, we *cannot* universally quantify any of these variables during the proof, nor can we (a subsidiary operation this; cf. 1.1.1.42) substitute a term into such "frozen" variables.

(2) The task ends as soon as we prove $\mathscr{B}$.

We do *not* need to repeat this kind of justification every time: "By the deduction theorem, we have proved $\mathscr{A} \to \mathscr{B}$ from just Γ, etc."

We will see this proof technique applied many times in this book, starting from the next subsection.

We conclude with the ancient technique of *proof by contradiction*.[33] We will define a *contradiction* to be a formula of the form $\mathscr{A} \wedge \neg\mathscr{A}$. From the truth tables we know that this evaluates as **f** regardless of whether $\mathscr{A}$ itself evaluates a **t** or **f**. The reader can verify at once, that for any $\mathscr{F}$, $\mathscr{A} \wedge \neg\mathscr{A} \models_{taut} \mathscr{F}$.

1.1.1.50 Theorem. (Proof by Contradiction) *For any closed $\mathscr{A}$, we have $\Gamma \vdash \mathscr{A}$ iff $\Gamma, \neg\mathscr{A} \vdash \mathscr{X} \wedge \neg\mathscr{X}$, for some $\mathscr{X}$.*

Proof. Indeed, for the *if*-part, let $\Gamma, \neg\mathscr{A} \vdash \mathscr{X} \wedge \neg\mathscr{X}$. By the deduction theorem (applicable without hedging as $\mathscr{A}$ is closed) we get

$$\Gamma \vdash \neg\mathscr{A} \to \mathscr{X} \wedge \neg\mathscr{X} \tag{1}$$

It is straightforward to see that $\neg\mathscr{A} \to \mathscr{X} \wedge \neg\mathscr{X} \models_{taut} \mathscr{A}$, hence by transitivity of $\vdash$, we get $\Gamma \vdash \mathscr{A}$.

only if-part. Say, $\Gamma \vdash \mathscr{A}$. Adding to the assumptions Γ we can still prove $\mathscr{A}$ (see that you agree! 1.1.1.34). Thus

$$\Gamma, \neg\mathscr{A} \vdash \mathscr{A} \tag{2}$$

---

[33]Often used by Euclid, for example.

But also (why?)

$$\Gamma, \neg\mathscr{A} \vdash \neg\mathscr{A} \tag{3}$$

The trivial $\mathscr{A}, \neg\mathscr{A} \models_{taut} \mathscr{A} \wedge \neg\mathscr{A}$ along with (2) and (3) above, and transitivity of $\vdash$, yield $\Gamma, \neg\mathscr{A} \vdash \mathscr{A} \wedge \neg\mathscr{A}$. □

The technique is used as follows: To prove $\Gamma \vdash \mathscr{A}$ (closed $\mathscr{A}$) we start by "Assume, by way of contradiction, $\neg\mathscr{A}$" and then proceed to indeed obtain one.

1.1.1.51 Definition. A mathematical theory, given by its set of nonlogical axioms, is *consistent* or *free from contradiction*, provided it is impossible to prove a contradiction from its axioms. Otherwise it is called inconsistent. □

> Thus we can rephrase 1.1.1.50 as $\Gamma \vdash \mathscr{A}$ iff (the theory with axioms) $\Gamma, \neg\mathscr{A}$ is inconsistent.

1.1.1.52 Remark. ("Everyday" Proof Style) A few important remarks are in order to conclude our digression into logic.

(1) The proof of truth of a formula using *first principles* from 1.1.1.13 and working *directly* with a reference set is now for us a thing of the past. The last time we utilized the method was to establish that all the logical axioms were indeed universal truths (1.1.1.39—see also 1.1.1.33). From then on we will ride on the shoulders of our *logical axioms* 1.1.1.38, and whatever other assumptions *we take as true* from time to time, to prove all our theorems, essentially "syntactically", that is, by writing proofs according to 1.1.1.34.

(2) The practicing mathematician or computer scientist uses a simplified, often shorter, and rather conversational version of the *proofs with annotation* that we presented so far (for example, in 1.1.1.43 and 1.1.1.46). We should get used to this relaxed style. Here is an example. We will establish that

$$\vdash (\forall x)(\mathscr{A} \wedge \mathscr{B}) \equiv (\forall x)\mathscr{A} \wedge (\forall x)\mathscr{B}$$

Proof. We employ ping pong. ($\rightarrow$) direction: Assume $(\forall x)(\mathscr{A} \wedge \mathscr{B})$ with all its free variables frozen (we are going via the deduction theorem). Remove the quantifier (Spec) to obtain $\mathscr{A} \wedge \mathscr{B}$ and apply two tautological implications to obtain $\mathscr{A}$ and $\mathscr{B}$. Apply Gen to each to get $(\forall x)\mathscr{A}$ and $(\forall x)\mathscr{B}$. A tautological implication yields what we want.

For the ($\leftarrow$) direction, assume $(\forall x)\mathscr{A} \wedge (\forall x)\mathscr{B}$, freezing all free variables of the formula. Two tautological implications yield $(\forall x)\mathscr{A}$ and $(\forall x)\mathscr{B}$. Two applications of Spec, followed by tautological implication yield $\mathscr{A} \wedge \mathscr{B}$. Via Gen we get what we want.

(3) We finally establish that $\mathscr{A} \rightarrow \mathscr{B} \vdash (\forall x)\mathscr{A} \rightarrow (\forall x)\mathscr{B}$. Indeed, the hypothesis yields $(\forall x)(\mathscr{A} \rightarrow \mathscr{B})$ by Gen. We are done via axiom (iv) (1.1.1.38) and modus ponens. □

1.1.1.53 Exercise. Establish the fact $\mathscr{A} \equiv \mathscr{B} \vdash (\forall x)\mathscr{A} \equiv (\forall x)\mathscr{B}$.
Hint. Use 1.1.1.52.   □

1.1.1.54 Exercise. Establish the fact $\mathscr{A} \equiv \mathscr{B} \vdash (\exists x)\mathscr{A} \equiv (\exists x)\mathscr{B}$.
Hint. Use 1.1.1.53.   □

1.1.1.55 Exercise. Establish the fact $\vdash (\exists x)(\mathscr{A} \vee \mathscr{B}) \equiv (\exists x)\mathscr{A} \vee (\exists x)\mathscr{B}$.
Hint. Use 1.1.1.52.   □

1.1.2   Sets and their Operations

We now return to our brief study of sets. The naïve set theory of Cantor is not axiomatic. In our very brief and elementary review of it we will deviate only slightly and adopt precisely one axiom. First off, consider the formulae $x \in A$ and $x \in B$. By 1.1.1.38(vi), we obtain[34]

$$A = B \rightarrow (x \in A \equiv x \in B)$$

and further, via 1.1.1.43 we get (since x, A, B are distinct variables)

$$A = B \rightarrow (\forall x)(x \in A \equiv x \in B) \tag{1}$$

Suppose next that A and B stand for sets—*no such restrictive assumption was made above.* Then (1) says that if two sets are equal, then every member of one (x) is a member of the other, and vice versa; they have precisely the same elements. Is the converse[35] true?

That it is so is a fundamental property of sets; a nonlogical axiom. It is the so-called *axiom of extensionality*.

For any sets A and B, $(\forall x)(x \in A \equiv x \in B) \rightarrow A = B$ (Ext)

Extensionality says that the extension—what sets contain— is what matters to determine their equality. In particular, "structure" does not matter. Nor does "intention": e.g., whether we say outright, "collect 1 and 2 into a set", or, in a roundabout way, "collect the roots of the equation $x^2 - 3x + 2 = 0$ into a set" we get the same set.

Taking (1) and (Ext) together (ping pong) we have [cf. (b) in 1.1.1.48]:

For any sets A and B, $A = B \equiv (\forall x)(x \in A \equiv x \in B)$ (2)

Since $x \in 2$ and $x \in 3$ are false, as 2 and 3 are numbers and thus contain no elements,

$$(\forall x)(x \in 2 \equiv x \in 3) \rightarrow 2 = 3$$

is false, since to the left of $\rightarrow$ we have a true formula, while to the right a false one. Thus, the restriction on the *type* of A and B in (Ext) and (2) is essential. Of course, (1) is *valid for any type of variables* A, B, x.

---

[34]The mathematician and computer scientist will rather say "we obtain $\mathscr{X}$" to indicate he proved so, without using the provability symbol $\vdash$. He will also let the context fend for itself as to what the assumptions were; here no nonlogical assumptions were made.
[35]The converse of the implication $\mathscr{A} \rightarrow \mathscr{B}$ is $\mathscr{B} \rightarrow \mathscr{A}$.

1.1.2.1 Example. Let us introduce the notation-by-listing of sets, $\{\ldots\}$, where the "..." is in each case replaced by an explicit listing of all the set members. Here are two examples: $\{1, 2, 1, 2, 2\}$ and $\{2, 1\}$. These two sets are equal by inspection, according to extensionality, for every element of one appears in the other, and vice versa. In particular, *neither multiplicity, nor order of listing matter.* Only the presence of an element matters, not where (in the listing), or how many times. So, we may write $\{1, 2, 1, 2, 2\} = \{2, 1\}$ or indeed $\{2, 1\} = \{1, 2, 1, 2, 2\}$.

It should be clear that only the (intuitively) *finite sets* (a concept we will soon mathematically define) can be depicted by listing; and this only in principle, since we may not want to list a set of one trillion elements. By the way, writing $\mathbb{N} = \{0, 1, 2, \ldots\}$ is not a by-listing depiction of $\mathbb{N}$, rather, it is a sloppy and *abused* notation (yet surprisingly common). The "..." indicates an unending, and understood from the *context, process* whereby the next element is *generated* by adding one to the previous. The notation taken out of context is nonsensical and gives no clue as to what "..." means. $\square$

The notation $A \subseteq B$, read "A is a *subset* of B", or "B is a *superset* of A", means that every member of A is in B as well. So it is given by the mathematical definition below:

$$A \subseteq B \overset{\text{Def}}{\equiv} (\forall x)(x \in A \rightarrow x \in B) \tag{3}$$

So how does one prove, given some sets A and B that $A \subseteq B$? One uses definition (3) above, and proves instead $(\forall x)(x \in A \rightarrow x \in B)$. But to so prove, it suffices to prove instead $x \in A \rightarrow x \in B$, since an application of Gen to this formula produces the preceding one.

Perfect! One can then proceed as follows: "Fix x and assume $x \in A$". All that one needs to do next is to prove $x \in B$ (1.1.1.49).

At the intuitive level, and from the "word description of equality and subset relations", we expect, for sets A and B, that if $A = B$, then also $A \subseteq B$ (and by symmetry, also $B \subseteq A$). This can be mathematically proved as well: The assumption means $(\forall x)(x \in A \equiv x \in B)$. Dropping $(\forall x)$ (Spec) and following up with a tautological implication we get $x \in A \rightarrow x \in B$. Reintroducing $(\forall x)$ (Gen) we get $A \subseteq B$ (Definition (3)).

Intuitively, for any to sets A and B, if we know that $A \subseteq B$ and $B \subseteq A$, then $A = B$ (the vice versa was the content of the preceding paragraph). Indeed, the two assumptions and (3) above expand to $(\forall x)(x \in A \rightarrow x \in B)$ and $(\forall x)(x \in B \rightarrow x \in A)$, respectively. Dropping $(\forall x)$ (Spec) we obtain $x \in A \rightarrow x \in B$ and $x \in B \rightarrow x \in A$. Following up with a tautological implication we get $x \in A \equiv x \in B$. Applying $(\forall x)$ we get $A = B$.

So, in practice, to prove set equality, $A = B$, we go about it like this: "($\subseteq$) direction: Fix $x \in A$ ... therefore, $x \in B$ is proved". Then we do: "($\supseteq$) direction: Fix $x \in B$ ... therefore, $x \in A$ is proved".

If $A \subseteq B$ but $A \neq B$ we say that "A is a *proper subset* of B" and write $A \subset B$. As is usual in mathematics, negating a relation is informally denoted by the relation

symbol superimposed with a "/". So $A \notin B$, $A \nsubseteq B$ and $A \not\subset B$ mean $\neg A \in B$, $\neg A \subseteq B$ and $\neg A \subset B$, respectively.

1.1.2.2 Definition. (Bounded Quantification) In much of what we do in this volume we will find *bounded quantifiers* very useful. That is, in set theory we will often want to says things like "for all x in A, $\mathscr{X}[x]$ holds". While this can be captured by $(\forall x)(x \in A \to \mathscr{X}[x])$, the alternative shorthand is much in use, and preferable: $(\forall x \in A)\mathscr{X}[x]$ or also $(\forall x)_{\in A}\mathscr{X}[x]$.

In arithmetic we will correspondingly often want to say "for all $x < y$, $\mathscr{X}[x]$ holds". This is coded directly as $(\forall x)(x < y \to \mathscr{X}[x])$. The preferred shorthand is: $(\forall x < y)\mathscr{X}[x]$ or also $(\forall x)_{<y}\mathscr{X}[x]$. In these two cases, respectively, A and y are free variables. $\square$

1.1.2.3 Remark. The corresponding "for some x in A, $\mathscr{X}[x]$ holds" and "for some $x < y$, $\mathscr{X}[x]$ holds" have the shorthands $(\exists x \in A)\mathscr{X}[x]$ or $(\exists x)_{\in A}\mathscr{X}[x]$ for the former and $(\exists x < y)\mathscr{X}[x]$ or also $(\exists x)_{<y}\mathscr{X}[x]$ for the latter.

The shorthand $(\exists x \in A)\mathscr{X}[x]$ and $(\exists x < y)\mathscr{X}[x]$ stand for $(\exists x)(x \in A \wedge \mathscr{X}[x])$ and $(\exists x)(x < y \wedge \mathscr{X}[x])$, respectively.

Translating to $\forall$ notation we do not get any nasty surprises. For example,

$$(\exists x)(x \in A \wedge \mathscr{X}[x]) \qquad (*)$$

is equivalent to $\neg(\forall x)\neg(x \in A \wedge \mathscr{X}[x])$. Using 1.1.1.53 and an obvious tautology, we see that this is the same as $\neg(\forall x)(x \in A \to \neg X[x])$; in shorthand: $\neg(\forall x \in A)\neg X[x]$. Neat! The original $(*)$ has the bounded-quantifier expression $(\exists x \in A)\mathscr{X}[x]$, so the "$\exists \equiv \neg\forall\neg$" property (1.1.1.20) holds for bounded quantifiers! $\square$

1.1.2.4 Exercise. Show that the "$\forall \equiv \neg\exists\neg$" property (1.1.1.21) holds for bounded quantifiers. $\square$

There is a more general way to build sets than by just collecting together and listing a finite number of elements; by "defining property". That is, for any formula $\mathscr{A}(x)$ we collect into a set all the x (values) for which $\mathscr{A}(x)$ is true. We denote this set by the term $\{x : \mathscr{A}(x)\}$. Of course, $\mathscr{A}(x)$ is the defining property or "entrance requirement" that determines membership. To make this precise we define

1.1.2.5 Definition. $S = \{x : \mathscr{A}(x)\}$ is shorthand, suggestive, notation for $(\forall x)(x \in S \equiv \mathscr{A}(x))$. $\square$

1.1.2.6 Remark. Several remarks are in order.

(1) The S that enters in $(\forall x)(x \in S \equiv \mathscr{A}(x))$ is *unique*, that is, if also $(\forall x)(x \in T \equiv \mathscr{A}(x))$, then $S = T$. Indeed, the two imply (Spec) $x \in S \equiv \mathscr{A}(x)$ and $x \in T \equiv \mathscr{A}(x)$, thus, $x \in S \equiv x \in T$ by tautological implication. Generalizing we get $(\forall x)(x \in S \equiv x \in T)$, and hence $S = T$.

Pause. Why not just say "The two mean $S = \{x : \mathscr{A}(x)\}$ and $T = \{x : \mathscr{A}(x)\}$, hence $S = T$ by transitivity of equality"?◀

Because the notation "$S = \{x : \mathscr{A}(x)\}$" is only shorthand for something else. The symbol of equality "=" is inserted in *anticipation*, but *not* due to an *a priori* knowledge, that it will behave correctly, *as equality*. Now we know—through the longer argument and *post facto*—that it was all right to have written "=" after all.

(2) Renaming the bound variable of $(\forall x)(x \in S \equiv \mathscr{A}(x))$ into (a new) z we get the equivalent formula (cf. 1.1.1.33) $(\forall z)(z \in S \equiv \mathscr{A}(z))$. The latter says $S = \{z : \mathscr{A}(z)\}$. So, x (and z) in $\{x : \mathscr{A}(x)\}$ is a bound variable that can be renamed without changing the meaning.

(3) By specializing $(\forall x)(x \in S \equiv \mathscr{A}(x))$ we get $t \in S \equiv \mathscr{A}(t)$ for any term t. This says what our intuition wants: To test if an object t is in the set S, just test that it passes the entrance requirement: $\mathscr{A}(t)$.

Another way to say the same thing is $t \in \{x : \mathscr{A}(x)\}$ iff $\mathscr{A}(t)$.

(4) It is time to be reminded (this was mentioned in passing earlier) that it is *not* the case that every formula $\mathscr{A}(x)$ leads to a *set* $\{x : \mathscr{A}(x)\}$. To think so leads to nasty contradictions, as it did in Cantor's naïve set theory. Examples of formulae that are *not* set-builders are $x \notin x$ and $x = x$ (cf. Section 1.3).

(5) The statements $\{x : \mathscr{A}[x]\} = \{x : \mathscr{B}[x]\}$ and $\mathscr{A}[x] \equiv \mathscr{B}[x]$ are equivalent. Indeed, if we assume the first, then by (1) on p. 27 [which is no more than an application of (vi) from 1.1.1.38] we get $x \in \{x : \mathscr{A}[x]\} \equiv x \in \{x : \mathscr{B}[x]\}$, which by (3) above is (replacing equivalents by equivalents) $\mathscr{A}[x] \equiv \mathscr{B}[x]$. These steps can be reversed [in this direction Ext is invoked rather than (1) on p. 27] to prove the converse.

This is not unexpected at an intuitive level, but it is nice to have it affirmed mathematically: If two "defining properties" are equivalent, then they yield the same result, true or false, on every object we apply them. Thus, precisely the same objects will "pass" each of the two.

(6) $\{x : \mathscr{A}(x, y, z, \ldots)\}$ denotes several different sets (modulo the previous warning), one for each choice of the unspecified values $y, z, \ldots$. These $y, z, \ldots$ are called *parameters*. □

1.1.2.7 Example. A few paragraphs ago we called the set-building process by defining property "more general" than the process of building sets by grouping members and listing them explicitly between braces { }. Here is why: $\{a, b\} = \{x : x = a \vee x = b\}$. This simple "trick" can be applied to any finite set, to represent it by a defining property, namely, the disjunction of atomic formulae of the form $x = a$ for every a that we want to include in the set. □

There is nothing in naïve set theory that helps us argue that the collection of just two objects—the so-called (unordered) *pair*—is a set (one that does not lead to contradictions, that is).[36] In the naïve approach we take it for granted as a "self evident" fact! Equipped with the hindsight of the early (naïve) set theory paradoxes and their workarounds, we can be content that a pair is so "small" as a collection— just two elements—and is not about to cause any problems. In axiomatic approaches there is an axiom which says that we can indeed form a *set* of two elements.

1.1.2.8 Example. $\{x \,:\, x \in \mathbb{N} \wedge x > y\}$ consists of all numbers in $\mathbb{N}$ greater than y. That is: $y + 1, y + 2, y + 3, \ldots$ We may also use the (abuse of) notation $\{x \in \mathbb{N} : x > y\}$ for this set. $\qquad\square$

1.1.2.9 Example. Sometimes we collect more complicated objects than values of variables. For example, $\{x^2 : x = 2 \vee x = 9\}$ is the set $\{2^2, 9^2\}$, i.e., $\{4, 81\}$.

A more complicated example is $S = \{x + y : 0 < x < y\}$, where x, y are varying over $\mathbb{N}$.

(1) Suppose that y is the only a parameter. Then $S = \{y + 1, y + 2, \ldots, 2y - 1\}$.

(2) Suppose x is the only a parameter. Then $S = \{2x + 1, 2x + 2, \ldots\}$. This is all of $\mathbb{N}$, except the segment from 0 to $2x$.

(3) Suppose neither of x or y are parameters. Then $S = \{3, 4, 5, \ldots\}$.

(4) Finally, suppose that both x and y are parameters. Then what $S = \{x + y : 0 < x < y\}$ denotes is an infinite family of one-element sets, using all the elements of $\mathbb{N}$ except $0, 1, 2$: $\{3\}, \{4\}, \{5\}, \ldots$ $\qquad\square$

Because of (4) in 1.1.2.6, mathematicians found a way to *limit the size of collections* to ensure they are, technically, sets. An easy (but not the only) way to do this is to build any new sets as parts (subsets) of some other set that we have already built.

Thus all our discussions in set theory will have some—usually unspecified, large enough to be useable but not too large to be troublesome; and totally unobtrusive[37]— *reference set* tucked away somewhere; let us call it **U**.

This **U** is our "resource" where we take our *set theory* objects from, give values to our variables from, and have our quantifiers vary over. Thus, in set *naïve* theory, when we write $(\forall x)$ or $(\exists x)$ we really mean $(\forall x \in \mathbf{U})$ or $(\exists x \in \mathbf{U})$, respectively. This reference set that we put aside for a discussion is also called the *domain* (of *discourse*).

In other branches of mathematics whose objects can be collected into a set we are less vague about the reference set; thus the calculus of one variable has $\mathbb{R}$ as its domain, while Peano arithmetic has $\mathbb{N}$ as its domain.

It is convenient to have a set with no elements around, the "empty set". This is the set S given by the condition

$$(\forall x)(x \in S \equiv x \neq x\} \qquad\qquad (*)$$

---

[36]Let's face it: naïve set theory is not exactly clear as to what a set is, nor does it care.
[37]We do not pin it down.

By (1) in 1.1.2.6, there is just one set S that satisfies $(*)$, and the symbol reserved for it is $\emptyset$. That it is a set is not in question as it is far too small! It contains nothing. Indeed, by 3 in 1.1.2.6,

$$x \in \emptyset \equiv x \neq x \qquad (**)$$

thus *no x passes the entrance test, the later being false for all x.* Of course, we can write

$$\emptyset = \{x : \neg x = x\}$$

It is useful to note that for any set A, we have $\emptyset \subseteq A$. Indeed, this means $x \in \emptyset \to x \in A$. This is true since, by $(**)$, $x \in \emptyset$ is false.

If we want to build more *complex* sets we will do well to devise operations on sets. Thus,

1.1.2.10 Definition. If A and B are sets, then their *union*, $A \cup B$, is the set $\{x : x \in A \vee x \in B\}$.   □

$A \cup B$ is formed by emptying the members of A and B in a single $\{\ldots\}$ "bag".

The union makes sense even if one or both of A and B stand for atomic elements with no set-theoretic structure, such as numbers.[38] For example, if B is atomic, then $x \in B$ is false, and hence $x \in A \vee x \in B \equiv x \in A$ by 1.1.1.14. That is, $A \cup B = A$ in this case. If both A and B are urelements, then $A \cup B = \emptyset$.

The reader may be wondering: Is it not better to not allow things like $x \in 2$—to make it "illegal", rather than false? No. For one thing, that would mean that before we build an atomic formula $t \in s$ we would then have to analyze first s to ensure it is *not* an urelement; betraying that syntax has to be determined, well, syntactically! Secondly, it would require far too many special cases to be considered in all our definitions.

Thus the following is true [with or without a leading $(\forall x)$; cf. 1.1.1.15]:

$$x \in A \cup B \equiv x \in A \vee x \in B$$

We can form the union of three sets as either $A \cup (B \cup C)$ or $(A \cup B) \cup C$. Since $x \in A \cup (B \cup C)$ is equivalent to

$$x \in A \vee \left(x \in B \vee x \in C \right) \qquad (1)$$

and $x \in (A \cup B) \cup C$ is equivalent to

$$\left(x \in A \vee x \in B \right) \vee x \in C \qquad (2)$$

The equivalence of (1) and (2) proves that $A \cup (B \cup C) = (A \cup B) \cup C$ which renders brackets irrelevant in a chain of two $\cup$—indeed, in any finite chain of $\cup$ by refining

---

[38] Such atomic elements are called *urelements* in the literature.

the previous argument (e.g., using induction[39] on the number of $\cup$ symbols in the chain).

1.1.2.11 Exercise. For any sets A and B, prove $A \cup B = B \cup A$. $\square$

1.1.2.12 Example. For any set A, we have $A \cup \emptyset = A$. Indeed, this translates to [using 1.1.1.15 to eliminate $(\forall x)$, and (3) in 1.1.2.6]

$$x \in A \vee x \neq x \equiv x \in A$$

which is clearly true, since $x \neq x$ is false (cf. 1.1.1.14). $\square$

The "big $\cup$" is a very important generalization of union applied to any collection of sets (and/or urelements), not just finitely many.

1.1.2.13 Definition. (Generalized Union) Let S be a set (may contain sets and/or urelements; may be, intuitively, finite or infinite). The symbol $\bigcup S$ denotes the set that we build by emptying the contents of *every set* in S into a new container.
Mathematically,

$$\bigcup S \overset{Def}{=} \{x : (\exists A \in S) x \in A\}$$

That is, an "x" is put into the new container iff we can spot it inside *some A*, which in turn is in S. $\square$

We have some special cases of the $\bigcup S$: If S is a collection of sets A_i—$\{A_0, A_1, A_2, \ldots\}$—indexed by $i \in \mathbb{N}$ we may write alternatively

$$\bigcup_{i=0}^{\infty} A_i \text{ or } \bigcup_{i \geq 0} A_i \text{ or } \bigcup_{i \in \mathbb{N}} A_i$$

More generally, we may have a collection of sets A_a indexed by a set I other than $\mathbb{N}$—e.g., $I = \mathbb{R}$, $I = \{2, 3\}$. We indicate $\bigcup S$ in this case by the alternative

$$\bigcup_{a \in I} A_a$$

1.1.2.14 Example. $\bigcup_{a \in \{2,3\}} A_a = A_2 \cup A_3$. $\square$

1.1.2.15 Definition. If A and B are sets, then their *intersection*, $A \cap B$, is the set $\{x : x \in A \wedge x \in B\}$. If $A \cap B = \emptyset$ then we call A and B *disjoint*. $\square$

$A \cap B$ is formed by emptying *only the common* members of A and B in a single $\{\ldots\}$ bag. The intersection makes sense even if one or both of A and B are urelements.

---

[39]Knowledge of induction, as well as of everything else in this review is presupposed; this is only a *review*! Induction will be our review-subject in Section 1.4.

For example, if B is atomic, then $x \in B$ is false, and hence $x \in A \wedge x \in B$ is false
1.1.1.14. That is, $A \cap B = \emptyset$ in this case.

1.1.2.16 Definition. (Generalized Intersection) Let S be a set (may contain sets
and/or urelements; may be, intuitively, finite or infinite). The symbol $\bigcap S$ denotes
the set that we build by emptying the contents *that are common to every member* of
S into a new container. Mathematically,

$$\bigcap S \overset{\text{Def}}{=} \{x : (\forall A \in S) x \in A\}$$

That is, an "x" is put into the new container iff we can spot it inside *every A in S.*   □

Thus, if a urelement or $\emptyset$ are members of S, then $\bigcap S = \emptyset$.

For this definition too we have some special cases of the $\bigcap S$: If S is a set of sets
$\{A_0, A_1, A_2, \ldots\}$ we may write alternatively

$$\bigcap_{i=0}^{\infty} A_i \text{ or } \bigcap_{i \geq 0} A_i \text{ or } \bigcap_{i \in \mathbb{N}} A_i$$

More generally, we may have a collection of sets A_a indexed by a set I other than
$\mathbb{N}$—e.g., $I = \mathbb{R}$, $I = \{0, 1, 2, 3, 11\}$. We indicate $\bigcap S$ in this case by the alternative

$$\bigcap_{a \in I} A_a$$

1.1.2.17 Example. $\bigcap_{a \in \{0,7\}} A_a = A_0 \cap A_7$.   □

1.1.2.18 Example. What is $\bigcap \emptyset$? By 1.1.2.16

$$x \in \bigcap \emptyset \equiv (\forall A \in \emptyset) x \in A$$

that is

$$x \in \bigcap \emptyset \equiv (\forall A)(A \in \emptyset \to x \in A)$$

Since $A \in \emptyset$ is false, the entire right hand side of $\equiv$ is true. That is, the left hand side
is true precisely for every x. Recalling that "every x" means "every x-value in the
domain **U**", we have

$$x \in \bigcap \emptyset \equiv x \in \mathbf{U}$$

hence $\bigcap \emptyset = \mathbf{U}$.

Were it not for the "protection" afforded us by the *domain*, "every x" would mean
"everything", and we cannot form the *set of everything*!   □

1.1.2.19 Definition. If A and B are sets, then their *difference*, $A - B$, is the set
$\{x : x \in A \wedge x \notin B\}$. We may also write this as $\{x \in A : x \notin B\}$. If $A = \mathbf{U}$ then
we write $\overline{B}$ for $\mathbf{U} - B$ and call it the *complement* of B.   □

1.1.2.20 Theorem. $A - B = A \cap \overline{B}$.

Proof. $x \in A - B$ means $x \in A \land x \notin B$. Given that $x \notin B \equiv x \in \overline{B}$, we are done. □

1.1.2.21 Example. Let us compute $\{a, b\} - \{a\}$. Now, if $a = b$, then $\{a, b\} = \{a\}$, hence the difference equals $\emptyset$. So let $a \neq b$. We have $\{a, b\} = \{x : x = a \lor x = b\}$ and $\{a\} = \{x : x = a\}$, thus

$$\{a, b\} - \{a\} = \{x : (x = a \lor x = b) \land x \neq a\} \tag{1}$$

The reader will have no trouble verifying that, since both $x = a \lor x = b$ and $\neg x = a$ are true in our context, we have

$$(x = a \lor x = b) \land \neg x = a \equiv x = b \tag{2}$$

Indeed, in the context of $\{a, b\} - \{a\}$, the truth-value of $x = a$ is **f** and thus the left hand side of $\equiv$ in (2) has the same truth-value as the right hand side—cf. 1.1.1.14 and note the truth-value of $\mathscr{A} \land \mathscr{B}$ when $\mathscr{A}$ is **t** and also the truth-value of $\mathscr{A} \lor \mathscr{B}$ when $\mathscr{A}$ is **f**. Therefore the right hand side of (1) simplifies to $\{x : x = b\}$ [cf. 1.1.2.6, item (5)], i.e., $\{b\}$. This is the difference. □

1.1.2.22 Example. What conclusions may we draw from the following equality?

$$\Big\{\{a\}, \{a, b\}\Big\} = \Big\{\{A\}, \{A, B\}\Big\} \tag{1}$$

Well, we get, first off, that

$$\bigcap\Big\{\{a\}, \{a, b\}\Big\} = \bigcap\Big\{\{A\}, \{A, B\}\Big\}$$

by an application of Exercise 1.8.4 (p. 85). That is,

$$\{a\} = \{A\}$$

hence

$$a = A \tag{2}$$

This time let's apply $\bigcup$ to both sides of (1), We get $\{a, b\} = \{A, B\}$, which, by (2), becomes

$$\{a, b\} = \{a, B\} \tag{3}$$

Applying again Exercise 1.8.4 (p. 85), to the function $x - \{a\}$ this time, we get via (3)

$$\{a, b\} - \{a\} = \{a, B\} - \{a\} \tag{4}$$

If $a = b$, then the left hand side of (4) is $\emptyset$, so $a = B$ and therefore

$$b = B \tag{5}$$

If $a \neq b$, then, also $a \neq B$ (else the right hand side, and hence the left, is $\emptyset$). By the previous example, (4) yields $\{b\} = \{B\}$ in this case, so we obtain (5) once more.
In summary, (1) implies (2) and (5).   □

1.1.2.23 Definition. (Kuratowski's Ordered Pair) For any objects x and y (sets or not), we reserve the symbol (x, y) as an *abbreviation* of the set $\{\{x\}, \{x, y\}\}$. We call (x, y) the *ordered pair* of x and y.   □

The nomenclature for (x, y) stems from the property established in 1.1.2.22, that

$$\text{If } (x, y) = (X, Y), \text{ then } x = X \text{ and } y = Y \qquad (pair)$$

that is, *order* or *position* matters in the pair. This property is not shared by $\{a, b\}$ since, by extensionality, $\{a, b\} = \{b, a\}$, as we know. Of course, the converse—that $x = X$ and $y = Y$ implies $(x, y) = (X, Y)$—is not miraculous at all, and simply follows by two applications of Exercise 1.8.4, p. 85: First, $(x, z) = (X, z)$ and then $(x, y) = (X, Y)$.

The reader is familiar with ordered pairs from analytic geometry, where ordered pairs of real numbers give the coordinates of points on the Cartesian plane. Indeed, the concept of Cartesian product relies on the (x, y) objects.

1.1.2.24 Example. So, $(1, 2) \neq (2, 1)$ lest $1 = 2$. Is $(1, 1) = \{1\}$? To ask explicitly, is $\{\{1\}, \{1, 1\}\} = \{1\}$—that is, by extensionality, twice—is $\{\{1\}\} = \{1\}$? Not unless $\{1\} = 1$, but this cannot be since 1 is an urelement, it has no set-theoretic structure.   □

How about

$$A = \{A\} \qquad (*)$$

in general, for some set A? We cannot use here a "type" argument as we did above, since both sides of = are of type set.

Can this be? An unfair question this, since naïve set theory cannot resolve it. If we grant $(*)$, then we have $A \in A$. Well, can *this* be?

Axiomatic foundations disallow this state of affairs, basing it on an intuitive concept[40] that "sets are formed by stages", so you can't have a set (built) before you have (built) its members. $A \in A$ requires the left A being available before the right A is—an untenable proposition. Thus set theorists have adopted an axiom (of *foundation*) which precludes bottomless (unfounded) chains such as

$$\ldots \in d \in c \in b \in a$$

The impossibility of $A \in A$ follows from this, since otherwise it would lead to $\ldots \in A \in A \in A \in A$.

---

[40]All reasonable axioms are based on intuitively acceptable concepts. The idea that sets are formed by stages led to many nice axioms of axiomatic set theory.

Ordered triples, quadruples and beyond can be easily defined using the ordered pair as the basic building block:

1.1.2.25 Definition. (Ordered Tuples) We define the symbol $\langle x_1, \ldots, x_n \rangle$, pronounced the *ordered n-tuple* or just—*n*-tuple—by two recurrence equations:

$$\langle x_1, x_2 \rangle \qquad = (x_1, x_2)$$
and, for $n \geq 2$,
$$\langle x_1, \ldots, x_n, x_{n+1} \rangle = (\langle x_1, \ldots, x_n \rangle, x_{n+1})$$

x_i is the i-th component of the tuple. $\langle a, b \rangle$ is also called an *2-tuple* (as well as an ordered pair).

We often employ the abbreviation $\vec{x}_n$ for the (ordered) sequence $x_1, \ldots, x_n$. The presence of "$\vec{\ }$" will not permit the confusion between the sequence $\vec{x}_n$ and the component x_n. If the length n is immaterial or known, we may just write $\vec{x}$. $\square$

The above is a simple *recursive* or inductive *definition*. It compactifies and renders finite an *infinite-length definition* such as:

$$\langle x, y \rangle \qquad = (y, y)$$
$$\langle x, y, z \rangle \qquad = (\langle x, y \rangle, z)$$
$$\langle x, y, z, u \rangle \quad = (\langle x, y, z \rangle, u)$$
$$\langle x, y, z, u, w \rangle = (\langle x, y, z, u \rangle, w)$$
$$\vdots$$

In essence, it *finitely describes* the "$\vdots$" above.

This is entirely analogous with *loops* in programming where a variable-length (and therefore syntactically illegal)—it depends on the value of N—program segment is correctly implemented as a loop; that is, the following, where $X \leftarrow X + 1$ occurs N times

read N, X
$X \quad \leftarrow 0$
$X \quad \leftarrow X + 1$
$\quad \vdots$
$X \quad \leftarrow X + 1$

is captured by this

read $\quad N, X$
$X \qquad \leftarrow 0$
repeat N **times**

$\{$

$X \qquad \leftarrow X + 1$

}

The reader has seen recursive definitions similar to the one in 1.1.2.25, for example, the one that defines nonnegative (integer) powers of a non zero real number a by two equations:

$a^0 \quad = 1$
  and, for $n \geq 0$,
$a^{n+1} = a \cdot a^n$

Recursive definitions of this and of more general types are reviewed in Section 1.4.

1.1.2.26 Exercise. Show that the name ordered (4-) tuple is apt for $\langle x, y, z, w \rangle$ by showing that $\langle x, y, z, w \rangle = \langle X, Y, Z, W \rangle$ implies that $x = X$, $y = Y$, $z = Z$ and $w = W$.   □

1.1.2.27 Exercise. Write down explicitly the set for which the tuple $\langle x, y, z, w \rangle$ is *compact notation*.   □

1.1.2.28 Definition. (Cartesian Product) Let $A_1, \ldots, A_n$ be sets. Then their *Cartesian product, in the given order*, is the set

$$\Big\{ \langle a_1, \ldots, a_n \rangle : a_i \in A_i, \text{ for } i = 1, \ldots, n \Big\}$$

We will employ the symbols

$$\underset{1 \leq i \leq n}{\text{\Large ✕}} A_i \quad \text{or} \quad \overset{n}{\underset{i=1}{\text{\Large ✕}}} A_i$$

as alternative shorthands for this product.

If $A_1 = A$ and $A_2 = B$ then we write $A \times B$ rather than $\overset{2}{\underset{i=1}{\text{✕}}} A_i$. It is all right, but sloppy, to write $A_1 \times \cdots \times A_n$ for the general case. If $A_i = C$, for all i, then we write C^n for $\overset{n}{\underset{i=1}{\text{✕}}} A_i$.   □

1.1.2.29 Example. $\{1\} \times \{2\} = \{\langle 1, 2 \rangle\}$ and $\{2\} \times \{1\} = \{\langle 2, 1 \rangle\}$. Thus $\{1\} \times \{2\} \neq \{2\} \times \{1\}$; the Cartesian product is not commutative in general.   □

$\overset{n}{\underset{i=1}{\text{\Large ✕}}} A_i$ can be given by a simple recursive (inductive) definition:

$$\overset{1}{\underset{i=1}{\text{\Large ✕}}} A_i = A_1$$
  and, for $n \geq 1$,
$$\overset{n+1}{\underset{i=1}{\text{\Large ✕}}} A_i = \Big(\overset{n}{\underset{i=1}{\text{\Large ✕}}} A_i \Big) \times A_{n+1}$$

The reader should verify that this is consistent with 1.1.2.28 and 1.1.2.25.

Similarly, A^n can be defined inductively (recursively) as

$A^1 \quad = A$

and, for $n \geq 1$,

$A^{n+1} = A^n \times A$

1.1.2.30 Example. $A \times \emptyset = \emptyset$, since $\langle x, y \rangle \in A \times \emptyset$ is equivalent to $x \in A \wedge y \in \emptyset$, which is false. Similarly, $\emptyset \times A = \emptyset$. □

We conclude our review of set operations with the *power set*.

1.1.2.31 Definition. (Power Set) For any set A, its *power set*—denoted as $\mathcal{P}(A)$ or 2^A—is $\{x : x \subseteq A\}$. □

1.1.2.32 Example. Thus, $2^\emptyset = \{\emptyset\}$; $2^{\{\emptyset\}} = \{\emptyset, \{\emptyset\}\}$; and $2^{\{\emptyset, \{\emptyset\}\}} = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}$.

$2^{\{0,1\}} = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$.

Since $\emptyset \subseteq A$ and $A \subseteq A$ for any set A, we have always $\emptyset \in 2^A$ and $A \in 2^A$. □

1.1.3 Alphabets, Strings and Languages

A *string* or *expression* or a *word* is just a tuple, all of whose components come from the same set, A, the latter being called the *alphabet*. We say that "*x is a string of length n* over *the alphabet A*" meaning $x \in A^n$.

Traditionally, strings are written down without separating commas or spaces, nor with enclosing angular brackets. So if $A = \{a, b\}$ we will write $aababa$ rather than $\langle a, a, b, a, b, a \rangle$.

Concatenation of $\langle a_1, \ldots, a_m \rangle$ and $\langle b_1, \ldots, b_n \rangle$ in that order, denoted as

$$\langle a_1, \ldots, a_m \rangle * \langle b_1, \ldots, b_n \rangle$$

is the string of length $m + n$

$$\langle a_1, \ldots, a_m, b_1, \ldots, b_n \rangle$$

Clearly, concatenation as defined above is *associative*, that is, for any strings x, y and z we have $(x * y) * z = x * (y * z)$.

It is convenient to introduce a *null* or *empty* string, that has no members, and hence has length 0. We will denote it by ϵ. We will not attempt to give it a precise tuple counterpart, but some people write "$\langle \rangle$" with nothing between brackets.
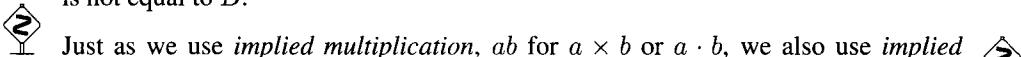
At the intuitive level, and given how concatenation was defined, we see that $x * \epsilon = \epsilon * x = x$ for any string x. We will distinguish $\emptyset$ and ϵ since one is an "unordered set" will the other is ordered; but both are empty.

The set of *all strings of non zero length* over A is denoted by A^+. This is, of course,

$$\bigcup_{i=1}^{\infty} A^i$$

Adding ϵ to the above we get the unqualified set of all strings over A, denoted by A^*; that is, $A^* = A^+ \cup \{\epsilon\}$. This set is often called *the Kleene star* of A.

A string A is a *prefix* of a string B if there is a string C such that $B = A * C$. It is a *suffix* of B if for some D, we have $B = D * A$. The prefix (suffix) is *proper* if it is not equal to B.
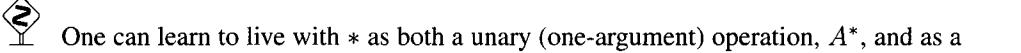
Just as we use *implied multiplication*, ab for $a \times b$ or $a \cdot b$, we also use *implied concatenation*, xy for $x * y$ —leaving it up to the context to fend off ambiguities.

1.1.3.1 Example. Not all alphabets are amenable to writing tuples in "string-notation". For example, $A = \{1, 11\}$ has a problem. The notation 111 is ambiguous: Do we mean $\langle 1, 1, 1 \rangle$, $\langle 11, 1 \rangle$, or $\langle 1, 11 \rangle$? □

1.1.3.2 Definition. (Languages) A *language*, L, over an alphabet A is just a subset of A^*. □

The "interesting" languages are those that are *finitely definable*. *Automata and language theory* studies the properties of such finitely definable languages and of the "machinery" that effects these finite definitions.

1.1.3.3 Definition. (Concatenation of Languages) If L and M are two languages over an alphabet A, then the symbol $L * M$ or simply (implied concatenation) LM means the set $\{xy : x \in L \land y \in M\}$. □

One can learn to live with $*$ as both a unary (one-argument) operation, A^*, and as a binary one, $L * M$, much the same way we can see no ambiguity in uses of minus as $-x$ and $y - z$.

1.2 RELATIONS AND FUNCTIONS

Intuitively, a relation is a formula, $\mathscr{A}(x, y, z)$. We say that a, b, c are related according to $\mathscr{A}(x, y, z)$ just in case $\mathscr{A}(a, b, c)$ is true. Influenced by the set theorist who wants to realize "everything" (even formulae) as some set, the modern mathematician views relations *extensionally* (by what they contain) *as sets*. For example, $\mathscr{A}(x, y, z)$ naturally defines this set, its *extension*: $\{\langle x, y, z \rangle : \mathscr{A}(x, y, z)\}$. One goes one step further and forgets the role of $\mathscr{A}$. As a result, we give a totally extensional definition of a relation as a set of tuples, disregarding how it may have been formed by a "defining property".

1.2.0.4 Definition. A *binary relation* —or simply *relation*—R is a set of 2-tuples. We use the notations $\langle x, y \rangle \in R$, xRy and $R(x, y)$ to *mean the same thing.* □

A relation R, on the other hand, immediately gives rise to an atomic *formula* — variably denoted by one of the forms $\langle x, y \rangle \in R$, xRy or $R(x, y)$— just as the specific relations $<$ and $\in$ lead to the atomic formulae $x < y$ and $x \in y$ (cf. 1.1.1.1).

Pause. So, every formula $\mathscr{A}(x, \ldots)$ defines the relation $A = \{\langle x, \ldots \rangle : \mathscr{A}(x, \ldots)\}$, and every (binary) relation R defines the (atomic) formula xRy; right?◄

Not exactly. If we have an "enormous"[41] supply of symbols for formulae, then we could do this, since for every relation we could then introduce a formula symbol (so-called "predicate")—say, R, ϕ, $\prec$, or whatever—by a *definition*, such as "$x \prec y$ if and only iff $\langle x, y \rangle$ is a member of the given relation". This fails in most practical cases, e.g., in set theory and arithmetic, where our symbol-alphabet is finite or enumerable (cf. Definition 1.3.0.40). To write down —that is, to "have"—a formula, we need *notation* for it. As we will see in Section 1.3, we have far "more" binary relations R than we have means to "write them down" as formulae xRy, *if our symbol-alphabet is finite or enumerable.*

Hm. Did I not just write down "xRy"? Well, yes; however, writing one or two symbols down, like "R" or "Q" and saying that they "stand for relations" does not equate to having *a system of notation to write down* all *binary relations.*

Intuitively, a relation is a table—possibly infinite in length—of pairs like

x	z
a_1^1	a_2^1
a_1^2	a_2^2
$\vdots$	$\vdots$

The head-row names the relation's variables. The entries in each row represent the tuples-members of the relation. It is standard convention to think of the left column, headed by x as the "input-side", while the right column as the "output-side". This is consistent with a "black box" view of the relation

$$a_1^i \longrightarrow \boxed{} \longrightarrow a_2^i$$

where we don't know or don't care what makes it tick, but we do know which inputs cause which output(s).

It is not *a priori* precluded to have the same input produce several outputs. For example, think of $R = \{\langle 1, 2 \rangle, \langle 1, 1 \rangle, \langle 1, 7 \rangle\}$.

Thus, the relation (table) establishes a *one-to-many* input/output correspondence.

Contrary to our viewpoint with formulae $\mathscr{A}(x, y)$—where the *input* variables *are* all the free variables, here x and y—in the case of relations we are allowed *two points of view*, one being the one presented above, and the other where *both* x and y are the inputs of the relation $R(x, y)$. The context will fend for us!

---

[41] See also Section 1.3 to appreciate that not all infinities are equal in size.

Of course, when we take *all* the variables of a relation as input, then the output that is implied—just as in the case of formulae—is one of **t** or **f**.

Since $\langle \vec{a}_{n+1} \rangle = (\langle \vec{a}_n \rangle, a_{n+1})$, there is no loss of generality in focusing mostly on binary relations. In other words, the left (input) column may well be a column of n-tuple entries $a_1^j = \langle A_1^j, A_2^j, \ldots, A_n^j \rangle$. The relation is then said to be $(n+1)$-ary and, in table form, would look like

x_1	$\ldots$	x_n	z
A_1^1	$\ldots$	A_n^1	a_1^1
A_1^2	$\ldots$	A_n^2	a_2^2
$\vdots$		$\vdots$	$\vdots$

The set consisting of the entries in the input column is the relation's domain—that is, those inputs that cause some output—while those in the output column constitute the range—that is, the set of all outputs.

1.2.0.5 Definition. Let R be a (binary) relation. Its *domain*, denoted by $\mathrm{dom}(R)$, is the set $\{x : (\exists y)xRy\}$. Its *range*, denoted by $\mathrm{ran}(R)$, is the set $\{x : (\exists y)yRx\}$.   □

1.2.0.6 Example. Let $R = \{\langle x, x \rangle : x \in \mathbb{N}\}$. Then $\mathrm{dom}(R) = \mathrm{ran}(R) = \mathbb{N}$.
  Let $Q = \{\langle 0, x \rangle : x \in \mathbb{N}\}$. Then $\mathrm{dom}(Q) = \{0\}$ and $\mathrm{ran}(Q) = \mathbb{N}$.
  Let $S = \{\langle x, 0 \rangle : x \in \mathbb{N}\}$. Then $\mathrm{ran}(S) = \{0\}$ and $\mathrm{dom}(S) = \mathbb{N}$.
  Let $T = \{\langle 0, 0 \rangle, \langle 0, 7 \rangle\}$. Then $\mathrm{dom}(T) = \{0\}$ and $\mathrm{ran}(T) = \{0, 7\}$.   □

An abstract *term* of logic captures well the intentional aspect of a *function*—indeed a *function call*—of mathematics and programming: we have a "rule" that defines the input/output dependence. For example, the "rule" $x + y$ that tells us how the output is to be obtained, once we have the x and y values.

While, in logic, a term is a totally different type of object from a formula, on the other hand, extensionally—i.e., in its set theory realization—a function is a subsidiary construct of a relation. Referring back to the black box analogy, a function is simply a relation that obeys the restriction that *no input can cause more than one output*. So a function, extensionally, is a *single-valued* relation.

1.2.0.7 Definition. A *function* R is a *single-valued* relation. That is, whenever we have both xRy and xRz, we will also have $y = z$.

It is traditional to use, generically, lower case letters from among f, g, h, k to denote functions but this is by no means a requirement.   □

1.2.0.8 Example. The empty set is a relation of course, the empty set of pairs. It is also a function since

$$\langle x, y \rangle \in \emptyset \wedge \langle x, z \rangle \in \emptyset \rightarrow y = z$$

vacuously, by virtue of the left hand side of $\rightarrow$ being false.   □

It is often the case that we study relations, and functions, that take their inputs from a given set A that is fixed throughout the study, and, similarly, produce their outputs in a given fixed set B.

For example, our work on computability in this volume deals exclusively with functions and relations whose inputs and outputs are from $\mathbb{N}$.

Additional terminology has been invented to name these fixed "input-" and "output-spaces" and also to name relations that *fully* utilize one or the other of these spaces. The input space is called the *left field* while the output space is called the *right field*.

If A and B are the adopted left and right fields of the function or relation R then clearly $R \subseteq A \times B$, and, in particular, $\text{dom}(R) \subseteq A$ while $\text{ran}(R) \subseteq B$. A well-established abbreviation—other than $R \subseteq A \times B$— for "R is relation with left field A and right field B" is $R : A \to B$, read "R is a relation from A to B".

If $A = B$, then we say "R is a relation *on* A".

If $\text{dom}(R) = A$, then R is *totally defined* on A. We just say "R is *total*". If $\text{ran}(R) = B$, then R "covers" the entire right field with its outputs. We say "R is *onto*".

Pause. Totalness and ontoness are *relative* to a left field and a right field, respectively; they are *not absolute notions*.◄

A relation $R : A \to B$ is either total or not (*nontotal*). An indifference toward which is which will be expressed by calling R *partial*. Thus "partial" is *not* synonymous with "nontotal". *All* relations are therefore *partial relations*.

All the terminology introduced in this -segment applies to the special case of functions as well.

We now turn to notation and concepts specific to functions. Let f be a function. First off, $f(a)$ denotes the unique b such that $a f b$ or $\langle a, b \rangle \in f$. Note that such a b exists iff $a \in \text{dom}(f)$. Thus

$$b = f(a) \text{ iff } \langle a, b \rangle \in f \text{ iff } a f b$$

We write $f(a) \downarrow$—pronounced "$f(a)$ is defined" or "$f(a)$ *converges*"—to mean $a \in \text{dom}(f)$. Otherwise we write $f(a) \uparrow$—pronounced "$f(a)$ is undefined" or "$f(a)$ *diverges*".

The set of *all* outputs of a function, when the inputs come from a particular set X, is called the *image of X under f* and is denoted by $f_\to(X)$. Thus,

$$f_\to(X) = \{f(x) : x \in X\} \tag{1}$$

Pause. So far we have been giving definitions regarding functions of one variable. Or have we?◄

Not really: We have already said that the multiple-input case is subsumed by our notation. If $f : A \to B$ and A is a set of n-tuples, then f is a function of "n-variables", essentially. We usually abuse the notation $f(\langle \vec{x}_n \rangle)$ and write instead $f(\vec{x}_n)$.

The *inverse image* of a set Y under a function is useful as well, that is, the set of *all* inputs the generate f-outputs in Y. It is denoted by $f_\leftarrow(Y)$ and is defined as

$$f_\leftarrow(Y) = \{x : f(x) \in Y\} \tag{2}$$

Regarding, say, the definition of $f_\rightarrow$:

> *What if* $f(a) \uparrow$*? How do you "collect" an* undefined value *into a set?*

Well, you don't. Both (1) and (2) have a rendering that is independent of the notation "$f(a)$".

$$f_\rightarrow(X) = \{y : (\exists x \in X)\langle x, y \rangle \in f\} \tag{1'}$$

$$f_\leftarrow(Y) = \{x : (\exists y \in Y)\langle x, y \rangle \in f\} \tag{2'}$$

1.2.0.9 Example. Thus, $f_\rightarrow(\{a\}) = \{f(x) : x \in \{a\}\} = \{f(x) : x = a\} = \{f(a)\}$.

Let now $g = \{\langle 1, 2 \rangle, \langle \{1, 2\}, 2 \rangle, \langle 2, 7 \rangle\}$. Thus, $g(\{1, 2\}) = 2$, but $g_\rightarrow(\{1, 2\}) = \{2, 7\}$. Also, $g(5) \uparrow$ and $g_\rightarrow(\{5\}) = \emptyset$.

On the other hand, $g_\leftarrow(\{2, 7\}) = \{1, \{1, 2\}, 2\}$ and $g_\leftarrow(\{2\}) = \{1, \{1, 2\}\}$, while $g_\leftarrow(\{8\}) = \emptyset$. $\square$

When $f(a) \downarrow$, then $f(a) = f(a)$ as is naturally expected. What about when $f(a) \uparrow$? This begs a more general question that we settle as follows:

First, seeking help from logic. For any formula $\mathscr{A}[x]$ and term t *that does not contain the variable* x,

$$\vdash \mathscr{A}[t] \equiv (\exists x)(x = t \wedge \mathscr{A}[x]) \tag{1}$$

We settle (1) by a ping pong argument (putting aside an urge to proclaim "but, it is obvious!").

($\rightarrow$) direction. We want to prove $\mathscr{A}[t] \rightarrow (\exists x)(x = t \wedge \mathscr{A}[x])$. Note that

$$\mathscr{A}[t] \rightarrow t = t \wedge \mathscr{A}[t]$$

is true. So is

$$t = t \wedge \mathscr{A}[t] \rightarrow (\exists x)(x = t \wedge \mathscr{A}[x])$$

by 1.1.1.40 since we may view $x = t \wedge \mathscr{A}[x]$ as $(x = t \wedge \mathscr{A}[x])[x]$ and thus view $t = t \wedge \mathscr{A}[t]$ as $(x = t \wedge \mathscr{A}[x])[t]$ due to the absence of x in t. Using this and the previous displayed formula along with tautological implication we get what we want.

($\leftarrow$) direction. We want to prove $(\exists x)(x = t \wedge \mathscr{A}[x]) \rightarrow \mathscr{A}[t]$. We will employ the deduction theorem, so we *freeze all free variables* in $(\exists x)(x = t \wedge \mathscr{A}[x])$, and *assume* it. So, let us call a an x-value that makes the quantification work (cf. 1.1.1.13). We have

$$a = t \wedge \mathscr{A}[a] \tag{2}$$

Since the $a = t$ part of (2) and the Leibniz axiom [(vi) of 1.1.1.38] yield $\mathscr{A}[a] \equiv \mathscr{A}[t]$, the remaining part of (2) yields the truth of $\mathscr{A}[t]$, as needed.

Transferring the above result to the specific case of substituting terms into input variables[42] of *relations*, we have the following.

1.2.0.10 Remark. For any ($m + n + 1$-ary) relation $R(z_1, \ldots, z_m, x, y_1, \ldots, y_n)$, function f, and object a, the substitution $R(z_1, \ldots, z_m, f(a), y_1, \ldots, y_n)$ is shorthand for

$$(\exists w)\Big(w = f(a) \wedge R(z_1, \ldots, z_m, w, y_1, \ldots, y_n)\Big) \tag{3}$$

Note that $w = f(a)$ entails that $f(a) \downarrow$, so that if *no such w exists* [the case where $f(a) \uparrow$], then (3) is *false*; *not* undefined!

This convention is prevalent in the modern literature [cf. Hinman (1978), p. 9]. Contrast with the convention in Kleene (1943), where, for example, an expression like $f(a) = g(b)$ [and even $f(a) = b$] is allowed to be undefined! □

1.2.0.11 Example. Thus, applying the above twice, $f(a) = g(b)$ means $(\exists u)(\exists w)(u = f(a) \wedge w = g(b))$ which simplifies to $(\exists u)(u = f(a) \wedge u = g(b))$. In particular, $f(a) = g(b)$ entails that $f(a) \downarrow$ and $g(b) \downarrow$. □

The above is unsettling as it fails to satisfy the reflexivity of equality [axiom (v) of 1.1.1.38]: If $f(a) \uparrow$, then $\neg f(a) = f(a)$. To get around this difficulty, Kleene (1943) has *extended* equality to include the *undefined* case, restoring reflexivity in this "generalized" equality relation. We will use this so-called *Kleene-complete-equality* quite often in the chapter on computability. This version of equality uses a different symbol, $\simeq$, to avoid confusion with the "standard" equality, =, of Remark 1.2.0.10 that compares only objects (not "undefined values"). For any two functions f and g, we define

$$f(a) \simeq g(b) \stackrel{\text{Def}}{\equiv} f(a) \uparrow \wedge g(b) \uparrow \vee (f(a) \downarrow \wedge g(b) \downarrow \wedge f(a) = g(b))$$

while $f(a) \simeq b$ means the same thing as afb, that is, $f(a) = b$.

1.2.0.12 Example. Let $g = \{\langle 1, 2 \rangle, \langle \{1, 2\}, 2 \rangle, \langle 2, 7 \rangle\}$. Then, $g(1) = g(\{1, 2\})$ and also $g(1) \simeq g(\{1, 2\})$. Also, $g(1) \not\simeq g(2)$ and also $g(1) \neq g(2)$. Moreover, $g(3) \simeq g(9)$. □

If f and g are functions and $f \subseteq g$ then g is an *extension* of f while f is a *restriction* of g. If $g : A \to B$, one way to restrict g to f is to choose for f a "smaller" left field, $C \subseteq A$, and take for f only those 2-tuples that have the their first component in C. We write this as $f = g \upharpoonright C$. Thus, $g \upharpoonright C = g \cap (C \times B)$.

Note that every function f extends the *totally undefined function* $\emptyset$ since $\emptyset \subseteq f$.

1.2.0.13 Definition. A function f is 1-1 if for all x and y, $f(x) = f(y)$ implies $x = y$. □

Note that $f(x) = f(y)$ implies that $f(x) \downarrow$ and $f(y) \downarrow$ (1.2.0.10).

---

[42]Here we view *every* variable of R as input; output is **t** or **f**. Cf. discussion on p. 42.

1.2.0.14 Example. $\{\langle 1, 1 \rangle\}$ and $\{\langle 1, 1 \rangle, \langle 2, 7 \rangle\}$ are 1-1. $\{\langle 1, 0 \rangle, \langle 2, 0 \rangle\}$ is not. $\emptyset$ is 1-1 vacuously. $\square$

1.2.0.15 Definition. (Relational Converse) If R is a relation, then its *converse*, denoted by R^{-1} is the relation $\{\langle x, y \rangle : yRx\}$. $\square$

1.2.0.16 Exercise. Prove that if f is a 1-1 function, then the relation converse f^{-1} is a function (that is, single-valued). $\square$

1.2.0.17 Definition. (1-1 Correspondence) A function $f : A \to B$ is called a *1-1 correspondence* iff it is all three: 1-1, total and onto.

Often we say that A and B are *in 1-1 correspondence* writing $A \sim B$, omitting mention of the function that *is* the 1-1 correspondence. $\square$

The terminology is derived from the fact that every element of A is paired with precisely one element of B and vice versa.

1.2.0.18 Definition. (Composition of Relations and Functions) Let $R : A \to B$ and $Q : B \to C$ be two relations. The relation $R \circ Q : A \to C$, their *relational composition*, is the relation

$$\left\{ \langle x, y \rangle : (\exists z)(xRz \land zQz) \right\} \tag{1}$$

If R and Q are functions, then their *functional composition*—or composition as functions—refers to their relational composition, but has a different notation: (QR) (no "∘") is an alternative notation for $R \circ Q$; *note the order reversal.* $\square$

So $xR \circ Qy$ iff $(\exists z)(xRz \land zQy)$. Let then $xR \circ Qy$ and also $xR \circ Qw$. For some a and b, guaranteed to exist, we have xRa and aQy on one hand and xRb and bQw on the other. Let next R and Q both be functions. Then $a = b$ (from R) and hence $y = w$ (from Q). Thus,

If R and Q are functions, then so is their composition, $R \circ Q$ or (QR).

Let R and Q still be functions. Assume that $(QR)(a) \downarrow$. Then, for some b, $aR \circ Qb$, and hence, for some c, aRc and cQb. That is,

$$R(a) = c \text{ and } Q(c) = b. \text{ For short, } (QR)(a) = Q(R(a)) \tag{2}$$

The above justifies the order reversal for the alternative notation of "functional composition".

1.2.0.19 Theorem. *Relational composition is associative, that is, $R \circ (Q \circ S) = (R \circ Q) \circ S$ for any relations R, Q, S. If the relations are functions we may also write $((SQ)R) = (S(QR))$.*

Proof. See Exercise 24 in Section 1.8. $\square$

1.2.0.20 Definition. The *identity function* on a set A is $\mathbf{1}_A : A \to A$ given by $\mathbf{1}_A(x) = x$ for all $x \in A$. $\square$

By 1.2.0.19, if R, Q, T, S are relations, then $R \circ Q \circ T \circ S$ is *unambiguous*, as it means the same thing regardless of how we insert brackets. In particular,

$$\underbrace{R \circ R \circ \cdots \circ R}_{n \geq 1 \text{ copies of } R}$$

is unambiguous regardless of the absence of brackets. We have the shorthand R^n for the above chain of compositions. We can put this into an inductive definition similar to the one that defines positive powers of a positive real:

1.2.0.21 Definition. (Relational Powers) The symbol R^n, for $n \geq 1$, is the *relational power* of R and is defined as

$$R^1 = R$$

and, for $n \geq 1$,

$$R^{n+1} = R \circ R^n$$

If R is a relation on A, then we replace the first equation by $R^0 = \mathbf{1}_A$ and the condition for the second becomes "and, for $n \geq 0$". $\square$

The following interesting result connects the notions of ontoness and 1-1ness with the "algebra" of composition.

1.2.0.22 Theorem. *Let $f : A \to B$ and $g : B \to A$ be functions. If $(gf) = \mathbf{1}_A$, then g is onto while f is total and 1-1.*

We say that g is a *left inverse* of f and f is a *right inverse* of g.

Proof. **About g:** Our goal, ontoness, means that, for each $x \in A$, a y exists such that $g(y) = x$. Fix then an $x \in A$. By $(gf) = \mathbf{1}_A$, we have $(gf)(x) = \mathbf{1}_A(x) = x$. But $(gf)(x) = g(f(x))$. So take $y = f(x)$.

About f: As seen above, $x = g(f(x))$ for each $x \in A$. Since this is the same as "$xf \circ gx$ is true", there must be a z such that xfz and zgx. The first of these says $f(x) = z$ and therefore $f(x) \downarrow$. This settles totalness.

For the 1-1ness, let $f(a) = f(b)$. Applying g to both sides (that is, using Exercise 1.8.4) we get $g(f(a)) = g(f(b))$. But this says $a = b$, by $(gf) = \mathbf{1}_A$, and we are done. $\square$

1.2.0.23 Example. *The above is as much as we can be expected to prove.* For example, say $A = \{1, 2\}$ and $B = \{3, 4, 5, 6\}$. Let $f : A \to B$ be $\{\langle 1, 4 \rangle, \langle 2, 3 \rangle\}$ and $g : B \to A$ be $\{\langle 4, 1 \rangle, \langle 3, 2 \rangle, \langle 6, 1 \rangle\}$, or in friendlier notation

$f(1) = 4$
$f(2) = 3$

and
$$g(3) = 2$$
$$g(4) = 1$$
$$g(5)\uparrow$$
$$g(6) = 1$$

Clearly, $(gf) = \mathbf{1}_A$ holds, but note:

(1) f is not onto.

(2) g is neither 1-1 nor total. $\square$

1.2.0.24 Example. With $A = \{1, 2\}$, $B = \{3, 4, 5, 6\}$ and $f : A \to B$ and $g : B \to A$ as in the previous example, consider also the functions $\tilde{f}$ and $\tilde{g}$ given by

$$\tilde{f}(1) = 6$$
$$\tilde{f}(2) = 3$$
and
$$\tilde{g}(3) = 2$$
$$\tilde{g}(4) = 1$$
$$\tilde{g}(5)\uparrow$$
$$\tilde{g}(6) = 2$$

Clearly, $(\tilde{g}f) = \mathbf{1}_A$ and $(g\tilde{f}) = \mathbf{1}_A$ hold, but note:

(1) $f \neq \tilde{f}$.

(2) $g \neq \tilde{g}$.

Thus, neither left nor right inverses need to be unique. The article "a" in the definition of said inverses was well-chosen. $\square$

The following two partial converses of 1.2.0.22 are useful.

1.2.0.25 Theorem. *Let $f : A \to B$ be total and 1-1. Then there is an onto $g : B \to A$ such that $(gf) = \mathbf{1}_A$.*

Proof. Consider the converse relation (1.2.0.15) of f and call it g:

$$g \overset{\text{Def}}{=} \{\langle x, y \rangle : f(y) = x\} \tag{1}$$

By Exercise 1.2.0.16, $g : B \to A$ is a (possibly nontotal) function. Note that, for any $a \in A$, there is a b such that $f(a) = b$ (f is total), and, by (1), $g(b) = a$. That is, $g(f(a)) = a$, or $(gf) = \mathbf{1}_A$. $\square$

1.2.0.26 Remark. By (1) above, $\text{dom}(g) = \{x : (\exists y)\langle x, y \rangle \in g\} = \{x : (\exists y)f(y) = x\} = \text{ran}(f)$. $\square$

1.2.0.27 Theorem. *Let $f : A \to B$ be onto. Then there is a total and 1-1 $g : B \to A$ such that $(fg) = \mathbf{1}_B$.*

Proof. By assumption, $\emptyset \neq f_\leftarrow(\{b\}) \subseteq A$, for all $b \in B$. To define $g(b)$ choose *one* $c \in f_\leftarrow(\{b\})$ and set $g(b) = c$. Since $f(c) = b$, we get $f(g(b)) = b$ for all $b \in B$, and hence g is 1-1 and onto by 1.2.0.22. $\square$

The above argument makes potentially infinitely many choices, one from each $f_{\leftarrow}(\{b\})$. Of course, these sets are pairwise disjoint.

Pause. Why is it that $f_{\leftarrow}(\{x\}) \cap f_{\leftarrow}(\{y\}) = \emptyset$ if $x \neq y$?◄

Contrast with the case where $B = \{b, b'\}$, a set of two elements. Then we can define g by simply saying

Let $c \in f_{\leftarrow}(\{b\})$, and set $g(b) = c$. Let $c' \in f_{\leftarrow}(\{b'\})$, and set $g(b') = c'$.

We can contain our (two) choices in the space of a proof. The same is true if B had 2^{350000} elements. We would just have to write a proof that would be, well, a bit longer, using a copy of the sentence "Let $y \in f_{\leftarrow}(\{x\})$, and set $g(x) = y$" once for each one of the 2^{350000} members of B that we generically called here "x".

However, the "Let ..." approach does not work for an infinite B, since we cannot contain infinitely many such sentences in the space of a finite-length proof; *unless* we have a way to *codify the infinitely many choices in a finite manner*. For example, if A is a set of natural numbers then so is $f_{\leftarrow}(\{b\})$ for each b and we can say precisely how a c can be chosen in each case: For example, "for each $b \in B$, choose the smallest c in $f_{\leftarrow}(\{b\})$" would do just fine.

Some mathematicians did not accept that one may effect infinitely many choices, *in the absence of* a finitely describable process of how to go about making them; this was not mathematically acceptable. They argued that in the absence of some kind of known "structure" in the various $f_{\leftarrow}(\{b\})$, all the elements of these sets "look the same" and therefore the infinite process of "choosing" cannot be compacted into a finite *well-defined* description. This observation hinges on the number of choices one needs to make rather than on the number of elements in a $f_{\leftarrow}(\{b\})$.

An example of the difficulty, in layman's terms, attributed to Russell, contrasts two cases: One where we have an infinite set of pairs of shoes, and another, where we have an infinite set of pairs of socks.

In the former case we can finitely define infinitely many choices of one shoe per pair by always choosing the *left* shoe in each pair. In the case of socks this "rule" does not define *well* which sock to pick, because, the two socks in a pair have no distinct "left" or "right" members.

I used past tense above, "Some mathematicians did not accept, etc.", for the dissenting opinion. This is because mathematicians nowadays feel comfortable with the notion of effecting infinitely many choices *without having a finite process to describe said choices*. They even have an axiom (the Axiom of Choice, or AC) that says they can do so [for a thorough discussion of AC, see Tourlakis (2003b)].

1.2.0.28 Definition. (Equivalence Relations) Let R be a relation on a set A. We call it an *equivalence relation* iff it has all the three following properties:

(1) It is *reflexive*, that is, xRx holds, for all $x \in A$

(2) It is *symmetric*, that is, xRy implies yRx, for all x and y

(3) It is *transitive*, that is, xRy and yRz imply xRz, for all x, y and z.   □

The concept "equivalence relation" does not apply to relations $R : A \to B$ with $A \neq B$. The concept of reflexivity requires reference to the left (and right, since they are equal) field. If we make the fields larger, without adding any pairs to the relation, a previously reflexive relation will cease being reflexive.

1.2.0.29 Example. The function $1_A : A \to A$ is an equivalence relation.

The relation $<$ on $\mathbb{N}$ is transitive, but neither symmetric, nor reflexive; on the other hand, $\leq$ has reflexivity (still fails symmetry).

The relation R on $\mathbb{Z}$ given by: "xRy iff the difference $x - y$ is divisible by 5" (divisible with 0 remainder, that is) can be easily verified to be an equivalence relation. □

Given an equivalence relation R on a set A, we define for each $x \in A$ the set of all its equivalents in A. This is known as an *equivalence class* of R. We employ the symbol $[x]_R$, thus

$$[x]_R \stackrel{\text{Def}}{=} \{y \in A : xRy\} \tag{1}$$

Despite employing the term "class" in this context, which is standard practice in the literature, we do not imply at all that these classes are "too large" to technically be sets. On the contrary, any such class is a subset of A.

1.2.0.30 Theorem. *Given an equivalence relation R on A. Its equivalence classes $[x]_R$ satisfy*

(1) $[x]_R \neq \emptyset$

(2) *if xRy iff $[x]_R = [y]_R$*

(3) *if $[x]_R \cap [y]_R \neq \emptyset$, then $[x]_R = [y]_R$*

(4) $\bigcup_{x \in A}[x]_R = A$

Proof.

(1) $[x]_R \neq \emptyset$: In fact $x \in [x]_R$ by xRx.

(2) if xRy iff $[x]_R = [y]_R$:

First, *assume the left hand side of the "iff"*, which also yields yRx by symmetry. For the ($\subseteq$) of the right hand side let $z \in [x]_R$. Thus xRz. Transitivity yields yRz, hence $z \in [y]_R$.

For the ($\supseteq$), let $z \in [y]_R$, i.e., yRz. Along with xRy and transitivity we have xRz, that is, $z \in [x]_R$.

Now *assume the right hand side of the "iff"*. By the proof of (1), $y \in [x]_R$, thus xRy.

(3) if $[x]_R \cap [y]_R \neq \emptyset$, then $[x]_R = [y]_R$: By the assumption, there is a z such that $z \in [x]_R$ and $z \in [y]_R$. Thus xRz and yRz, the latter implying zRy. By transitivity, xRy; done by (2).

(4) $\bigcup_{x\in A}[x]_R = A$: The ($\subseteq$) is trivial, since $[x]_R \subseteq A$ for any $x \in A$. For ($\supseteq$), let $x \in A$. But $x \in [x]_R$ and this meets the entrance requirement for x in $\bigcup_{x\in A}[x]_R$. $\square$

1.2.0.31 Remark. (Partitions) Thus the equivalence classes of an R on A meet the three conditions a *partition* of A must satisfy, by definition:

A *partition* on A is *a set of sets P such that*

(a) If $C \in P$, then $C \neq \emptyset$

(b) (Nonoverlap) If C and D are in P and $C \cap D \neq \emptyset$, then $C = D$

(c) (Coverage) $\bigcup S = A$.

So equivalence classes furnish an example of partitions. More is true (cf. Exercise 1.8.33): If P is a partition on A, then an equivalence relation on A can be defined in a natural way, whose equivalence classes are precisely the members of P. $\square$

1.2.0.32 Definition. (Order) A relation R on a set A is called an *order* or *order relation* iff it is transitive and *irreflexive*, the latter meaning $\neg(\exists x)xRx$.

We normally use the abstract symbol $<$ for orders and let the context fend off confusion with concrete usage of the symbol as the order on $\mathbb{N}$ or $\mathbb{R}$. $\square$

We call all orders *partial*, since some orders, $<$ on A, are *total* or *linear*, while others are not.

Indeed, we will seldom use the qualifier "partial" for orders as it is automatically understood. Exception: *Often one presents the "package" consisting of the order* and *the underlying set A together, in symbols $(A, <)$, and calls it a* partially ordered set *or* POset.

That an order $<$ on A is total means that every pair of members x and y of A are *comparable*: That is, one of $x = y$, $x < y$ or $y < x$ holds (this is also know as the *trichotomy* property of linear orders).

1.2.0.33 Example. A standard example of a total order is $<$ on $\mathbb{N}$. A standard example of a nontotal (nonlinear) order is $\subset$ on 2^A. For example, taking as $A = \{0, 1\}$, we see that $\{0\}$ and $\{1\}$ are not comparable under $\subset$. That the latter is an order is trivial to verify (it is irreflexive by definition), a task that we leave to the reader. $\square$

1.3 BIG AND SMALL INFINITE SETS; DIAGONALIZATION

Two broad distinctions of sets by size are *finite* vs. *infinite*. Intuitively, we can count the elements of a finite set and come up with a (natural) number at some distinct point in (future) time. No such possibility is open for infinite sets. Just as finite sets come in various sizes, a 5-element set, vs. a 0-element set, vs. a $2^{3500000}$-element set,

Cantor has taught us that infinite sets also come in various sizes. The *technique* he used to so demonstrate is of interest to us, as it applies to computability, and is the key topic of this section.

1.3.0.34 Definition. (Finite sets) A set A is *finite* iff it is either empty, or is in 1-1 correspondence with $\{x \in \mathbb{N} : x \leq n\}$. We prefer to refer to this "normalized" finite set by the sloppy notation $\{0, \ldots, n\}$.

In this case we say that "A has $n + 1$ elements". If $A = \emptyset$ we say that "A has 0 elements". If a set is *not* finite, then it is *infinite*. $\square$

1.3.0.35 Example. If A and B have $n+1$ elements, then $A \sim B$ (cf. Exercise 1.8.31). $\square$

1.3.0.36 Theorem. *If $X \subset \{0, \ldots, n\}$, then there is no onto function $f : X \to \{0, \ldots, n\}$.*

Proof. First off, the claim holds if $X = \emptyset$, since then $f = \emptyset$ and its range is empty. Let us otherwise proceed by way of contradiction, and assume that it *is* possible to have such onto functions, for some n. Suppose then that the smallest n that allows this to happen is n_0, and let X_0 be a corresponding set "X" that works, that is, we have an onto $f : X_0 \to \{0, \ldots, n_0\}$. Thus $X_0 \neq \emptyset$, by the preceding remark, and therefore $n_0 > 0$, since otherwise $X_0 = \emptyset$.

Let us set $H = f_\leftarrow(\{n_0\})$. $\emptyset \neq H \subseteq X_0$; the $\neq$ by ontoness.

Case 1. $n_0 \in H$. Then $f \restriction (X_0 - H)$ is onto, from $X_0 - H$ to $\{0, \ldots, n_0 - 1\}$—where $X_0 - H \subset \{0, \ldots, n_0 - 1\}$—contradicting minimality of n_0.

Case 2. $n_0 \notin H$. If $n_0 \notin X_0$, then we are back to Case 1. Otherwise, $X_0 - H \not\subset \{0, \ldots, n_0 - 1\}$ and we need a bit more work to get a $Y \subset \{0, \ldots, n_0 - 1\}$, and an onto function from left to right, to get our contradiction.

Well, we first look at the subcase where $f(n_0) \uparrow$: then just ignore n_0; that is, take $Y = X_0 - H - \{n_0\}$. Our function (onto $\{0, \ldots, n_0 - 1\}$) is $f \restriction Y$.

Finally, consider the subcase where $f(n_0) = m$. Take $g = \Big(f - \big(\{\langle n_0, m \rangle\} \cup H \times \{n_0\} \big) \Big) \cup (H \times \{m\})$. Essentially, g is f; except that it ensures that (a) we get no output n_0, (b) $n_0 \notin \text{dom}(g)$, and yet (c) we do obtain output m—to maintain ontoness. Now, taking $Y = X_0 - \{n_0\}$ we see that $g : Y \to \{0, \ldots, n_0 - 1\}$ is onto. $\square$

1.3.0.37 Corollary. (Pigeon-Hole Principle) *If $m < n$, then $\{0, \ldots, m\} \not\sim \{0, \ldots, n\}$.*

Proof. If the conclusion fails then we have an onto $f : \{0, \ldots, m\} \to \{0, \ldots, n\}$, contradicting 1.3.0.36. $\square$

Here is a "quick proof" of 1.3.0.37 that does not utilize 1.3.0.36: Since $A \sim A$ for any non-empty set, $\{0, \ldots, m\}$ has $m + 1$ elements. If $\{0, \ldots, m\} \sim \{0, \ldots, n\}$, then, by 1.3.0.34, it also has $n + 1$ elements. Impossible!

Pause. Do you accept this "proof"?◄

You shouldn't. "A has $n + 1$ elements" is just informal jargon for "$A \sim \{0, \ldots, n\}$". It may well be that this naming was unfortunate, and that it *fails to* uniquely *assign a* number *to a finite set as "the number of its elements"*. That the nomenclature in quotes is apt is the content of Corollary 1.3.0.37, not the other way around.

1.3.0.38 Corollary. *There is no onto function from* $\{0, \ldots, n\}$ *to* $\mathbb{N}$.

"For all $n \in \mathbb{N}$" is, of course, implied (cf. 1.1.1.10).

Proof. Fix an n. By way of contradiction, let $g : \{0, \ldots, n\} \to \mathbb{N}$ be onto. The function f given below is onto from $\mathbb{N}$ to $\{0, \ldots, n+1\}$

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(2) &= 2 \\
&\vdots \\
f(n+1) &= n+1 \\
f(m) &= 0, \text{ for all } m > n+1
\end{aligned}
$$

Thus (cf. Exercise 1.8.34) $(fg) : \{0, \ldots, n\} \to \{0, \ldots, n+1\}$ is onto, contradicting 1.3.0.36. □

Our mathematical definitions have led to what we hoped they would: That $\mathbb{N}$ is infinite!

$\mathbb{N}$ is a "canonical" infinite set, and sets that can be enumerated using natural number indices

$$a_0, a_1, \ldots$$

have a special name.

1.3.0.39 Definition. (Countable Sets) A set A is *countable*, if it is empty or (in the opposite case) if there is a way to arrange all its members in an infinite sequence, in a "row of locations", utilizing one location for each member of $\mathbb{N}$. It *is* allowed to repeatedly list any element of A, so that finite sets are countable. For example, $\{1\}$:

$$1, 1, 1, \ldots$$

Technically, this enumeration is a *total and onto* function $f : \mathbb{N} \to A$. We say that $f(n)$ is the nth element of A in the enumeration f. We often write f_n instead of $f(n)$ and then call n a "subscript" or "index". □

A closely related notion is that of a set that can be enumerated using the elements of $\mathbb{N}$ as indices, but *without repetitions*.

1.3.0.40 Definition. (Enumerable Sets) A set A is *enumerable* iff it is in 1-1 correspondence with $\mathbb{N}$. □

1.3.0.41 Example. Every enumerable set is countable, but the converse fails. For example, $\{1\}$ is countable but not enumerable due to 1.3.0.38. $\{2n : n \in \mathbb{N}\}$ is enumerable, with $f(n) = 2n$ effecting the 1-1 correspondence $f : \mathbb{N} \to \{2n : n \in \mathbb{N}\}$. □

1.3.0.42 Theorem. *If A is an infinite subset of $\mathbb{N}$, then $A \sim \mathbb{N}$.*

Proof. We will build a 1-1 and total enumeration of A, presented in a finite manner as a (pseudo) program below:

$$
\begin{aligned}
X &\quad \leftarrow A \\
n &\quad \leftarrow 0
\end{aligned}
$$
repeat forever:
pick a, **the smallest member of** X
tag a **with** n **as a subscript; print** a_n
$$
\begin{aligned}
n &\quad \leftarrow n + 1 \\
X &\quad \leftarrow X - \{a_n\}
\end{aligned}
$$

Since A in not finite, this process never ends. In particular, all the members of A will be picked (picking always the smallest avoids gaps) and all numbers from $\mathbb{N}$ will be utilized as indices, considering the non-ending nature of the process, the sequential choice of indices, and the starting point $n = 0$. That is, the function $f : \mathbb{N} \to A$, given for all n by $f(n) = a_n$, is total and onto. Since f is strictly increasing—$f(n) < f(n+1)$—it is 1-1 (distinct inputs cause distinct outputs). □
See also Exercise 1.8.35.

1.3.0.43 Theorem. *Every infinite countable set is enumerable.*

Proof. Let $f : \mathbb{N} \to A$ be onto and total, where A is infinite. Let $g : A \to \mathbb{N}$ such that $(fg) = 1_A$ (1.2.0.27). Let us set $B = \mathrm{ran}(g)$. Thus, g is onto B, and by 1.2.0.22 is also 1-1 and total. Thus it is a 1-1 correspondence $g : A \to B$, or $A \sim B$.
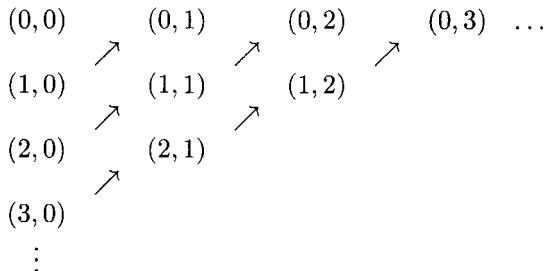
B *must* be infinite, otherwise (1.3.0.34), for some n, $A \sim B \sim \{0, \ldots, n\}$. By transitivity of $\sim$ (Exercise 31), this proves that A is finite, contradicting the hypothesis. Thus, by 1.3.0.42, $A \sim B \sim \mathbb{N}$, hence (again, Exercise 1.8.31) A is enumerable. □

So, if we can enumerate an infinite set at all, then we can enumerate it without repetitions. It is useful to observe that we can convert a multirow enumeration

$$(f_{i,j})_{\text{for all } i,j \text{ in } \mathbb{N}}$$

into a single-row enumeration quite easily. This is shown diagrammatically below. The "linearization" or "unfolding" of the infinite matrix of rows is effected by walking

along the arrows.

$$(0,0) \qquad (0,1) \qquad (0,2) \qquad (0,3) \quad \ldots$$
$$\nearrow \qquad \nearrow \qquad \nearrow$$
$$(1,0) \qquad (1,1) \qquad (1,2)$$
$$\nearrow \qquad \nearrow$$
$$(2,0) \qquad (2,1)$$
$$\nearrow$$
$$(3,0)$$
$$\vdots$$

Technically, the set $\mathbb{N} \times \mathbb{N}$—the set of "double subscripts" (i, j)—is countable. This can be seen by a less informal argument; in fact, $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$:

Perhaps the simplest way to see this is to consider the function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ given by $f(m, n) = 2^m 3^n$. It is clearly total, and (less) clearly 1-1: For the latter just show that

$$2^m 3^n = 2^{m'} 3^{n'} \text{ implies } m = m' \text{ and } n = n'$$

But $\operatorname{ran}(f)$ is infinite (see Exercise 1.8.36). Thus $\mathbb{N} \times \mathbb{N} \sim \operatorname{ran}(f) \sim \mathbb{N}$.

This unfolding of a matrix into a straight line yields a very useful fact regarding strings over countable sets (alphabets):

If the string alphabet V is countable, then the set of all strings *of length* 2 over V is also countable. Why? Because the arbitrary string of length 2 is of the form $d_i d_j$, where d_i and d_j represent the ith and j elements of the enumeration of V, respectively. Unfolding the infinite matrix exactly as above we get a single-row enumeration of these strings.

By induction on the length $n \geq 2$ of strings we see that the set of strings of any length $n \geq 2$ is also countable. Indeed, a string of length $n + 1$ is a string ab, where a has length n and $b \in V$. By the induction hypothesis, the set of all strings a can be arranged in a single row (is countable), and we are done exactly as in the case of the $d_i d_j$ above (think of d_i as an "a" and d_j as a "b").
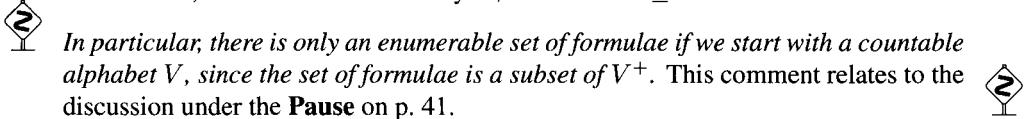
Finally, let us collect *all* the strings over V into a set S. Is S countable? Yes! We can arrange S, at first, into an infinite matrix of strings $m_{i,j}$, that is, the *jth string of length i*. Then we employ our matrix-unfolding trick above.

Given what we understand as a "string" (cf. subsection 1.1.3), the above argument translates as

(1) If V is countable, then so is V^n for any $n \geq 2$.

(2) If V is countable, then so is V^+

With little additional effort one can see that if A and B are countable, then so is $A \times B$ and generalize to the case of $\bigtimes_{i=1}^{n} A_i$.

1.3.0.44 Remark. Let us collect a few more remarks on countable sets here. Suppose now that we start with a countable set A. Is every subset of A countable? Yes, because the composition of onto functions is onto (Exercise 1.8.34). As a special case, if A is countable, then so is $A \cap B$ for any B, since $A \cap B \subseteq A$.

In particular, there is only an enumerable set of formulae if we start with a countable alphabet V, since the set of formulae is a subset of V^+. This comment relates to the discussion under the **Pause** on p. 41.

How about $A \cup B$? If both A and B are countable, then so is $A \cup B$. Indeed, and without inventing a new technique, let

$$a_0, a_1, \ldots$$

be an enumeration of A and

$$b_0, b_1, \ldots$$

for B. Now form an infinite matrix with the A-enumeration as the 1st row, while every other row is the same as the B-enumeration. Now unfold this matrix!

Of course, we may alternatively adapt the unfolding technique to an infinite matrix of two rows.   □

1.3.0.45 Example. Suppose we have a 3×3 matrix

$$\begin{array}{ccc} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{array}$$

and we are asked: Find a sequence of three numbers, *using only* 0 *or* 1, that does not *fit* as a row of the above matrix—i.e., is *different from all rows*.

Sure, you reply: Take 0   0   0.

That is correct. But what if the matrix were big, say, $10^{350000} \times 10^{350000}$, or even *infinite*?

Is there a *finitely describable technique* that can produce an "unfit" row for any square matrix, even an infinite one? Yes, Cantor's *diagonal method* or technique.

He noticed that any row that fits in the matrix as the, say, i-th row, intersects the main diagonal at the same spot that the i-th column does.

Thus if we take the main diagonal—a sequence that has the same length as any row—and *change every one of its entries*, then it will not fit anywhere as a row! Because no row can have an entry that is different than the entry at the location where it intersects the main diagonal!

This idea would give the answer 0   1   0 to our original question. While 1000   11   3 also follows the principle and works, we were constrained by "using only 0 or 1". More seriously, in a case of a very large or infinite matrix it is best to have a simple technique that works even if we do not know much about the elements of the matrix. Read on!   □

1.3.0.46 Example. We have an infinite matrix of 0-1 entries. Can we produce an infinite sequence of 0-1 entries that does not match any row in the matrix? Yes, take the main diagonal and flip every entry (0 to 1; 1 to 0).

If the diagonal has an a in row i, the constructed row has an $1 - a$ in column i, so it will not fit as row i. So it fits nowhere, i being arbitrary.   □

1.3.0.47 Example. (Cantor) Let S denote the set of all infinite sequences of 0s and 1s.

Pause. What is an *infinite sequence*? Our intuitive understanding of the term is captured mathematically by the concept of a total function f with left field (and hence domain) $\mathbb{N}$. The n-th member of the sequence is $f(n)$.◄

Can we arrange *all* of S in an infinite matrix—one element per row? No, since the preceding example shows that we would miss at least one infinite sequence (i.e., we would fail to list it as a row), for a sequence of infinitely many 0s and/or 1s can be found, that does not match any row!

But arranging all members of S as an infinite matrix—one element per row—is tantamount to saying that we can enumerate all the members of S using members of $\mathbb{N}$ as indices.

So we cannot do that. S is not countable!   □

1.3.0.48 Definition. (Uncountable Sets) A set that is not countable is called *uncountable*.   □

So, an uncountable set is neither finite, nor enumerable. The first observation makes it infinite, the second makes it "more infinite" than the set of natural numbers since it is not in 1-1 correspondence with $\mathbb{N}$ (else it would be enumerable, hence countable) nor with a subset of $\mathbb{N}$: If the latter, our uncountable set would be finite or enumerable (which is absurd) according as it is in 1-1 correspondence with a finite subset or an infinite subset (cf. 1.3.0.42 and Exercise 1.8.31).

Example 1.3.0.47 shows that uncountable sets exist. Here is a more interesting one.

1.3.0.49 Example. (Cantor) The set of real numbers in the interval

$$(0, 1] \overset{\text{Def}}{=} \{x \in \mathbb{R} : 0 < x \leq 1\}$$

is uncountable. This is done via an elaboration of the argument in 1.3.0.47.

Think of a member of $(0, 1]$, *in form*, as an infinite sequence of numbers from the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ prefixed with a dot; that is, think of the number's decimal notation.

Some numbers have representations that end in 0s after a certain point. We call these representations *finite*. Every such number has also an "infinite representation" since the non zero digit d immediately to the left of the infinite tail of 0s can be converted to $d - 1$, and the infinite tail into 9s, without changing the value of the number.

Disallow all finite representations.

Assume now by way of contradiction that a listing of all members of $(0, 1]$ exists, listing them via their infinite representations

$$.a_{00}a_{01}a_{02}a_{03}a_{04}\cdots$$
$$.a_{10}a_{11}a_{12}a_{13}a_{14}\cdots$$
$$.a_{20}a_{21}a_{22}a_{23}a_{24}\cdots$$
$$.a_{30}a_{31}a_{32}a_{33}a_{34}\cdots$$
$$\vdots$$

The argument from 1.3.0.47 can be easily modified to get a "row that does not fit", that is, a representation

$$.d_0d_1d_2\cdots$$

not in the listing.

Well, just let

$$d_i = \begin{cases} 2 & \text{if } a_{ii} = 0 \vee a_{ii} = 1 \\ 1 & \text{otherwise} \end{cases}$$

Clearly $.d_0d_1d_2\cdots$ does not fit in any row i as it differs from the expected digit at the i-th decimal place: should be a_{ii}, but $d_i \neq a_{ii}$. It is, on the other hand, an infinite decimal expansion, being devoid of zeros, and thus *should* be listed. This contradiction settles the issue. □

1.3.0.50 Example. (1.3.0.47 Revisited) Consider the set of all total functions from $\mathbb{N}$ to $\{0, 1\}$. Is this countable?

Well, if there is an enumeration of these one-variable functions

$$f_0, f_1, f_2, f_3, \ldots \tag{1}$$

consider the function $g : \mathbb{N} \to \{0, 1\}$ given by $g(x) = 1 - f_x(x)$. Clearly, this *must* appear in the listing (1) since it has the correct left and right fields, and is total.

Too bad! If $g = f_i$ then $g(i) = f_i(i)$. By definition, it is also $1 - f_i(i)$. A contradiction.

This is a "mathematized" version of 1.3.0.47; as already noted, an infinite sequence of 0s and 1s is just a total function from $\mathbb{N}$ to $\{0, 1\}$. □

The same argument as above shows that the set of all functions from $\mathbb{N}$ to itself is uncountable. Taking $g(x) = f_x(x) + 1$ also works here to "systematically change the diagonal" $f_0(0), f_1(1), \ldots$ since we are not constrained to keep the function values in $\{0, 1\}$.

1.3.0.51 Remark. Worth Emphasizing. Here is how we constructed g: We have a list of *in principle available indices* for g. We want to make sure that *none applies*. A convenient method to do that is to inspect each available index, i, and using the diagonal method do this: Ensure that g differs from f_i at input i, setting $g(i) = 1 - f_i(i)$.

This ensures that $g \neq f_i$; period. We say that *we cancelled the index i* as a possible "f-index" of g.

Since the process is applied *for each i, we have cancelled all possible indices for g*: For no i can we have $g = f_i$. $\square$

1.3.0.52 Example. (Cantor) What about the set of all subsets of $\mathbb{N}$ —$\mathcal{P}(\mathbb{N})$ or $2^{\mathbb{N}}$?

Cantor showed that this is uncountable as well: If not, we have an enumeration of its members as

$$S_0, S_1, S_2, \ldots \tag{1}$$

Define the set

$$D \overset{\text{Def}}{=} \{x \in \mathbb{N} : x \notin S_x\} \tag{2}$$

So, $D \subseteq \mathbb{N}$, thus it must appear in the list (1) as an S_i. But then $i \in D$ iff $i \in S_i$ by virtue of $D = S_i$. However, also $i \in D$ iff $i \notin S_i$ by (2). This contradiction establishes that $2^{\mathbb{N}}$ is uncountable.

In particular, it establishes that D is not an S_i. $\square$

1.3.0.53 Example. (Characteristic functions) First a definition: Given a set S in the context of a reference set $\mathbf{U}$, the characteristic function of S, denoted by χ_S, is given by

$$\chi_S(x) = \begin{cases} 0 & \textbf{if } x \in S \\ 1 & \textbf{if } x \in \overline{S} \end{cases}$$

If the reference set is $\mathbb{N}$, the characteristic function of $S \subseteq \mathbb{N}$ is

$$\chi_S(x) = \begin{cases} 0 & \textbf{if } x \in S \\ 1 & \textbf{if } x \in \mathbb{N} - S \end{cases}$$

Note that there is a 1-1 correspondence F between subsets of $\mathbb{N}$ and 0-1-valued functions from $\mathbb{N}$ simply given by $F(S) = \chi_S$—cf. Exercise 1.8.37. Thus

The set of 0-1-valued functions from $\mathbb{N}$ is in 1-1 correspondence with $\mathcal{P}(\mathbb{N})$

In particular, the concept of characteristic functions shows that Example 1.3.0.52 fits the diagonalization methodology. Indeed, $\chi_D(x) = 1 - \chi_{S_x}(x)$ for all x. In other words, χ_D is nothing else but the *altered* "main diagonal" (in bold face type) of the infinite matrix

$$\begin{matrix} \mathbf{S_0(0)} & S_0(1) & S_0(2) & S_0(3) & \ldots \\ S_1(0) & \mathbf{S_1(1)} & S_1(2) & S_1(3) & \ldots \\ S_2(0) & S_2(1) & \mathbf{S_2(2)} & S_2(3) & \ldots \\ \vdots & \vdots & \vdots & \vdots & \end{matrix}$$

$\square$

1.3.0.54 Example. By Exercise 1.8.38 we have that $2^{\mathbb{N} \times \mathbb{N}} \sim 2^{\mathbb{N}}$ so that

The set of all subsets of $\mathbb{N} \times \mathbb{N}$ is uncountable

The above can be rephrased to

The set of all binary relations on $\mathbb{N}$ is uncountable

Thus, if we build our formulae with symbols out of a countable alphabet, then we *do not have enough symbols* to represent (in our notation) *all* binary relations on $\mathbb{N}$ by formulae. This observation concludes our discussion that started on p. 41, following Definition 1.2.0.4 and continued in 1.3.0.44. □

1.3.0.55 Example. (Russell's Paradox is a Diagonalization) Russell formed the collection of sets x given as

$$R = \{x : x \notin x\} \tag{1}$$

He argued that it is contradictory to accept R as a *set*: For if it is, and given that (1) is equivalent to the statement (for all sets x)

$$x \in R \equiv x \notin x \tag{2}$$

we can substitute the specific set R into the set variable x to obtain—from the truth of (2)—the truth of the special case

$$R \in R \equiv R \notin R$$

This, of course, is absurd!

Let us now argue *intuitively*—taking liberties with working with *all sets* at once!—that the above argument is *a diagonalization over all sets*.

Imagine an infinite matrix, M, whose columns and rows are labeled by *all* sets, arranged in the same order along rows and columns. Assume that the matrix has as entries only the numbers 0 and 1, entered such that in the location determined by the row (named) x and the column (named) y we have a 0 iff $y \in x$ is true (we have 1 otherwise). That is,

$$y \in x \text{ iff } M(x, y) = 0 \tag{1}$$

It follows that *each row represents a set as an array of 0s and 1s*—that is, as the set's *characteristic function*.[43]

Thus, the partial depiction of the row for set a informs us that the following are true: $a \notin a$, $b \in a$ and $x \notin a$. Indeed, any array, X, of 0s and 1s whose entries are *labeled by the column names* represents a *collection* of sets that has y as a member iff the y-th entry of X is 0. For example, the diagonal collection
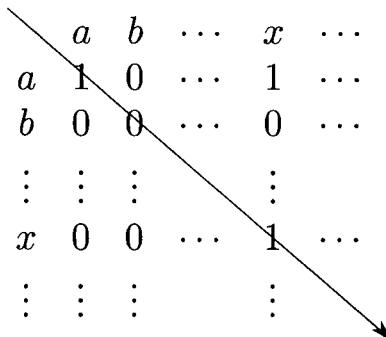
$$
\begin{array}{ccc}
a\, b & & x \\
\downarrow\downarrow & & \downarrow \\
d = 1\, 0 \cdots & & 1 \cdots
\end{array}
$$

---

[43] Recall that this is an intuitive argument showing the (ironic) indebtedness of Russell's argument to Cantor's original diagonalization method. Thus we will not be splitting hairs about qualms such as: "Hmm, is this characteristic function defined over the collection of *all sets*? Can we do that?" Yes, because this is only a qualitative argument to tease out the diagonal argument that was hidden in Russell's proof.

contains b, but neither a, nor x.

The matrix M

$$
\begin{array}{c|cccccc}
 & a & b & \cdots & x & \cdots \\
\hline
a & 1 & 0 & \cdots & 1 & \cdots \\
b & 0 & 0 & \cdots & 0 & \cdots \\
\vdots & \vdots & \vdots & & \vdots & \\
x & 0 & 0 & \cdots & 1 & \cdots \\
\vdots & \vdots & \vdots & & \vdots & \\
\end{array}
$$

Let us do Cantor's trick now: We take the main diagonal d and form an array $\widetilde{d}$ from it, by swapping *all* 1s with 0s. This $\widetilde{d}$ cannot fit as a row anywhere in the matrix M since it will disagree at the diagonal entry in any placement.

The fact that the collection of sets (named) $\widetilde{d}$ does *not* fit as a row of M means that *it is not a set*—because all sets are accounted for as row labels in M!

But which collection does $\widetilde{d}$ represent?

Well, using the analogy of X above, y is in $\widetilde{d}$ iff the y-th entry of $\widetilde{d}$ is 0 iff the y-th entry of d is 1 iff $y \notin y$. Thus,

$$\widetilde{d} = R, \text{ Russell's "paradoxical collection"}$$

□

1.4 INDUCTION FROM A USER'S PERSPECTIVE

In this section we will review the two widely used forms of induction, *complete* (or strong) induction (also called *course-of-values* induction) and *simple* induction. We will see how they are utilized, and when one is more convenient than the other; relate them to each other, but also to another principle that is valid on natural numbers, the *least (integer) principle*.

1.4.1 Complete, or Course-of-Values, Induction

Suppose that $\mathscr{P}(n)$ is a "property"—that is, a *formula* of one free variable, n—of the natural number n. To prove that $\mathscr{P}(n)$ holds *for all $n \in \mathbb{N}$* it suffices to prove *for the arbitrary n* that $\mathscr{P}(n)$ holds.

What we mean by "arbitrary" is that we do not offer the proof of $\mathscr{P}(n)$ for some specific n such as $n = 42$; or n even; or any n that has precisely 105 digits, etc. If the proof indeed has not cheated by using some property of n beyond the generic

"$n \in \mathbb{N}$", then our proof is *equally valid for any* $n \in \mathbb{N}$; we have succeeded in effect to prove $\mathscr{P}(n)$, *for all* $n \in \mathbb{N}$ (cf. 1.1.1.10 and 1.1.1.15).

1.4.1.1 Example. Suppose $\mathscr{P}(n)$ stands for the statement

$$0 + 1 + 2 + 3 + \cdots + 2n = n(2n + 1) \tag{1}$$

One way to prove (1), for all n, is as follows: Fix, but *do not specify*, n—that lack of specification makes it *arbitrary*. Note the pairs below—separated by semicolons—each consisting of two numbers that are equidistant from the two ends of the sequence $0, 1, 2, 3, \ldots, 2n$

$$0, 2n; 1, 2n - 1; 2, 2n - 2; \ldots; n, 2n - n$$

The above sequence is (almost) a permutation of the sequence $0, 1, 2, 3, \ldots, 2n$, hence the sum of its terms is the same as the left hand side of (1), plus n.

Pause. Why "plus n"?◄

We have $n + 1$ pairs, the sum of each being $2n$, thus the left hand side of (1) equals $(n + 1)2n - n$. An easy calculation shows that $(n + 1)2n - n = n(2n + 1)$.   □

Now the above endeavor—proving some $\mathscr{P}(n)$ for the arbitrary n—is not always easy. *In fact, the above proof—attributed to Gauss—had a rabbit-off-a-hat flavor.* It would probably come as a surprise to the uninitiated that *we can pull an extra assumption out of the blue* and use it toward proving $\mathscr{P}(n)$, and not only that: When all is said and done, this process, *with* the extra assumption, is as good as if we have proved $\mathscr{P}(n)$ *without* the extra assumption!

This out-of-the-blue assumption is that

$$\mathscr{P}(k) \text{ holds for all } k < n \tag{I}$$

or, put another way, that *the history*, or the *course-of-values*, of $\mathscr{P}(n)$, namely,

$$\mathscr{P}(0), \mathscr{P}(1), \ldots, \mathscr{P}(n - 1) \tag{II}$$

holds—that is, it is a sequence of valid statements.

The extra assumption, (I) or (II), goes by the name *induction hypothesis* (I.H.) The technique of proving

$$\text{for all } n, \text{ we have that } \mathscr{P}(n) \text{ holds} \tag{2}$$

using an I.H. toward the proof, is called *proof by strong (complete, course-of-values) induction.*

The *application* (technique) of the proof by strong induction is:

(a) Pick an arbitrary n and prove the validity of $\mathscr{P}(n)$ having *also assumed the validity of (I) or (II).*

(b) Once step (a) is completed, we conclude (2).

We note that the history, (II), of $\mathscr{P}(n)$ is *empty* if $n = 0$. Thus every proof by strong induction has two cases to consider: the one where the history helps, *because it exists*, i.e., when we have $n > 0$; and the one where the history *does not help*, because it simply does not exist, i.e., when $n = 0$.

Thus, the application of strong induction morphs into a two-step method:

(A) Pick an arbitrary $n > 0$ and prove the validity of $\mathscr{P}(n)$, *having assumed the validity of (I) or (II)*.

(B) For $n = 0$ prove $\mathscr{P}(n)$—i.e., $\mathscr{P}(0)$—directly.

(C) Once steps (A) and (B) are completed, we conclude (2).

Some jargon: As we noted, (I) or (II) are called the I.H. Step (A) above is called the *Induction Step* (I.S.). Step (B) is called the *basis* of the induction. The process (A)–(C) is *proof by induction on n*.

One often sees the basis done first, but it should be clear that it is just one of two cases to be considered, and the cases can be taken care of in any order.

It cannot be emphasized enough that the phrase "Pick an arbitrary $n > 0$ and prove…" is synonymous with "*Fix*, but do not specify, an $n > 0$ and prove …"

Clearly the I.H. is for a *fixed but unspecified n—not* for all n, as the latter would beg the very question we are called to settle by induction!

1.4.1.2 Example. (Example 1.4.1.1 Revisited) We will prove (1) above, for all n, by strong induction, faithfully following the plan (A)–(B) above. Fix an arbitrary n. We have two cases:

Case $n > 0$. We assume the I.H. and try to prove (1). Well, we calculate as follows:

$$
\begin{aligned}
0 + 1 + 2 + \cdots + 2(n-1) + 2n - 1 + 2n &= (n-1)\big(2(n-1) + 1\big) + 4n - 1 \\
&= (n-1)(2n-1) + 4n - 1 \\
&= 2n^2 - n - 2n + 1 + 4n - 1 \\
&= 2n^2 + n \\
&= n(2n+1)
\end{aligned}
$$

Note that the I.H. says

$$0 + 1 + 2 + 3 + \cdots + 2k = k(2k+1), \text{ for } k < n \tag{3}$$

This, in particular, is true for $k = n - 1$, a fact we have used in the first calculation step above.

Case $n = 0$. In this case the statement to prove, namely, (1), becomes $0 = 0$, which is true. $\qquad\square$

Hm. The I.H. (3) above seems to be an overkill given that only the case $k = n - 1$ was utilized in the I.S. Good point! We take it up in the next example.

1.4.1.3 Example. This time we prove, for all n,

$$\text{if } n > 1, \text{ then } n \text{ has a prime factor} \tag{4}$$

The reader will recall that a *factor* of n is a natural number m such that for some natural number k we have $n = mk$. A natural number p is a *prime* number (or just a prime) if and only if it is greater than 1, and all its factors are p and 1.

By strong induction, we take up first the case for an arbitrary (but fixed) n that allows a non-empty history; thus we assume the I.H. corresponding to (4):

$$\text{for all } k < n, \text{ if } k > 1, \text{ then } k \text{ has a prime factor} \tag{$I.H.$}$$

The non-empty history case corresponds to $n \geq 3$, since $1 < k$ *and* $k < 2$ are inconsistent.

Let then $n \geq 3$. If n is a prime, then we are done (n is a factor of n). Alternatively, suppose that it is not. Then *there exist* a and b such that $n = ab$, where $a \neq 1 \neq b$— else n would be prime! Can $a < 1$? No, for then $a = 0$ and hence $n = 0$, contrary to the case we are in. Thus $a > 1$. Similarly $b > 1$. The latter yields $n = ab > a$.

Therefore the I.H. applies to a, that is, a has a prime factor, p. This means that for some m, $a = pm$. But then, $n = pmb$, and hence n has a prime factor.

The "basis" encompasses all the cases that have empty history: $n = 0, 1, 2$. For the first two the claim is vacuously satisfied as $n > 1$ is false. For $n = 2$ it is satisfied by virtue of 2 being a prime. □

This example shows the value of an I.H. that refers to the entire history below n: We have no way of controlling where a falls in the sequence $0, 1, 2, \ldots, n - 1$. It is unreasonable to expect that $a = n - 1$ in general. For example, if $n = 6$, then $a = 2$ and $b = 3$, or $a = 3$ and $b = 2$. But $n - 1 = 5$.

1.4.2   Simple Induction

Since on occasion we will also employ *simple* induction in this book, let me remind the reader that in this kind of induction the I.H. is not the assumption of validity of the entire history, but that of just $\mathscr{P}(n - 1)$. As before, simple induction is carried out for the arbitrary n, so we need to work out two cases: when the I.H. exists ($n > 0$) and when it does not ($n = 0$). The case of proving $\mathscr{P}(0)$ directly is still called the *basis* of the (simple) induction.

The reader will notice that Example 1.4.1.2 can be recast under a simple induction proof since in the first step of the $n > 0$ case we only have used the assumption that (1) is true for $k = n - 1$.

Common practice has it that in performing simple induction the majority of users in the literature take as I.H. $\mathscr{P}(n)$ while the I.S. involves proving $\mathscr{P}(n + 1)$.

1.4.3 The Least Principle

The least principle states that each non-empty subset of natural numbers contains a smallest (least) number.

1.4.3.1 Example. (Euclid) We prove that given a natural number $b > 1$, each natural number n can be expressed as $n = bq + r$ for some natural numbers q and r , where $0 \le r < b$. *We only argue the case $n > 0$ since the case $n = 0$ is trivial:* $n = 0b + 0$.

So let $n > 0$. Note that the set $S = \{bx - n : x \in \mathbb{N} \wedge bx - n > 0\}$ is not empty. For example, since $bn > n$ (by $b > 1$), it is $bn - n > 0$. By the least principle, S contains a smallest number, which has the form $bm - n$ for some m. From $m \neq 0$ (since $-n$ is not positive and cannot be in S) we get $q = m - 1 \in \mathbb{N}$. Since $bq - n < bm - n$, it is $bq - n \notin S$. Thus $bq - n \le 0$, i.e., $n - bq \ge 0$.

We set $r = n - bq$. Now, since $bq \le n < b(q + 1)$ (recall, $m = q + 1$) we have $0 \le n - bq < b(q + 1) - bq$, that is, $0 \le r < b$. $\qquad\square$

A related result that does not need the least principle (nor induction) is that the *quotient* q and *remainder* r are uniquely determined by n and b: Indeed, suppose that we have

$$n = bq' + r' \tag{5}$$

$$0 \le r' < b \tag{6}$$

$$n = bq'' + r'' \tag{7}$$

$$0 \le r'' < b \tag{8}$$

By (5) and (7) we have

$$b|q' - q''| = |r' - r''| \tag{9}$$

Can it be that $|q' - q''| \neq 0$? If so, $|q' - q''| \ge 1$, hence, multiplying both sides by b and using (9),

$$|r' - r''| \ge b \tag{10}$$

(6) and (8) tell a different story though! They yield [e.g., think of (8) as $-b < -r'' \le 0$ and add with (6), term by term] $-b < r' - r'' < b$, that is $|r' - r''| < b$, which contradicts (10).

We thus must answer the earlier question "Can it be that $|q' - q''| \neq 0$?" by "no". But then (9) yields also $r' = r''$, as needed.

1.4.4 The Equivalence of Induction and the Least Principle

Somewhat surprisingly, all three proof techniques, by least principle, by simple or by course-of-values induction, have exactly the same power.

1.4.4.1 Theorem. *The least principle is equivalent to course-of-values induction.*

Proof. This proof requires two directions.

One, we can prove the least principle, *using strong induction*: Indeed, let $\emptyset \neq S \subseteq \mathbb{N}$. We will argue, by way of contradiction, that S has a least element.

So let instead S have *no* such element. The plan is to use strong induction to arrive at a contradiction. We may encounter more than one such contradictions, but the "primary" one that we will strive for is to prove that $S = \emptyset$—contrary to hypothesis—which is tantamount to $\mathbb{N} = \mathbb{N} - S$, or in many words:

$$\text{for all natural numbers } n, n \in \mathbb{N} - S \tag{1}$$

For the basis, we argue that $0 \in \mathbb{N} - S$. Indeed, if not, then $0 \in S$ will be least in S, contradicting what we assumed for S. Let then pick an $n > 0$ and accept as I.H. that for all $k < n$ we have $k \in \mathbb{N} - S$. It immediately follows that $n \in \mathbb{N} - S$ for otherwise it is the first n to enter S, which makes it least in S! We have proved (1).

Two, we prove that strong induction is valid, by *assuming that the least principle is*. That is, we will show the following, for any property $\mathscr{P}(n)$:

If $\mathscr{P}(0)$ holds, and if, for any $n > 0$, $\mathscr{P}(n)$ holds whenever all of $\mathscr{P}(0), \ldots, \mathscr{P}(n-1)$ hold; then $\mathscr{P}(n)$ holds for all n.

So we assume that the if-part of the italicized statement above is valid and prove the then-part, that "$\mathscr{P}(n)$ holds for all n".

Well, *assume we are wrong* in our conjectured conclusion (then-part). But then

$$S = \{n : \neg \mathscr{P}(n)\} \text{ is not empty.}$$

By the least principle, we have a smallest member of S, let us call it m. Now, $m \neq 0$, since the italicized statement's if-part includes the validity of $\mathscr{P}(0)$. What about $0, 1, 2, \ldots, m - 1$ then? (Now that we know that $m - 1 \geq 0$, we may ask.) Well, none are in S (all being smaller than m), that is, they all satisfy $\mathscr{P}$. But then, the if-part of the italicized statement guarantees that $\mathscr{P}(m)$ must hold as well. This is no good because it says $m \notin S$!

This contradiction forces us to *backtrack over* our "*assume we are wrong*" above. So it is, after all, the case that $\mathscr{P}(n)$ *does* hold for all n. $\square$

1.4.4.2 Theorem. *Simple induction and course-of-values induction have the same power.*

Proof. That is, one tool can simulate the other. We need to prove two things:

One, whatever property $\mathscr{P}(n)$ we can prove (for all n) via simple induction, we can also prove it *using strong induction*. Simple induction achieves this:

If $\mathscr{P}(0)$ holds, and if, for any $n > 0$, $\mathscr{P}(n)$ holds whenever $\mathscr{P}(n - 1)$ holds; then $\mathscr{P}(n)$ holds for all n.

So *assume the if-part of the italicized statement*. Can *course-of-values induction* prove the then-part, namely, that "$\mathscr{P}(n)$ holds for all n"?

Well, strong induction will have to check that $\mathscr{P}(0)$ holds: That much is given by the if-part above. Now, for the arbitrary $n > 0$, strong induction's I.H. is that $\mathscr{P}(0), \ldots, \mathscr{P}(n - 1)$ all hold. Can this assumption produce the truth of $\mathscr{P}(n)$? *Yes,*

because this strong I.H. yields the truth of $\mathscr{P}(n-1)$. By the if-part of the italicized statement above, this alone yields the truth of $\mathscr{P}(n)$.

Now, by strong induction, we indeed get the *then-part*: $\mathscr{P}(n)$ holds *for all n*.

Two, conversely, we prove that strong induction is valid, by *assuming that simple induction is*. That is, we will show that the following statement is valid, for any property $\mathscr{P}(n)$:

> If $\mathscr{P}(n)$ holds on the assumption that, for all $k < n$, $\mathscr{P}(k)$ holds;
> then $\mathscr{P}(n)$ holds for all n. $\hspace{2cm}$ (2)

So we will *assume* the validity of if-part of (2), and then *employ simple induction* to *prove* the then-part, that

$$\mathscr{P}(n) \text{ holds for all } n \hspace{2cm} (3)$$

We will be a bit trickier here, so let us consider the new property $\mathscr{Q}(m)$ defined as follows:

$$\text{for all } k < m, \mathscr{P}(k) \text{ holds} \hspace{2cm} (4)$$

So, instead of directly proving (3),

$$\text{I will prove that, for all } n \in \mathbb{N}, \mathscr{Q}(n) \text{ holds} \hspace{2cm} (5)$$

I deliver on the promise (5) by simple induction, which, by assumption, is *the* tool at my disposal in this part of the proof: First, by (4), $\mathscr{Q}(0)$ says "for all $k < 0$, $\mathscr{P}(k)$ holds". I need to verify this, my (simple) induction's basis. Fortunately, the statement in quotes is *vacuously true* since it is impossible to refute it since a refutation requires a $k < 0$ [that makes $\mathscr{P}(k)$ false].

Next, let us fix an n and take the I.H. that $\mathscr{Q}(n)$ is true. We proceed to show that $\mathscr{Q}(n+1)$ is true too, and this will conclude (5). Now, $\mathscr{Q}(n+1)$ says "for all $k < n+1$, $\mathscr{P}(k)$ holds", or, "for all $k \leq n$, $\mathscr{P}(k)$ holds". Another way of putting it is: for all $k < n$, *and* for $k = n$, $\mathscr{P}(k)$ holds. That is, we want to show that

$$\mathscr{Q}(n) \text{ and } \mathscr{P}(n) \text{ hold} \hspace{2cm} (6)$$

Now $\mathscr{Q}(n)$ is true by the I.H. of our simple induction. That is, for all $k < n$, $\mathscr{P}(k)$ is true, by the definition of $\mathscr{Q}$ in (4). But we have assumed the if-part of (2), and this yields the truth of $\mathscr{P}(n)$. Thus (6) is established, i.e., $\mathscr{Q}(n+1)$, is true. Hence we have concluded (5). Having moreover just seen that $\mathscr{Q}(n)$ implies $\mathscr{P}(n)$, for any n, (5) implies that $\mathscr{P}(n)$ too holds for all n—and this statement is (3). $\hspace{1cm}\square$

At this point we can "strengthen" our inductions to "start" (basis) at any integer $n_0 > 0$.

Simple induction with non zero basis: To prove that, for all $n \geq n_0$, $\mathscr{P}(n)$ holds just do:

(A) Prove the truth of $\mathscr{P}(n_0)$.

(B) Fix an arbitrary $n \geq n_0$ and prove the truth of $\mathscr{P}(n+1)$ on the assumption that $\mathscr{P}(n)$ holds.

Strong induction with non zero basis: To prove that, for all $n \geq n_0$, $\mathscr{P}(n)$ holds just do:

(a) Prove the truth of $\mathscr{P}(n_0)$.

(b) Fix an arbitrary $n > n_0$ and prove the truth of $\mathscr{P}(n)$ on the assumption that $\mathscr{P}(k)$ holds for all $n_0 \leq k < n$.

1.4.4.3 Exercise. Start by the trivial observation that the least principle holds on the set $\mathbb{N}_{n_0} = \{n_0, n_0 + 1, n_0 + 2, \ldots\}$, namely: *Every non-empty subset of* $\mathbb{N}_{n_0}$ *has a least element.* Now modify the proof of 1.4.4.1 (using $\mathbb{N}_{n_0}$ instead of $\mathbb{N}$, judiciously) to conclude that the proof schema (a)–(b) above is equivalent to the least principle on the set $\mathbb{N}_{n_0}$.

Conclude that the proof schema (a)–(b) is valid.   □

1.4.4.4 Exercise. Imitate the proof of 1.4.4.2 to prove that the schemata (a)–(b) and (A)–(B) above are equivalent in power.

Conclude that the proof schema (A)–(B) is valid.   □

1.5   WHY INDUCTION TICKS

Induction is neat, but is it a valid principle? Why should we believe such a thing? Unfortunately, the previous section does not shed much light other than the somewhat surprising equivalence of the two induction principles with the least principle.

It turns out that we cannot prove either of the three as valid from any substantially simpler and therefore more readily believable facts of arithmetic. We can build a *plausible case*, however. Given the equivalence of the three, let us use simple induction as the pivot of our plausibility argument.

Simple induction is, intuitively, a proof generator that, for each given property $\mathscr{P}(n)$, certifies the latter's validity for any n that we want: Recall that the combination of the I.H. and I.S. establish *for the arbitrary* n, that *if* $\mathscr{P}(n)$ *is valid, then so is* $\mathscr{P}(n+1)$. Thus given the starting point, that is, the validity of $\mathscr{P}(0)$, we can certify the validity of $\mathscr{P}(1)$. And then of $\mathscr{P}(2)$. If we repeat this process—of inferring the truth of $\mathscr{P}(n+1)$ from that of $\mathscr{P}(n)$—for $n = 0, 1, 2, 3, \ldots, k-1$, for any k that we desire, then we will obtain the validity of $\mathscr{P}(k)$ (in k steps).

Imagine the process running for ever. Then the truth of $\mathscr{P}(n)$, for $n = 0, 1, 2, \ldots$ is established!

This argument is quite plausible, but glosses over two things: A mathematical proof has *finite length* so it cannot be an *infinite process* running for for $n = 0, 1, 2, \ldots$ Moreover, we must be sure that "for all $n \in \mathbb{N}$" really means *the same thing as* "for $n = 0, 1, 2, 3, \ldots$", or that $\mathbb{N}$ is *the smallest set around* with the properties[44]

(a) it contains 0

---

[44]Some bigger sets that have the properties (a) and (b) include $\mathbb{Z}$, the set of all integers; $\mathbb{Q}$, the set of all rational numbers; $\mathbb{R}$, the set of all real numbers; and more.

(b) if it contains n, it also contains $n + 1$.

By the way, by "smallest" we mean that any other set T with the properties will satisfy $\mathbb{N} \subseteq T$.

Hm. This sounds right! $\mathbb{N}$ *is* the smallest set there is that satisfies (a) and (b), is it not? And if we are content with that, then here is a "real" proof of the simple induction principle, one that has *finite length*!

Pick any property $\mathscr{P}(n)$ and assume that we have performed the steps of simple induction, that is, we have already proved that

(A) $\mathscr{P}(0)$ is true.

(B) On the I.H. that $\mathscr{P}(n)$ is true we have proved that $\mathscr{P}(n + 1)$ is true too.

Now let us form the set $S = \{n : \mathscr{P}(n)\}$. By (A), we have that $0 \in S$—that is, S satisfies (a) above. By (B), if $n \in S$, then also $n + 1 \in S$—again, S satisfies (b) above. Since $\mathbb{N}$ is the smallest that satisfies (a) and (b), we have $\mathbb{N} \subseteq S$. That is, for all $n \in \mathbb{N}$ we have $n \in S$. Expressing this in terms of $\mathscr{P}(n)$ we have

$$\text{for all } n \in \mathbb{N}, \mathscr{P}(n) \text{ holds} \tag{1}$$

That is, performing successfully the steps of simple induction—(A) and (B)—on $\mathscr{P}(n)$ we have succeeded in obtaining (1) as induction promises. Induction works!

Not so fast. Let us pick any set R that satisfies (a) and (b) above. I will show by induction that

$$\text{for all } n \in \mathbb{N}, n \in R \text{ holds} \tag{2}$$

Well, the basis $0 \in R$ is satisfied, since R obeys (a). Let us fix an n now and take the I.H. $n \in R$. But, because R obeys (b), we will also have $n + 1 \in R$. By simple induction, we have proved (2). But that says $\mathbb{N} \subseteq R$. Since R was *arbitrary* we have used induction to prove that $\mathbb{N}$ is the smallest set satisfying (a) and (b).

Thus the validity of induction and the just stated property of $\mathbb{N}$ are equivalent principles and we are back to square one: We have *not* succeeded in providing a proof of the validity of induction that is based *on more primitive, non equivalent to induction*, principles.

However, it is expected that our discussion brought some degree of comfort to the reader about the plausibility (and naturalness) of the induction principle! Mathematicians have long ago stopped worrying about this, and have adopted the induction principle as one of the starting points, i.e., nonlogical axioms, of (Peano) arithmetic.

1.6   INDUCTIVELY DEFINED SETS

One frequently encounters *inductive*—or, as they are increasingly frequently called, *recursive*—definitions of sets. This starts like this: Suppose that we start with the alphabet $\{0, 1\}$ and want to build strings as follows: We want to include ϵ, the empty

string. We also want the *rule* or *operation* that asks us to include $0A1$ if we know that the string A is included. So, some strings we might include are $\epsilon, 01, 0011$ and 001. The first was included outright, while the second and third are justified by the rule, via the presence of ϵ and 01, respectively. The last one would be legitimate if we knew that 0 was included. But is it? That is not a fair question. It becomes fair if we consider the *smallest*—with respect to inclusion $\subseteq$—set of strings that we can build, by including ϵ and repeatedly applying the rule. Then it can be proved that neither 0 nor 001 can be included in this smallest set.

There are several examples in mathematics and theoretical computer science of "smallest" sets defined from some start-up objects via a set of operations or rules whose application on existing objects yields new ones to include. Another one is the set of terms, formulae and proofs of logic. Further down we will encounter more examples such as the set of partial recursive and primitive recursive functions. But why look that far: Perhaps the simplest such smallest set built from initial objects and the application of operations is $\mathbb{N}$, as we have noted already: the initial object is 0 and the operation is " $+ \, 1$", the *successor function*.

The purpose of this section is to offer some unifying definitions and discuss their connection to each other.

1.6.0.5 Definition. (Operations) An n-ary *operation* or *rule* is a (binary) relation R such that whenever aRb, then a is an n-tuple. We will write $R(a_1, \ldots, a_n, b)$ rather than $R(\langle a_1, \ldots, a_n \rangle, b)$ or $\langle a_1, \ldots, a_n \rangle Rb$. We will call the sequence of objects $a_1, \ldots, a_n$ inputs, and the object b an *output*, or a *result* of R applied to the listed inputs.

It is *not* required that the relation be single-valued in its outputs. $\qquad\square$

1.6.0.6 Definition. (Derivations) Given a set of objects, $\mathcal{I}$—the *initial* objects—and a set of operations $\mathcal{O}$. An $(\mathcal{I}, \mathcal{O})$-derivation, or just *derivation* if the context makes clear which $\mathcal{I}$ and $\mathcal{O}$ we have in mind, is a finite sequence of objects, $a_1, \ldots, a_n$ such that *every* a_i is one of

(1) a member of $\mathcal{I}$

(2) a result of some k-ary operation, from the set $\mathcal{O}$, applied on k inputs among the a_j that appear *before* a_i in the sequence —i.e., $j < i$ for all such inputs a_j.

We call the number n the *length of the derivation*. $\qquad\square$

Since the legitimacy of any a_i in a derivation never depends on a a_k with $k > i$, it is clear that if $a_1, \ldots, a_r, \ldots, a_n$ is a derivation, then so is $a_1, \ldots, a_r$.

Note also that nowhere does the definition ask that the a_i be distinct. Indeed, once an a_i is placed as the i-th element, for the first time, it can be placed again thereafter—as $a_j = a_i$, with $j > i$—any number of times we wish. The same reason of legitimacy that applied originally to a_i still applies to all the additional placements a_j.

1.6.0.7 Example. Let $\mathcal{I} = \{0\}$ and $\mathcal{O}$ contain just the relation (given in atomic formula form) $x + 1 = y$, with y being the output variable. Then the reader can readily verify that the sequences

$$0, 0, 1, 2, 2, 2, 3, 0, 0, 4$$

and

$$0, 1, 2, 3, 4$$

are derivations. $\qquad\qquad\square$

1.6.0.8 Example. Let $\mathcal{I} = \{0\}$ and $\mathcal{O}$ contain just the relation (given in atomic formula form) $x + 1 = y$, this time x being the output variable. Then the reader can readily verify that the sequences

$$0, 0, -1, -2, -2, -2, -3, 0, 0, -4$$

and

$$0, -1, -2, -3, -4, -5, -6$$

are derivations. $\qquad\qquad\square$

1.6.0.9 Definition. A set S is *built by steps* from a set of initial objects, $\mathcal{I}$, and a set of operations $\mathcal{O}$ as follows: $S = \{a : a$ appears in some $(\mathcal{I}, \mathcal{O})$-derivation$\}$. $\qquad\square$

If S is a set built by steps, then we can prove properties of its members by induction on the lengths of their derivations.

1.6.0.10 Example. Given the alphabet $\{0, 1\}$. Let $\mathcal{I} = \{\epsilon\}$, while $\mathcal{O}$ contains just the operation on strings $0x1 = y$—x being the input and y the output variables. We will show that the set S built by steps from the given pair $(\mathcal{I}, \mathcal{O})$ is $\{0^n 1^n : n \geq 0\}$, where, for any string A, and $n > 0$, A^n means

$$\underbrace{AA \cdots A}_{n \text{ copies of } A}$$

while A^0 means ϵ.

We have two directions to establish set equality:

$\subseteq$. For any $a \in S$ we do induction on its derivation length to show $a = 0^m 1^m$ for some m. If the length is 1, then it can only contain ϵ (initial object). Thus $a = 0^0 1^0$. We take as I.H. the truth of the claim when a is in a derivation of length $< n$.

For the I.H., suppose that a has a derivation, $a_1, \ldots, a_n$.

If $a = a_i$ with $i < n$, then since $a_1, \ldots, a_i$ is a shorter derivation, we are done by the I.H. If $a = a_n$ we have two cases: One, a is initial. This has already been dealt with. Two, $a = 0a_i 1$, for some $i < n$.

By the I.H. $a_i = 0^m 1^m$. Thus, a has the same form.

$\supseteq$.   For any $n \geq 0$ we prove that $0^n 1^n$ *must* appear in some derivation. This is done by (simple) induction on n. For $n = 0$ (basis) $0^n 1^n = \epsilon$; in S. Fix an n and assume that $0^n 1^n \in S$ (this is the I.H.)

For the I.S. note that $0^{n+1} 1^{n+1} = 0 0^n 1^n 1$. The I.H. guarantees a derivation exists in which $0^n 1^n$ occurs. Without loss of generality (see remark following 1.6.0.6) the derivation has the form $a_1, a_2, \ldots, 0^n 1^n$. This can be extended to the derivation $a_1, a_2, \ldots, 0^n 1^n, 0 0^n 1^n 1$, hence $0 0^n 1^n 1 \in S$. $\square$

1.6.0.11 Definition. A set S is *closed under an n-ary operation* iff, for every n-tuple of inputs chosen from S, all the results that the operation produces are also in S. $\square$

For example, $\mathbb{N}$ is closed under $x + y = z$ (z is output), $x \times y = z$ (z is output), but not under $x - y = z$ (z is output). For example, $0 - 1 = -1 \notin \mathbb{N}$.

1.6.0.12 Definition. (Closure) Given a set of initial objects, $\mathcal{I}$, and a set of operations, $\mathcal{O}$. A set S is called the *closure of $\mathcal{I}$ under $\mathcal{O}$*—in symbols $S = \text{Cl}(\mathcal{I}, \mathcal{O})$—iff it is the *smallest* set that contains $\mathcal{I}$ as a subset and is closed under *all* of the operations of $\mathcal{O}$.

A set such as $\text{Cl}(\mathcal{I}, \mathcal{O})$ is also called *recursively* or *inductively* defined from the initial objects $\mathcal{I}$ and rules $\mathcal{O}$. $\square$

Note that "smallest" means $\subseteq$-smallest, that is, if a set T contains $\mathcal{I}$ and is closed under $\mathcal{O}$, then $S \subseteq T$. This attribute, smallest, directly leads to the technique of (structural) induction over $\text{Cl}(\mathcal{I}, \mathcal{O})$:

Structural Induction: Let $S = \text{Cl}(\mathcal{I}, \mathcal{O})$ and $\mathscr{P}(x)$ be a property (formula). To show that all $a \in S$ have the property, do the following:

(1) Prove $\mathscr{P}(a)$ for all $a \in \mathcal{I}$.

(2) Prove that the *property propagates* with every $R \in \mathcal{O}$, that is, whenever the inputs of R have the property, then so does the output.

The part "the inputs of R have the property" above is the I.H. for R. *There will be one I.H. for each $R \in \mathcal{O}$.* The I.S. for the R in question is to prove that, based on the I.H., the output has the property—i.e., the property propagates from the input side to the output side of the "black box" R.

Why "structural"? Because the induction is with respect to how the set was built.

The process (1)–(2) is *(structural) induction over S*, or *induction with respect to S*.

1.6.0.13 Theorem. *Structural Induction works: That is, if (1) and (2) above are indeed proved, then, for all $a \in S$, $\mathscr{P}(a)$ holds.*

Proof. Let $\mathbb{P} = \{a : \mathscr{P}(a) \text{ holds}\}$. Now (1) translates into $\mathcal{I} \subseteq \mathbb{P}$, while, by (2), for any $R \in \mathcal{O}$, whenever all the inputs of R are in $\mathbb{P}$ [i.e., they all satisfy $\mathscr{P}(x)$], then so is the output, that is, $\mathbb{P}$ is closed under *all* the operations of $\mathcal{O}$. By the "smallest" property of S (1.6.0.12), we have $S \subseteq \mathbb{P}$, that is, for all $a \in S$, $\mathscr{P}(a)$ holds. $\square$

It turns out that not all properties $\mathscr{P}(x)$ lead to *sets* $\{x : \mathscr{P}(x)\}$—some such collections are "too big" to be, *technically*, "sets" (cf. Section 1.3).

Our proof above was within Cantor's informal or naïve set theory that glosses over such small print. However, formal set theory, that is meant to save us from our naïveté, upholds the "principle" of structural induction, (1)–(2), albeit using a slightly more complicated proof. Cf. Tourlakis (2003b).

1.6.0.14 Theorem. *Given a set of initial objects, $\mathcal{I}$, and a set of operations, $\mathcal{O}$. The two sets: S*—built by steps *(1.6.0.9) from $\mathcal{I}$ and $\mathcal{O}$—and* $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ *are equal.*

Proof. For $\subseteq$ we do induction on derivation length of $a \in S$. If the length equals 1, then $a \in \mathcal{I}$. Since $\mathcal{I} \subseteq \mathrm{Cl}(\mathcal{I}, \mathcal{O})$ by 1.6.0.12, the basis is settled. Assume next (I.H.) that for all $k < n$, if a occurs in a derivation of length k, then a is in the closure.

I.S.: Let a occur in a derivation of length n. If it does not occur at the right end, then the I.H. kicks in and a is in the closure. So let a be the last object in the derivation. If it is initial, we have nothing to add to what we said for the basis. Suppose instead that a is the result of an operation from $\mathcal{O}$ that was applied on inputs $a_{j_1}, \ldots, a_{j_r}$ that appeared to the left of a in the derivation. By the I.H. all the a_{j_m} are in the closure. The later being closed under all operations from $\mathcal{O}$ we conclude that the result of the operation, a, is in the closure.

For $\supseteq$ we do induction over $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$: For the basis, if $a \in \mathcal{I}$ then $a \in S$ via a derivation of length 1. We now show that the property "$a \in S$" propagates with every $R \in \mathcal{O}$. To unclutter exposition, and without loss of generality, fix an R—and pretend without loss of generality that its arity is 3—and let its inputs a, b, c be all in S. Let $R(a, b, c, d)$. We want $d \in S$.

Well, by I.H. there are three derivations $\ldots, a, \ldots; \ldots, b, \ldots;$ and $\ldots, c, \ldots$

If we concatenate them into one sequence

$$\ldots, a, \ldots, \ldots, b, \ldots, \ldots, c, \ldots$$

we have a derivation (why?). Due to the way d is obtained, so is

$$\ldots, a, \ldots, \ldots, b, \ldots, \ldots, c, \ldots, d$$

But then $d \in S$ by the way S is obtained. $\qquad\square$

1.6.0.15 Remark. The above is a significant theorem: If we want to prove properties of $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ as a whole, the best idea is to do structural induction over the set. If on the other hand we want to prove that some a is a member of $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$, then the best idea is to provide a derivation for it.

Compare: If we want to prove a property of all theorems of a theory, then we do induction over the theory that is built using 1.1.1.34 (and 1.1.1.38). If on the other hand we want to verify that a formula is a theorem, then we produce a proof for it. Evidently, by 1.6.0.14 we have that the iterative definition of "theorem" in 1.1.1.34 is equivalent with the inductive one: *The set of all theorems is* $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ *where $\mathcal{I}$ is*

the adopted set of axioms and $\mathcal{O}$ is the adopted rules of inference. Cf. also 1.6.0.17 below.

> Note that since b appears in a derivation iff it is either initial or a *result* of some $R \in \mathcal{O}$ applied on *prior members of the derivation*—and the latter is tantamount to saying "*members of* $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$" because of 1.6.0.14—we have the following very useful "membership test":
>
> $b \in \mathrm{Cl}(\mathcal{I}, \mathcal{O})$ *iff* $b \in \mathcal{I}$ *or b is the result of some rule applied to members of* $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$.

In words, the theorem says that the inductive approach—forming the closure—and the iterative approach, building one element at a time via a derivation, yield the same result.   □

1.6.0.16 Example. Let $\mathcal{I} = \{3\}$ and $\mathcal{O}$ consist of just $x + y = z$ and $x - y = z$, where in both cases z is the output variable. We are thinking of $\mathbb{Z}$ as our reference set here. Let us see why we have

$$\mathrm{Cl}(\mathcal{I}, \mathcal{O}) = \{3k : k \in \mathbb{Z}\} \tag{1}$$

For the $\subseteq$ we do, of course, induction over $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$. Well, $\mathcal{I}$ contains just 3, and $3 = 3 \times 1$, hence is in the right hand side (rhs) of (1).

Let us see that *membership to the right* propagates with the two rules: So let a and b be in the rhs. Then $a = 3m$ and $b = 3r$ for some m, r in $\mathbb{Z}$. Trivially, each of $a + b$ and $a - b$ is a multiple of 3.

As for $\supseteq$, let a be in the rhs, that is, $a = 3k$ for some $k \in \mathbb{Z}$.

Case 1. $k \geq 0$. Let us do induction on $k \geq 0$ to show that $3k$ in the left hand side (lhs). Well, if $k = 0$, then we are done by the derivation $3, 0$ (why is this a derivation?).

Take as I.H. the truth of the claim for (fixed) k and go to $k + 1$. Given that $3(k + 1) = 3k + 3$, we are done by the I.H. and since the lhs is closed under $x + y = z$ (of course, 3 is in lhs).

Case 2. $k < 0$. Well, $0 \in \mathrm{Cl}(\mathcal{I}, \mathcal{O})$: indeed, apply $x - y = z$ to input $3, 3$. But then $3k \in \mathrm{Cl}(\mathcal{I}, \mathcal{O})$ as well, since $3k = 0 - 3(-k)$; now apply the same rule on inputs 0 and $3(-k)$ with the help of Case 1.   □

1.6.0.17 Example. Let us work within arithmetic (simply for the sake of having a fixed alphabet of symbols). We take as $\mathcal{I}$ the set of all logical axioms (1.1.1.38), and these two rules form $\mathcal{O}$:

$$M(\mathscr{X}, \mathscr{Y}, \mathscr{Z}) \text{ holds iff } \mathscr{Y} \text{ has the form } \mathscr{X} \to \mathscr{Z} \tag{MP}$$

and

$$G(\mathscr{X}, \mathscr{Y}) \text{ holds iff } \mathscr{Y} \text{ has the form } (\forall x)\mathscr{X} \text{ for some } x \tag{Gen}$$

That is, our familiar MP and Gen. So, what is $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$? But of course—immediately from 1.6.0.14—it is the set of all absolute theorems (provable without nonlogical axioms) that we can prove if we employ as our *only* rules Gen and MP (cf. 1.1.1.34).

By induction over this $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$—or as logicians prefer to say, by *induction on theorems*—we can prove the soundness of this proof system: That every theorem, i.e., member of $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$, is true.

Well, the claim holds for $\mathcal{I}$ as we already know (1.1.1.39).

We only need to show that the claim propagates with the two rules above: Indeed, the MP is a special case of tautological implication, and Gen preserves truth by 1.1.1.15. □

In 1.1.1.34 we adopted all tautological implications—not just MP—as rules. This was expedient. It suffices to include just one such implication: MP. The interested reader can see why in Tourlakis (2008, 2003a).

Both examples 1.6.0.7 and 1.6.0.16 employ rules that are functions (single valued). Example 1.6.0.17 on the other hand has a rule that is not a function:

$$\text{input: } \mathscr{A}; \text{ output: } (\forall x)\mathscr{A}$$

since for each of the infinitely many choices of x we have a different output (why "infinitely many"?)

A more crucial—and troublesome—observation is this: In 1.6.0.7 every member of the closure has a unique *immediate predecessor*. Not so in Examples 1.6.0.16 and 1.6.0.17. In the former, 12 could be $15 - 3$ or $6 + 6$ or $9 + 3$. Indeed, 3 is both initial, and something that can be (re)built: $6 - 3$, for example. In the latter example, if $\mathscr{A}, \mathscr{A} \to \mathscr{B}$ yield $\mathscr{B}$ so do infinitely many pairs $\mathscr{X}, \mathscr{X} \to \mathscr{B}$ for all possible choices of $\mathscr{X}$. This phenomenon is called *ambiguity*.

1.6.0.18 Definition. (Ambiguity) A pair $\mathcal{I}, \mathcal{O}$ is *ambiguous* if one or more of the following hold. Otherwise it is *unambiguous*.

(1) For some $a \in \mathrm{Cl}(\mathcal{I}, \mathcal{O})$ and some n-ary rule $R \in \mathcal{O}$, there are $\langle p_1, \ldots, p_n \rangle \neq \langle q_1, \ldots, q_n \rangle$ such that $R(p_1, \ldots, p_n, a)$ and $R(q_1, \ldots, q_n, a)$;

(2) For some $a \in \mathrm{Cl}(\mathcal{I}, \mathcal{O})$ and two distinct n-ary and m-ary rules R and Q in $\mathcal{O}$, there are $\langle p_1, \ldots, p_n \rangle$ and $\langle q_1, \ldots, q_m \rangle$ such that $R(p_1, \ldots, p_n, a)$ and $Q(q_1, \ldots, q_m, a)$;

(3) For some element $a \in \mathcal{I}$, there is an n-ary rule $R \in \mathcal{O}$, and a tuple $\langle p_1, \ldots, p_n \rangle$ such that $R(p_1, \ldots, p_n, a)$.

If $a \in \mathrm{Cl}(\mathcal{I}, \mathcal{O})$ and $R(p_1, \ldots, p_n, a)$ holds for some $R \in \mathcal{O}$, we will call $\langle p_1, \ldots, p_n \rangle$ a vector (or sequence) of *immediate predecessors* of a. For short, *i.p.* □

1.6.0.19 Example. Here is why ambiguity is trouble. Let us start with the alphabet of symbols $A = \{1, 2, 3, +, \times\}$. We will inductively define a restricted set of arithmetic expressions (for example, we employ no variables) as follows. Let $\mathcal{I} = \{1, 2, 3\}$ and let $\mathcal{O}$ consist of just two *string* operations:

$$\text{from strings } X \text{ and } Y \text{ form } X + Y \tag{1}$$

<div align="center">from strings X and Y form $X \times Y$ (2)</div>

Some examples are $1 + 1$, 2×1 and, more interestingly, $1 + 2 \times 3$. What do these strings *mean*? Let us assign the "natural" meaning: "1" means 1, "2" means 2, and "3" means 3. "+" means add (plus) and "×" means multiply. Thus, extending this to an arbitrary member of $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ we will opt for the natural approach: As $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ is defined inductively itself, why not effect a *recursive definition of meaning* via a function "EV" (for "evaluate"), which will compute the value of a member A of $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ *by calling itself recursively on A's i.p.*

Therefore, we define (if you will, we *program*) EV by:

$$
\begin{aligned}
EV(1) &= 1 \\
EV(2) &= 2 \\
EV(3) &= 3 \\
EV(X + Y) &= EV(X) + EV(Y) \\
EV(X \times Y) &= EV(X) \times EV(Y)
\end{aligned}
$$

So what is the value (meaning) of $1 + 2 \times 3$? Well,

$$
\begin{aligned}
EV(1 + 2 \times 3) &= EV(1 + 2) \times EV(3) \\
&= \Big(EV(1) + EV(2)\Big) \times 3 \\
&= \Big(1 + 2\Big) \times 3 \\
&= 9
\end{aligned}
$$

No, no, you say. It is

$$
\begin{aligned}
EV(1 + 2 \times 3) &= EV(1) + EV(2 \times 3) \\
&= EV(1) + \Big(EV(2) \times EV(3)\Big) \\
&= 1 + \Big(2 \times 3\Big) \\
&= 7
\end{aligned}
$$

We are both "right", of course. The pair $\mathcal{I}, \mathcal{O}$ is ambiguous; in particular, the string $1 + 2 \times 3$ has two i.p.: $\langle 1 + 2, 3 \rangle$ on which the first computation is based, and $\langle 1, 2 \times 3 \rangle$ on which we based the second computation.   □

While we are on the subject of closures, let us look at the very important *transitive closure* of a relation.

1.6.0.20 Definition. (Transitive Closure) The *transitive closure* of a relation R is the *smallest* (in the sense of inclusion, $\subseteq$) transitive relation that includes R, that is, if Q is *a* transitive closure of R, then we must have

    (1) $R \subseteq Q$
    (2) Q is transitive, and
    (3) If T is transitive and $R \subseteq T$, then $Q \subseteq T$.   □

(1) While we have no *a priori* reason to expect that transitive closures exist just by virtue of us coining this term, we can say one thing:

A relation R *cannot* have *more than one* transitive closure. Indeed, if Q and Q' are both transitive closures of R, then having Q' pose as "T" we get $Q \subseteq Q'$. Next, having Q so pose, we have $Q' \subseteq Q$. Thus, $Q = Q'$.

(2) Intuitively, we can imagine the (we can now say "*the*") transitive closure of a relation R as the relation that we get from R by step-by-step adding pairs $\langle a, c \rangle$ to the relation that we have built so far, as long as, for some b, $\langle a, b \rangle$ and $\langle b, c \rangle$ are already included. We stop this process of adding pairs as soon as we obtain a transitive relation via this process. This observation is made precise below.

1.6.0.21 Theorem. *For any relation R, its unique transitive closure* exists *and equals* $\text{Cl}(\mathcal{I}, \mathcal{O})$, *where $\mathcal{I} = R$—a set of ordered pairs—and $\mathcal{O}$ contains just one* operation *on pairs that, for any two input pairs $\langle a, b \rangle$ and $\langle b, c \rangle$ (note the common b), the operation produces the pair $\langle a, c \rangle$.*

We will denote the transitive closure of R by R^+.

Proof. We show that $\text{Cl}(\mathcal{I}, \mathcal{O})$ satisfies (1)–(2) of 1.6.0.20, which will confirm that $R^+ = \text{Cl}(\mathcal{I}, \mathcal{O})$. For (1), we are done by the property $\mathcal{I} \subseteq \text{Cl}(\mathcal{I}, \mathcal{O})$ of any closure. For (2) we are done since $\text{Cl}(\mathcal{I}, \mathcal{O})$ is closed under the operation in $\mathcal{O}$: If $\langle a, b \rangle$ and $\langle b, c \rangle$ are in $\text{Cl}(\mathcal{I}, \mathcal{O})$, then so is $\langle a, c \rangle$.

For (3), let T be transitive and $R \subseteq T$. We want to show that $\text{Cl}(\mathcal{I}, \mathcal{O}) \subseteq T$. Well, both T and $\text{Cl}(\mathcal{I}, \mathcal{O})$ are supersets of R and are closed under the operation "if $\langle a, b \rangle$ and $\langle b, c \rangle$ are included, then so is $\langle a, c \rangle$". But, as a closure, $\text{Cl}(\mathcal{I}, \mathcal{O})$ is $\subseteq$-smallest with these two properties, therefore $\text{Cl}(\mathcal{I}, \mathcal{O}) \subseteq T$ as needed. $\square$

1.6.0.22 Corollary. *For any relation R, its transitive closure R^+ is equal to $\bigcup_{n \geq 1} R^n$.*

We also may write

$$R^+ = \bigcup_{n=1}^{\infty} R^n$$

Proof. Let us set $Q = \bigcup_{n=1}^{\infty} R^n$ and prove that $Q = \text{Cl}(\mathcal{I}, \mathcal{O})$, where $\mathcal{I}, \mathcal{O}$ are as in 1.6.0.21.

For $Q \subseteq \text{Cl}(\mathcal{I}, \mathcal{O})$ it suffices to prove that $R^n \subseteq \text{Cl}(\mathcal{I}, \mathcal{O})$, for $n \geq 1$, by induction on n: For $n = 1$, $a\, R^1\, b$ means $a\, R\, b$ thus $\langle a, b \rangle \in \text{Cl}(\mathcal{I}, \mathcal{O})$ since $R = \mathcal{I}$. Taking the obvious I.H. for n we next let $a\, R^{n+1}\, b$. This means that for some c we have $a\, R\, c$ and $c\, R^n\, b$. By the basis and I.H. respectively we have $\langle a, c \rangle$ and $\langle c, b \rangle$ in $\text{Cl}(\mathcal{I}, \mathcal{O})$, hence $\langle a, b \rangle$ is in $\text{Cl}(\mathcal{I}, \mathcal{O})$ (transitivity).

For $Q \supseteq \text{Cl}(\mathcal{I}, \mathcal{O})$ we do induction on the closure. Since $\mathcal{I} = R \subseteq Q$, we only need show that Q is transitive. Let then $a\, Q\, c$ and $c\, Q\, b$, hence, for some m and n, $a\, R^m\, c$ and $c\, R^n\, b$. Therefore $a\, R^m \circ R^n\, b$ and thus $a\, R^{m+n}\, b$ by Exercise 1.8.42. Thus $a\, Q\, b$. $\square$

1.6.0.23 Remark. (1) Thus, we have $a\, R^+\, b$ iff, for some n, $a\, R^n\, b$ iff, for some sequence

$$a_0, \ldots, a_n$$

where $a_0 = a$ and $a_n = b$, we have

$$a_i R a_{i+1}, \text{ for } i = 0, 1, \ldots, n-1$$

The notation below is also common.

$a R^+ b$ iff, for some a_j, it is $a R a_1 R a_2 R a_3 \cdots a_j R a_{j+1} \cdots a_{n-2} R a_{n-1} R b$

(2) If R is on A, then its *reflexive transitive closure* is denoted by R^* and is defined by $1_A \cup R^+$. That is, $a R^* b$ iff $a = b$ or $a R^+ b$. □

1.7 RECURSIVE DEFINITIONS OF FUNCTIONS

We often encounter a definition of a function over the natural numbers such as

$$f(0, m) = 0$$
$$f(n + 1, m) = f(n, m) + m$$

Is this a *legitimate* definition? That is, *is there* really a function *that satisfies* the above two equalities for all n and m? And if so, is there *only one* such function, or is the definition ambiguous? We address this question in this section through a somewhat more general related question.

1.7.0.24 Example. Let us look at a simpler question than the above and see if we can produce a good answer. First off, is there a function g given by the following two equalities for all values of n?

$$g(0) = 1$$
$$g(n + 1) = 2^{g(n)}$$

Well, let's see: By induction on n we can show that $g(n) \downarrow$ for all n: Indeed, this is true for $n = 0$ by the first equality. Taking the I.H. that $g(n) \downarrow$ (for a fixed unspecified n) we can compute $g(n + 1)$ so indeed

$$g(n) \downarrow \text{ for all } n$$

We can say then that the function g exists; right?

Wrong! A function is a set, in this case an infinite table of pairs (if we take its existence for granted). We did *not* show that it exists as *a table of pairs*; rather *we have only shown* that

If *a g satisfying the given equalities exists, then it is total.*[45]

---

[45]The set of texts on the subject of *the theory of computation*, which seriously propose the above erroneous "proof" of existence is non-empty.

We can however prove that any two functions satisfying the equalities must be equal. That is, if h is another such function, then $h = g$ or, on an input by input basis,

$$g(n) = h(n), \text{ for all } n \qquad (1)$$

Note that we wrote = in (1), rather than $\simeq$ (cf. 1.2.0.11), since we already know that if a function satisfies the given equalities, then it is total.

As for (1), for $n = 0$ we are done by the first equality. Taking as I.H. the case for some fixed n, the case for $n+1$ is easily settled: $g(n+1) = 2^{g(n)} = 2^{h(n)} = h(n+1)$, where the middle = is due to the I.H. □

So how do we settle existence? We *build* a function f (or g) given as above either iteratively, *by stages* [which is the usual approach in the literature, when done correctly; cf. Tourlakis (1984)] or as a *closure*; a set of the form $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ for appropriate $\mathcal{I}, \mathcal{O}$. The latter approach is from Tourlakis (2003b), which also develops the iterative approach. Mindful of the example in 1.6.0.19, we will define functions recursively on an inductively defined set $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ that is given via an unambiguous pair $\mathcal{I}, \mathcal{O}$.

1.7.0.25 Theorem. (Function definition by recursion) *Let $\mathcal{I}, \mathcal{O}$ be an* unambiguous *pair and* $\mathrm{Cl}(\mathcal{I}, \mathcal{O}) \subseteq A$, *for some set A. Let $h : \mathcal{I} \to B$ and $g_R : A \times B^r \to B$, for each r-ary $R \in \mathcal{O}$, be given functions.*[46]

Under these assumptions, **there is** *a* **unique** *function $f : \mathrm{Cl}(\mathcal{I}, \mathcal{O}) \to B$ such that*

$$y = f(x) \text{ iff } \begin{cases} y = h(x) & \text{and } x \in \mathcal{I} \\ \quad \text{OR,} \quad \text{for some } R \in \mathcal{O}, \\ y = g_R(x, o_1, \dots, o_r) \quad \text{and } R(a_1, \dots, a_r, x) \text{ holds,} \\ \qquad \text{where } o_i = f(a_i), \text{ for } i = 1, \dots, r \end{cases}$$

$$\qquad (1)$$

The reader may wish to postpone studying this proof, but he should become thoroughly familiar with the statement of the theorem, and study the examples that follow the proof.

Proof. To *prove the existence* of a "solution" f to the given recursive definition (1), we will *build* a single-valued binary relation $F \subseteq A \times B$, which, when we rewrite $F(x, y)$ as "$y = F(x)$"—with y as the output variable—satisfies (1) above. To build it, we will realize it as an appropriate $\mathrm{Cl}(\mathcal{J}, \mathcal{T})$, for appropriately chosen initial set $\mathcal{J}$ and set of rules $\mathcal{T}$.

For each r-ary rule $R \in \mathcal{O}$, define the r-ary rule $\widehat{R}$ by

$$\widehat{R}\big(\langle a_1, o_1 \rangle, \dots, \langle a_r, o_r \rangle, \langle b, g_R(b, o_1, \dots, o_r) \rangle\big) \text{ iff } R(a_1, \dots, a_r, b) \qquad (2)$$

---

[46]An r-ary operation (rule) is an $(r + 1)$-ary relation; cf. 1.6.0.5.

For any $a_1, \ldots, a_r, b$, the above definition of $\widehat{R}$ is effected for all possible choices of $o_1, \ldots, o_r$ in B for which $g_R(b, o_1, \ldots, o_r)$ is defined.

By the way, there is no mystery in the definition of $\widehat{R}$ (the name chosen to show the close association with R): If we anticipate that (1) *does* have an f-solution, we can then view the o_i as the $f(a_i)$. Then $\widehat{R}$'s job is—*once we give it, in the form of input/output pairs, where the outputs are those of f on all the i.p. of b*—to compute $f(b)$ using g_R and to output the input/output pair $\langle b, f(b) \rangle$).

Collect now all the rules $\widehat{R}$ as defined, to form the rule-set $\mathcal{T}$.

As the initial objects-set $\mathcal{J}$, which we will associate with the rule-set $\mathcal{T}$, we take $\mathcal{J} = h$—that is, $\langle x, y \rangle \in \mathcal{J}$ iff $h(x) = y$.

Claim 1. The set $F = \mathrm{Cl}(\mathcal{J}, \mathcal{T})$ is a single-valued binary relation $\mathrm{Cl}(\mathcal{I}, \mathcal{O}) \to B$.

Proof of Claim 1. First off, that $F \subseteq \mathrm{Cl}(\mathcal{I}, \mathcal{O}) \times B$ is immediate: $\mathcal{J} \subseteq \mathrm{Cl}(\mathcal{I}, \mathcal{O}) \times B$, and each relation of $\mathcal{T}$ has as *output* a pair in $\mathrm{Cl}(\mathcal{I}, \mathcal{O}) \times B$, by definition (2).

We next establish that F is single valued in its second component, doing induction over $\mathrm{Cl}(\mathcal{J}, \mathcal{T})$. The claim to prove will be

$$\text{if } \langle a, b \rangle \in F \text{ and } \langle a, c \rangle \in F, \text{ then } b = c \tag{$*$}$$

Basis: Suppose that $\langle a, b \rangle \in \mathcal{J}$ and let also $\langle a, c \rangle \in F$.

By 1.6.0.15, the latter entails, in *principle*:

(i) $\langle a, c \rangle \in \mathcal{J}$. Then $c = h(a) = b$,

OR,

(ii) for some r-ary $\widehat{R} \in \mathcal{T}$, we have $\widehat{R}(\langle a_1, o_1 \rangle, \ldots, \langle a_r, o_r \rangle, \langle a, c \rangle)$, where $R(a_1, \ldots, a_r, a)$, $c = g_R(a, o_1, \ldots, o_r)$, and $\langle a_1, o_1 \rangle, \ldots, \langle a_r, o_r \rangle$ are in F.

The right hand side of the capitalized "or" cannot be applicable—due to its requirement that $R(a_1, \ldots, a_r, a)$—given that $a \in \mathcal{I}$ and $(\mathcal{I}, \mathcal{O})$ is unambiguous.

We next prove that the property $(*)$ propagates with each $\widehat{Q} \in \mathcal{T}$. So, let

$$\widehat{Q}(\langle a_1, o_1 \rangle, \ldots, \langle a_r, o_r \rangle, \langle a, b \rangle)$$

Since by the previous argument $\langle a, c \rangle \notin \mathcal{J}$, let also

$$\widehat{P}(\langle a_1', o_1' \rangle, \ldots, \langle a_l', o_l' \rangle, \langle a, c \rangle)$$

where $Q(a_1, \ldots, a_r, a)$ and $P(a_1', \ldots, a_l', a)$, but also [cf. (2)]

$$b = g_Q(a, o_1, \ldots, o_r) \text{ and } c = g_P(a, o_1', \ldots, o_l') \tag{3}$$

Since $(\mathcal{I}, \mathcal{O})$ is unambiguous, $Q = P$ (hence also $\widehat{Q} = \widehat{P}$); thus $r = l$, and $a_i = a_i'$, for $i = 1, \ldots, r$.

By I.H., $o_i = o_i'$, for $i = 1, \ldots, r$, hence $b = c$ by (3). *End of proof: Claim 1.*

Claim 2. F satisfies (1). Now that we know that F is a function we can write

$$b = F(a) \text{ for } \langle a, b \rangle \in F$$

Our task here is to show that if we replace the "(function) variable" f in (1) by the *constructed* F, the "iff" stated in (1) will hold.

($\leftarrow$) direction: We prove that the right hand side (rhs) of (1) implies the left, if the letter f is replaced throughout by F. The rhs is a disjunction, so we have two cases to consider [cf. 1.1.1.48(a)]. First, let $x \in \mathcal{I}$ and $y = h(x)$. Since $h = \mathcal{J} \subseteq F$, we have that $F(x, y)$ is true, that is, $y = F(x)$.

Second, consider the complicated side of OR in (1): So let, for some $R \in \mathcal{O}$, $y = g_R(x, o_1, \ldots, o_r)$, where $R(a_1, \ldots, a_r, x)$ and $o_i = F(a_i)$, for $i = 1, \ldots, r$.

By (2), $\widehat{R}(\langle a_1, o_1 \rangle, \ldots, \langle a_r, o_r \rangle, \langle x, y \rangle)$, thus—$F$ being closed under all the rules in $\widehat{R}$—$\langle x, y \rangle \in F$; for short, $y = F(x)$.

($\rightarrow$) direction   Now we assume that $F(x, y)$ holds. We want to infer the right hand side (of *iff*) in (1)—with f replaced by F.

So let $y = F(x)$. There are two cases according to 1.6.0.15:

Case 1. $\langle x, y \rangle \in \mathcal{J}$. Thus (by $\mathcal{J} = h$) $y = h(x)$, and $x \in \mathcal{I}$ (definition of $\mathcal{J}$); the top case of (1).

Case 2. Suppose next that $\langle x, y \rangle \in F$ because, for some $\widehat{Q} \in \mathcal{T}$, the following hold (see (2)):

(a) $\widehat{Q}(\langle a_1, o_1 \rangle, \ldots, \langle a_r, o_r \rangle, \langle x, y \rangle)$

(b) $Q(a_1, \ldots, a_r, x)$

(c) $y = g_Q(x, o_1, \ldots, o_r)$

(d) All of $\langle a_1, o_1 \rangle, \ldots, \langle a_r, o_r \rangle$ are in F

By (d), $o_i = F(a_i)$, for $i = 1, \ldots, r$. But then the conjunction of the foregoing observation with (b) and (c) is the right hand side of the OR; as needed. *End of proof: Claim 2.*

Uniqueness of F. Let the function K also satisfy (1). We show by induction over $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ that

$$\text{For all } x \in \mathrm{Cl}(\mathcal{I}, \mathcal{O}) \text{ and all } y \in B, \quad y = F(x) \textbf{ iff } y = K(x) \qquad (4)$$

($\rightarrow$)   Let $x \in \mathcal{I}$, and $y = F(x)$. By lack of ambiguity, x has no i.p. Thus, $y = h(x)$. But then, $y = K(x)$, since K satisfies (1).

Let now $Q(a_1, \ldots, a_r, x)$ and $y = F(x)$. By (1), there are (unique, as we now know) $o_1, \ldots, o_r$ such that $o_i = F(a_i)$, for $i = 1, \ldots, r$, and $y = g_Q(x, o_1, \ldots, o_r)$. By the I.H., $o_i = K(a_i)$. As K satisfies (1), $y = K(x)$.

($\leftarrow$)   The roles of the letters F and K in the above argument being symmetric, we need say no more. $\square$

The above formulation with the so-called "graphs" of the utilized functions—a term applying to the relation $y = f(x)$[47]—rather than writing, say,

$$f(x) = \begin{cases} h(x) & \textbf{if } x \in \mathcal{I} \\ g_Q(x, f(a_1), \ldots, f(a_r)) & \textbf{if } Q(a_1, \ldots, a_r, x) \text{ holds,} \end{cases}$$

appears to be unnecessarily cumbersome. Cumbersome, yes, but not "unnecessarily":

In the used formulation, by keeping an eye on both the input and output sides at once, we took care of the partial function case (*total* or *not*) without having to worry about points of undefinition of the defined function or to use Kleene equality. In fact, using "=" above is incorrect in the nontotal case.

1.7.0.26 Example. Referring back to Example 1.7.0.24, we see that the defined by recursion g exists (and is, of course, unique): It is defined over the set $\mathbb{N} = \mathrm{Cl}(\{0\}, \{S\})$, where I denoted by S the successor rule:

$$S(x, x + 1), \text{ for all } x \text{ in } \mathbb{N}$$

The rule is clearly unambiguous, so Theorem 1.7.0.25 applies. □

1.7.0.27 Example. Fix $n > 0$ from $\mathbb{N}$ and consider the rule R below

$$R(\langle x, \vec{y}_n \rangle, \langle x + 1, \vec{y}_n \rangle), \text{ for all } x, y_i \text{ in } \mathbb{N}$$

and form $\mathbb{N}_n = \mathrm{Cl}(\mathcal{I}, \{R\})$, where $\mathcal{I} = \{\langle 0, \vec{y}_n \rangle : \text{ for all } y_i \text{ in } \mathbb{N}\}$.

The rule R is clearly unambiguous: every $\langle x, y_n \rangle$ is either initial or has the unique i.p. $\langle x - 1, y_n \rangle$. Thus Theorem 1.7.0.25 applies—$\mathbb{N}_n = \mathbb{N}^{n+1}$, of course—and enables recursive definitions like the following, based on two given functions h and g from $\mathbb{N}^n$ and $\mathbb{N}^{n+1} \times B$ respectively to some set B, to produce unique f-solutions.

$$f(x, \vec{y}_n) \simeq \begin{cases} h(\vec{y}_n) & \textbf{if } x = 0 \\ g(x, \vec{y}_n, f(x - 1, \vec{y}_n)) & \textbf{otherwise} \end{cases} \tag{1}$$

As is usual, we listed the arguments, e.g., in f, in their proper order, however omitting the $\langle \ldots \rangle$ brackets.

The above recurrence is the *primitive recursion schema* of Kleene, and it will play quite an active role in the next chapter. It is customary to write the schema in this form:

$$f(0, \vec{y}_n) \simeq h(\vec{y}_n)$$
$$f(x + 1, \vec{y}_n) \simeq g'(x, \vec{y}_n, f(x, \vec{y}_n))$$

---

[47]To plot $f(x)$ you plot the pairs (x, y) such that $y = f(x)$; hence the terminology "graph".

where g' is g, but modified to accept input x rather than $x + 1$ in the first argument slot.

Incidentally, the recursion given at the very opening of this section—taking there $B = \mathbb{N}$—fits the primitive recursion schema above, so the function it defines exists and is unique. The reader will immediately recognise that the function defined there is multiplication, $n \times m$. $\qquad\square$

1.7.0.28 Exercise. Prove that if h and g are total, then so is f defined by (1) above, so we can replace $\simeq$ by $=$. $\qquad\square$

1.7.0.29 Example. What about a recursion like this, where we still take our inputs (of f) from $\mathbb{N}$?

$$f(x, \vec{y}_n) \simeq \begin{cases} h(\vec{y}_n) & \textbf{if } x = 0 \\ g\Big(x, \vec{y}_n, \big\{f(0, \vec{y}_n), \dots, f(x-2, \vec{y}_n), f(x-1, \vec{y}_n)\big\}\Big) & \textbf{otherwise} \end{cases}$$

Before we answer this, a few comments on intentions, and notation. We intend that the value $f(x, \vec{y}_n)$ is computed based on our knowledge of the *entire history of values* of f—or *course-of-values* [Kleene (1952)]—at the set of all previous "points"

$$\Big\{ \langle 0, \vec{y}_n \rangle, \langle 1, \vec{y}_n \rangle, \langle x-1, \vec{y}_n \rangle \Big\} \tag{1}$$

As we can have no functions of a *variable number of arguments*, we have tentatively grouped the entire history into a single *set*-argument. It turns out that it is more profitable to use a set of *pairs* of inputs *and* outputs (of f) rather than just outputs in the recursive call embedded in g above, since such pairs can naturally handle nontotal functions—the pair $\langle a, f(a) \rangle$ is listed iff $f(a) \downarrow$.

$$\Big\{ \langle \langle 0, \vec{y}_n \rangle, f(0, \vec{y}_n) \rangle, \dots, \langle \langle x-2, \vec{y}_n \rangle, f(x-2, \vec{y}_n) \rangle, \langle \langle x-1, \vec{y}_n \rangle, f(x-1, \vec{y}_n) \rangle \Big\}$$

$$f(0, \vec{y}_n) \simeq h(\vec{y}_n)$$
$$f(x+1, \vec{y}_n) \simeq g'\Big(x, \vec{y}_n, \big\{ \langle \langle 0, \vec{y}_n \rangle, f(0, \vec{y}_n) \rangle, \dots, \langle \langle x-1, \vec{y}_n \rangle, f(x-1, \vec{y}_n) \rangle \big\}\Big)$$

The switch to g' from g reflects the modification to the x-argument and to the set-argument.

Now, if we call the set (1) "$S_{x-1, \vec{y}_n}$" for the sake of convenience, we may rewrite the above recurrence more correctly, and without "...", as

$$f(0, \vec{y}_n) \simeq h(\vec{y}_n) \tag{2}$$

$$f(x+1, \vec{y}_n) \simeq g'\Big(x, \vec{y}_n, f \restriction S_{x, \vec{y}_n}\Big) \tag{3}$$

Here is now why the recurrence (2)–(3) has a unique f-solution: Let us write $H(x, \vec{y}_n)$ as an abbreviation of $f \restriction S_{x,\vec{y}_n}$. We can then have—using (2)–(3)—a (simple) primitive recursion (as in 1.7.0.27) for H:

$$H(0, \vec{y}_n) \simeq \left\{ \left\langle \langle 0, \vec{y}_n \rangle, h(\vec{y}_n) \right\rangle \right\} \tag{4}$$

$$H(x + 1, \vec{y}_n) \simeq H(x, \vec{y}_n) \cup \left\{ \left\langle \langle x + 1, \vec{y}_n \rangle, g'\left(x, \vec{y}_n, H(x, \vec{y}_n)\right) \right\rangle \right\} \tag{5}$$

A unique H exists that satisfies (4)–(5), by 1.7.0.27. But $f(x, \vec{y}_n) \simeq H(x, \vec{y}_n)\left(x, \vec{y}_n\right)$ for all $x, \vec{y}_n$; so f exists and is unique.[48]

> Given that $H(x, \vec{y}_n) = f \restriction S_{x,\vec{y}_n}$—a single-valued table of tuples—we see that evaluating $H(x, \vec{y}_n)$ at $\langle i, \vec{y}_n \rangle$, for $i = 0, 1, \dots, x$, we end up with the output (if it exists) of f at input $\langle i, \vec{y}_n \rangle$. That is, the expression "$H(x, \vec{y}_n)(x, \vec{y}_n)$" above makes sense, and, when defined, teases out $f(x, \vec{y}_n)$.

If we work strictly within arithmetic—that is, we allow no set arguments, in particular—then one neat way to deal with a sequence of numbers is to *code them by a single number*. Applying this trick reduces once again the original recursion to the standard schema of Example 1.7.0.27. This approach has additional important side benefits and we will revisit it in the next chapter. □

1.7.0.30 Example. Let $A = \{1, 2\}$. In this example we consider only functions with inputs from A^* and outputs in A^*. Suppose that h, g_1 and g_2 are such given functions of n for the first and $n + 2$ variables for the other two. The recursion (for fixed $n > 0$)

$$
\begin{aligned}
f(\epsilon, \vec{y}_n) &\simeq h(\vec{y}_n) \\
f(x * 1, \vec{y}_n) &\simeq g_1(x, \vec{y}_n, f(x, \vec{y}_n)) \\
f(x * 2, \vec{y}_n) &\simeq g_2(x, \vec{y}_n, f(x, \vec{y}_n))
\end{aligned}
$$

is called *right primitive recursion on notation*—"right" since we change x by concatenating a 1 or 2 to its right; "on notation" since we are thinking in terms of the notation rather than value of x when we increment it.

Given the h and the g_i there is a unique f that satisfies the three equalities above, for all $x, \vec{y}_n$. To apply Theorem 1.7.0.25 we define A_n^* as $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ where $\mathcal{O}$ has two rules,

$$R_1(\langle x, \vec{y}_n \rangle, \langle x * 1, \vec{y}_n \rangle), \text{ for all } x, \vec{y}_n \text{ in } A^*$$

and

$$R_2(\langle x, \vec{y}_n \rangle, \langle x * 2, \vec{y}_n \rangle), \text{ for all } x, \vec{y}_n \text{ in } A^*$$

We define $\mathcal{I}$ as $\{\langle \epsilon, \vec{y}_n \rangle : \text{ for all } y_i \text{ in } A^*\}$.

Clearly the pair $\mathcal{I}, \mathcal{O}$ is unambiguous and 1.7.0.25 applies to the recursion on notation schema above, proving existence and uniqueness of f.

---

[48]The "g-function" in (5) is $G(x, \vec{y}_n, Z) \simeq Z \cup \left\{ \left\langle \langle x + 1, \vec{y}_n \rangle, g'(x, \vec{y}_n, Z) \right\rangle \right\}$, where Z is a set argument. If the right field of the original h and g is a set B, then Z takes its values from $\mathcal{P}(\mathbb{N}^{n+1} \times B)$.

One can similarly utilize *left* recursion on notation, going from x to $i * x$, for $i = 1$ or $i = 2$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

1.8 ADDITIONAL EXERCISES

1. *Let us first define:* The set of propositional formulae of, say, set theory, denoted here by **Prop**, is the smallest set such that

   (1) Every Boolean variable is in **Prop** (cf. 1.1.1.26)

   (2) If $\mathscr{A}$ and $\mathscr{B}$ are in **Prop**, then so are $(\neg\mathscr{A})$ and $(\mathscr{A} \circ \mathscr{B})$ —where I used $\circ$ as an abbreviation of any member of $\{\wedge, \vee, \rightarrow, \equiv\}$.

   If we call **WFF** the set of all formulae of set theory as defined in 1.1.1.3, then show that **WFF** = **Prop**.

   Hint. This involves two structural inductions, one each over **WFF** and **Prop**.

2. Prove the general case of proof by cases (cf. 1.1.1.48): $\mathscr{A} \rightarrow \mathscr{B}, \mathscr{C} \rightarrow \mathscr{D} \vdash \mathscr{A} \vee \mathscr{C} \rightarrow \mathscr{B} \vee \mathscr{D}$.

3. Let us prove $\vdash x = y$. By way of contradiction, let us assume $\neg x = y$ (i.e., $x \neq y$). Using substitution (1.1.1.42) we obtain $\neg x = x$ which along with axiom (v) (1.1.1.38) and tautological implication yields the contradiction $x = x \wedge \neg x = x$. Done.

   Hm. There is something very wrong here! Clearly, $x = y$ is not true, hence a proof of it must be impossible (cf. soundness, p. 20). What *exactly* went wrong with our "proof"?

4. Verify that for any (formal) function $f[x]$ we have $\vdash x = y \rightarrow f[x] = f[y]$.

   Hint. Start with (vi) of 1.1.1.38 taking as "$\mathscr{A}[z]$" the formula $f[x] = f[z]$.

5. Give the missing details of Example 1.1.2.14.

6. This is a useful but simple exercise! For all sets A, B, C prove:

   $(i)\quad A \cup A = A$ $\qquad\qquad\qquad\qquad$ and $A \cap A = A$
   $(ii)\quad A \cup (A \cap B) = A$ $\qquad\qquad\quad$ and $A \cap (A \cup B) = A$
   $(iii)\, A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ and $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
   $(iv)\, A \subseteq B \equiv A \cup B = B$ $\qquad\qquad$ and $A \subseteq B \equiv A \cap B = A$

7. Compute $\bigcup\{2\}$.

8. Compute $\bigcap \emptyset$.

9. Compute $\bigcap\{7\}$.

10. Use induction on $n \geq 2$ to prove that if $\langle x_1, x_2, \ldots, x_n \rangle = \langle y_1, y_2, \ldots, y_n \rangle$, then, for $i = 1, \ldots, n$, we have $x_i = y_i$.

11. Can A^n be also defined as

$$A^1 \quad = A$$

and, for $n \geq 1$

$$A^{n+1} = A \times A^n$$

Why?

12. What is $A \times 1$? Why?

13. Prove that $A \times \emptyset = \emptyset \times A = \emptyset$.

14. Prove that $A \times B = \emptyset$ iff $A = \emptyset$ or $B = \emptyset$.

15. What is $\mathcal{P}(A)$ if A is an urelement?

16. Assume an intuitive understanding of "the set A has n elements". Prove by induction on n that 2^A has 2^n elements. This motivates the notation "2^A".

Hint. Show carefully that adding one new element to A doubles the number of its subsets.

17. Show that for any function $f : A \to B$, $f(a) \uparrow \equiv f_\to(\{a\}) = \emptyset$; and f is onto iff $(\forall x \in B) f_\leftarrow(\{x\}) \neq \emptyset$.

18. Show by an example that function composition is not commutative. That is, in general, $(gf) \neq (fg)$.

19. For any function g and sets X and Y, we have $g_\leftarrow(X - Y) = g_\leftarrow(X) - g_\leftarrow(Y)$.

20. Let $f : X \to Y$ be given as well as $A \subseteq Y$ and $B \subseteq Y$. Prove that

- $f_\leftarrow(A \cup B) = f_\leftarrow(A) \cup f_\leftarrow(B)$
- $f_\leftarrow(A \cap B) = f_\leftarrow(A) \cap f_\leftarrow(B)$

21. Let $f : X \to Y$ be given as well as $A \subseteq X$ and $B \subseteq X$. Prove that

- $f_\to(A \cup B) = f_\to(A) \cup f_\to(B)$
- $f_\to(A \cap B) \subseteq f_\to(A) \cap f_\to(B)$
- $B \subseteq A$ implies that $f_\to(A - B) \supseteq f_\to(A) - f_\to(B)$

22. Let $f : X \to Y$ be given as well as $A \subseteq X$ and $B \subseteq Y$. Prove that

- $f_\to\left(f_\leftarrow(B)\right) \subseteq B$
- $f_\leftarrow\left(f_\to(A)\right) \supseteq A$

23. Let the relation $R : \underset{i=1}{\overset{n}{\times}} A_i \to A_k$ be given by

$$R = \{\langle\langle a_1, \ldots, a_k, \ldots, a_n\rangle, a_k\rangle : a_j \in A_j, \text{ for } j = 1, \ldots, n\}$$

(1) Prove that R is a total function. We call it the k-th *Cartesian projection* function of $\underset{i=1}{\overset{n}{\times}} A_i$, and often denote it by p_k^n.

(2) Prove that if $f : B \to \underset{i=1}{\overset{n}{\times}} A_i$ is a (*vector-* or *tuple-valued*) function, then we may *decompose* it into n *functions*, f_i, for $i = 1, \ldots, n$: $f_i : B \to A_i$, so that, for all $a \in \operatorname{dom}(f)$, we have $f(a) = \langle f_1(a), \ldots, f_n(a)\rangle$.

We say that the f_i is the i-th *component* or *projection* of the tuple-valued (vector-valued) function f.

Hint. Consider $(p_i^n f)$ (i.e., $f \circ p_i^n$).

24. Prove Theorem 1.2.0.19.

25. Suppose that $f : A \to B$. Then $(1_B f) = f$ and $(f 1_A) = f$.

26. Let $f : A \to B$ be a 1-1 correspondence. Then show that $g = f^{-1} : B \to A$ is a 1-1 correspondence as well and $(fg) = 1_B$ while $(gf) = 1_A$.

27. Consider $f : A \to B$, $g : B \to A$ and $h : B \to A$ such that $(fg) = 1_B$ and $(hf) = 1_A$ hold.

Show that f is a 1-1 correspondence and that $g = h = f^{-1}$.

Hint. Start with expanding $(h(fg)) = (h 1_B)$, using associativity and Exercise 25.

28. Let $f : A \to B$ be a 1-1 correspondence and $g : B \to A$ be a function for which $(gf) = 1_A$. Then show that $g = f^{-1}$ and therefore $(fg) = 1_B$ as well.

29. Let $f : A \to B$ be a 1-1 correspondence and $g : B \to A$ be a function for which $(fg) = 1_B$. Then show that $g = f^{-1}$ and therefore $(gf) = 1_A$ as well.

Exercises 27, 28 and 29 show that if an f has both a left and a right inverse, then the two are equal to f^{-1} and in fact f is a 1-1 correspondence. Moreover, a 1-1 correspondence has a unique left and right inverse, equal in each case to f^{-1}.

This unique inverse is called—for 1-1 correspondences—"the" (two-sided) inverse.

30. Prove that R is transitive iff $R^2 \subseteq R$.

31. Prove that if $A \sim B$ and $B \sim C$ then $A \sim C$.

32. Prove, for any two functions f and g, that $f = g$ (as sets of tuples) iff $(\forall x)(f(x) \simeq g(x))$.

33. Prove the claims made in Remark 1.2.0.31.

34. Let $f : A \to B$ and $g : B \to C$ be onto functions. Show that $(gf) : A \to C$ is onto.

Hint. An $h : X \to Y$ is onto iff for any $b \in Y$, the "equation" $h(x) = b$ has a solution.

35. Revisit Theorem 1.3.0.42 and give it a mathematical proof, using tools from Section 1.4. Specifically, if $A \subseteq \mathbb{N}$ is infinite, define by recursion the function f by $f(0) = \min A$ and $f(n+1) = \min(A - \{f(0), \ldots, f(n)\})$ and prove that $\mathrm{dom}(f) = \mathbb{N}$, $\mathrm{ran}(f) = A$ and f is 1-1.

36. Prove that the range of $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ given by $f(x, y) = 2^x 3^y$ is infinite.

37. Prove that there is a 1-1 correspondence that corresponds each $S \subseteq \mathbb{N}$ to χ_S.

38. Let A and B be enumerable. Then $2^A \sim 2^B$.

Hint. Let $f : \mathbb{N} \to A$ and $g : \mathbb{N} \to B$ be 1-1 correspondences. Define $F : 2^A \to 2^B$ so that $F(\emptyset) = \emptyset$ and F sends the set $\{f(i_0), f(i_1), f(i_2), f(i_3), \ldots\}$ to the set $\{g(i_0), g(i_1), g(i_2), g(i_3), \ldots\}$. Argue that F is total, 1-1, and onto.

39. Refer to 1.6.0.19. Define the simple arithmetic terms of that example differently: Let us start with the alphabet of symbols $A = \{1, 2, 3, +, \times, (,)\}$. We let $\mathcal{I} = \{1, 2, 3\}$ and $\mathcal{O}$ consist of just two *string* operations:

$$\text{from strings } X \text{ and } Y \text{ form } (X + Y) \tag{1}$$

$$\text{from strings } X \text{ and } Y \text{ form } (X \times Y) \tag{2}$$

Prove that $\mathcal{I}, \mathcal{O}$ is unambiguous and thus EV, defined as in 1.6.0.19, over $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ exists and is unique.

Toward a proof of lack of ambiguity you may want to prove a couple of lemmata:

(a) Every member of $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$ has an equal number of left and right brackets.

(b) Every *proper* non-empty string *prefix* of an $A \in \mathrm{Cl}(\mathcal{I}, \mathcal{O})$ has an excess of left brackets.

(c) Every $A \in \mathrm{Cl}(\mathcal{I}, \mathcal{O})$ has unique i.p. and $1, 2, 3$ have no i.p.

40. Prove, as Euclid did, that every natural number $n > 1$ is a product of primes in a unique way, except for permutation of factors.

Hint. Use strong induction and 1.4.1.3.

41. Prove that if $R : A \to A$ is reflexive and also satisfies, for all x, y and z,

$$xRy \wedge xRz \to yRz$$

then it is also symmetric and transitive, hence an equivalence relation.

42. Prove, by induction on n, that for any relation R on a set A we have

(1) $R^m \circ R^n = R^{m+n}$

(2) $(R^m)^n = R^{mn}$

43. Suppose that R is on a finite set of n elements. Prove that $R^+ = \bigcup_{i=1}^{n} R^i$.

Hint. Cf. 1.6.0.22. Prove the redundancy of all terms beyond R^n in this case.

44. Suppose that R, defined on a finite set of n elements, is reflexive. Prove that $R^+ = R^{n-1}$.

Hint. Prove the redundancy of all terms but R^{n-1} in this case.

45. Let $m > 1$ be an integer. Prove that any integer $n > 0$ can be *uniquely* written as $n = mq + r$, where $0 < r \le m$.

Hint. Note the inequalities! Either imitate the *proof* given in 1.4.3.1, or base a proof on the *result* of 1.4.3.1.

46. (*m*-**ary notation.**) Prove that every integer $n \ge 0$ has a unique representation as

$$n = d_r m^r + d_{r-1} m^{r-1} + d_{r-2} m^{r-2} + \cdots + d_0 \tag{1}$$

where $0 \le d_i < m$ for all $i = 0, \ldots, r$. (1) is called *the m-ary notation of n*, and the d_i are the m-ary digits.

47. (*m*-**adic notation.**) Prove that every integer $n > 0$ has a unique representation as

$$n = d_r m^r + d_{r-1} m^{r-1} + d_{r-2} m^{r-2} + \cdots + d_0 \tag{2}$$

where $0 < d_i \le m$ for all $i = 0, \ldots, r$. (2) is called *the m-adic notation of n* [cf. Smullyan (1961); Bennett (1962)], and the d_i are the m-adic digits.

48. Prove that for any set X, we have $X \not\sim 2^X$.

CHAPTER 2

ALGORITHMS, COMPUTABLE
FUNCTIONS AND COMPUTATIONS

2.1 A THEORY OF COMPUTABILITY

Computability is the part of logic and theoretical computer science that gives a math-
ematically precise formulation to the concepts *algorithm*, *mechanical procedure*, and
calculable function (or relation). Its advent was strongly motivated, in the 1930s,
by Hilbert's *program* to found mathematics on a (metamathematically provably)
consistent (cf. 1.1.1.51) axiomatic basis, in particular by his belief that the *Entschei-
dungsproblem*, or *decision problem*, for axiomatic theories, that is, the problem "is
this formula a theorem of that theory?" was solvable by a mechanical procedure that
was yet to be discovered.

Now, since antiquity, mathematicians have invented "mechanical procedures",
e.g., Euclid's algorithm for the "greatest common divisor",[49] and had no problem
recognizing such procedures when they encountered them. But how do you math-
ematically *prove* the *nonexistence* of such a mechanical procedure for a particular

---

[49]That is, the largest positive integer that is a common divisor of two given integers.

problem? You need a *mathematical formulation* of what *is* a "mechanical procedure" in order to do that!

Intensive activity by many [Post (1936, 1944), Kleene (1936), Church (1936b), Turing (1936, 1937), and, later, Markov (1960)] led in the 1930s to several alternative formulations, each purporting to mathematically characterize the concepts *algorithm*, *mechanical procedure*, and *calculable function*. All these formulations were soon proved to be equivalent; that is, the calculable functions admitted by any one of them were the same as those that were admitted by any other. This led Alonzo Church to formulate his *conjecture*, famously known as "Church's Thesis", that any *intuitively* calculable function is also calculable within any of these mathematical frameworks of calculability or computability.[50]

By the way, Church proved [Church (1936a,b)] that Hilbert's fundamental *Entscheidungsproblem* admits no solution by functions that are calculable within any of the known mathematical frameworks of computability. Thus, if we accept his "thesis", the Entscheidungsproblem admits no algorithmic solution, period!

The eventual introduction of computers further fueled the study of and research on the various mathematical frameworks of computation, "models of computation" as we often say, and "computability" is nowadays a vibrant and very extensive field. The model of computation that I will present here, due to Shepherdson and Sturgis (1963), is a later model that has been informed by developments in computer science, in particular, by the advent of so-called *high level*[51] programming languages.

2.1.1   A Programming Framework for Computable Functions

So, what *is* a computable function, mathematically speaking? There are two main ways to approach this question. One is to define a programming formalism—that is, a programming language—and say: "a function is computable precisely if it can be 'programmed' in the programming language". Such programming languages are the *Turing Machines* (or TMs) of Turing and the *unbounded register machines* (or URMs) of Shepherdson and Sturgis. Note that the term *machine* in each case is a misnomer, as both the TM and the URM formulations are really programming languages, the first being very much like assembly language of "real" computers, the latter reminding us more of (subsets of) Algol (or Pascal).

The other main way is to define a set of computable functions inductively, as a $\text{Cl}(\mathcal{I}, \mathcal{O})$—cf. Section 1.6. To do so we start with some set of initial functions $\mathcal{I}$ that are immediately recognizable as "intuitively computable", and choose a set $\mathcal{O}$ of function-building operations that preserve the "computable" property. This approach was originally due to Dedekind (1888) for what we nowadays call *primitive recursive*

---

[50] I will stress that even if this sounds like the "completeness *theorem*" of logic—that states the provability of all universally true formulae—Church's Thesis, existing in the realm of computability, is not a completeness result. It is just an *empirical* belief, rather than a provable result—unlike Gödel's completeness theorem. For example, Kalmár (1957) and Péter (1967) have argued that it is conceivable that the intuitive concept of calculability may in the future be extended so much as to transcend the power of the various mathematical models of computation that we currently know.

[51] The level is "higher" the more the programming language is distanced from machine-dependent details.

functions. It evolved later [Kleene (1936)] into what we nowadays call *partial recursive functions*. The definition of computable functions as members of some $\text{Cl}(\mathcal{I}, \mathcal{O})$ is very elegant mathematically [cf. Tourlakis (1984)], but is less intuitively immediate, whereas the programming approach has the attraction of appearing "natural" to those who have done some programming.

We now embark on defining the high level programming language *URM*. The alphabet of the language is

$$\leftarrow, +, \dot{-}, :, X, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathbf{if}, \mathbf{else}, \mathbf{goto}, \mathbf{stop} \qquad (1)$$

Just like any other high level programming language, URM manipulates the contents of *variables*. However, these are restricted to be of *natural number type*—i.e., the only data type that such variables can denote (or "hold", or "contain", in programming jargon) are members of $\mathbb{N}$. Since this programming language is for theoretical considerations only—rather than practical implementation—every variable is allowed to hold *any natural number* whatsoever, without limitations to its size, hence the "UR" in the language name ("unbounded register", used synonymously with *variable of unbounded capacity*).

The syntax of the variables is simple: A variable (name) is a string that starts with X and continues with one or more 1:

$$\text{URM variable set:} \quad X1, X11, X111, X1111, \ldots \qquad (2)$$

Nevertheless, as is customary for the sake of convenience, we will utilize the bold face lower case letters $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{u}, \mathbf{v}, \mathbf{w}$, with or without subscripts or primes as *metavariables* in most of our discussions of the URM, and in examples of specific programs (where yet more, convenient metanotations for variables may be employed).

A URM program is a finite (ordered) sequence of instructions (or commands) of the following five types:

$$\begin{aligned} &L: \ \mathbf{x} \leftarrow a \\ &L: \ \mathbf{x} \leftarrow \mathbf{x} + 1 \\ &L: \ \mathbf{x} \leftarrow \mathbf{x} \dot{-} 1 \\ &L: \ \mathbf{stop} \\ &L: \ \mathbf{if} \ \mathbf{x} = 0 \ \mathbf{goto} \ M \ \mathbf{else} \ \mathbf{goto} \ R \end{aligned} \qquad (3)$$

where L, M, R, a, written in decimal notation, are in $\mathbb{N}$, and $\mathbf{x}$ is some variable. We call instructions of the last type *if-statements*.

Each instruction in a URM program must be numbered by its *position number*, L, in the program, where ":" separates the position number from the instruction. We call these numbers *labels*. Thus, the label of the first instruction is always "1". The instruction **stop** must occur only once in a program, as the last instruction.

The semantics of each command is given in the context of a URM *computation*. The latter we will let have its intuitive meaning in this subsection, and we will defer a mathematical definition until Section 2.3, where such a definition will be needed.

Thus, *for now*, a computation is the process that cycles along the instructions of a program, during which process each instruction that is visited upon—the *current instruction*—causes an *action* that we usually term "the result of the execution" of the instruction. I said "cycles along" because instructions of the last two types (may) cause the computation to loop back or cycle, revisiting an instruction that was already visited by the computation.

Every computation begins with the instruction labeled "1" as the *current* instruction. The semantic action of instructions of each type is defined if and only if they are current, and is as follows:

(i) $L : \mathbf{x} \leftarrow a$. Action: The value of $\mathbf{x}$ becomes the (natural) number a. Instruction $L + 1$ will be the next current instruction.

(ii) $L : \mathbf{x} \leftarrow \mathbf{x} + 1$. Action: This causes the value of $\mathbf{x}$ to increase by 1. The instruction labeled $L + 1$ will be the next current instruction.

(iii) $L : \mathbf{x} \leftarrow \mathbf{x} \mathbin{\dot-} 1$. Action: This causes the value of $\mathbf{x}$ to decrease by 1, *if* it was originally non zero. Otherwise it remains 0. The instruction labeled $L + 1$ will be the next current instruction.

(iv) $L : \mathbf{stop}$. Action: No variable (referenced in the program) changes value. The next current instruction is still the one labeled L.

(v) $L : \mathbf{if}\ \mathbf{x} = 0\ \mathbf{goto}\ M\ \mathbf{else}\ \mathbf{goto}\ R$. Action: No variable (referenced in the program) changes value. The next current instruction is numbered M if $\mathbf{x} = 0$; otherwise it is numbered R.

This command is syntactically illegal (meaningless) if any of M or R exceed the label of the program's $\mathbf{stop}$ instruction.

We say that a computation *terminates*, or *halts*, iff it ever *makes current* (as we say "reaches") the instruction $\mathbf{stop}$. Note that the semantics of "$L : \mathbf{stop}$" *appear* to require the computation to continue *ad infinitum*, but it does so in a trivial manner where no variable changes value, and the current instruction remains the same: Practically, the computation is over.

One usually gives names to URM programs, or as we just say, "to URMs", such as M, N, P, Q, R, F, H, G.

2.1.1.1 Definition. (Computing a Function) We say that a URM, M, *computes* a function $f : \mathbb{N}^n \to \mathbb{N}$ of n arguments *provided*—for some choice of variables $\mathbf{x_1}, \ldots, \mathbf{x_n}$ of M that we designate as *input variables* and a choice of one variable $\mathbf{y}$ that we designate as *the output variable*—the following precise conditions hold for *every choice* of input sequence (or n-tuple), $a_1, \ldots, a_n$ from $\mathbb{N}$:

(1) We *initialize the computation*, by doing two things:

    (a) We *initialize* the input variables with the input values $a_1, \ldots, a_n$. We *initialize all other variables* of M to be 0.

   (b) We next make the instruction labeled "1" current, and thus start the computation.

(2) The computation terminates iff $f(a_1, \ldots, a_n)$ is defined, or, symbolically, iff $f(a_1, \ldots, a_n) \downarrow$ (cf. p. 43).

(3) If the computation terminates, that is, if at some point the instruction **stop** becomes current, then the value of **y** at that point (and hence at any future point, by (iv) above), is $f(a_1, \ldots, a_n)$.   □

(1) We recall that the notation "$f(a_1, \ldots, a_n) \uparrow$" means that $f(a_1, \ldots, a_n)$ is undefined (cf. p. 43).

(2) The function computed by a URM, M, with inputs and output designated as above, can also be denoted by the symbol $M_{\mathbf{y}}^{\mathbf{x_1}, \ldots, \mathbf{x_n}}$. This symbol, with no need for comment, makes it clear as to which are the input variables (superscript) of M, and which is the output variable (subscript). The variables $\mathbf{x_1}, \ldots, \mathbf{x_n}$ in $M_{\mathbf{y}}^{\mathbf{x_1}, \ldots, \mathbf{x_n}}$ are "apparent", or not free for substitution, since $M_{\mathbf{y}}^{\mathbf{x_1}, \ldots, \mathbf{x_n}}$ is not a term (in the logic sense of the word; cf. p. 3); it does not denote an object value. Note also that any attempt to effect such substitutions, for example, $M_{\mathbf{y}}^{\mathbf{3}, \mathbf{x_2}, \ldots, \mathbf{x_n}}$, would lead, in general, to nonsensical situations like "$L : 3 \leftarrow 3 + 1$", a command that wants to change the (standard) value of the symbol "3" (from 3 to 4)!

   Thus, we may write $f = M_{\mathbf{y}}^{\mathbf{x_1}, \ldots, \mathbf{x_n}}$, but *not* $f(a_1, \ldots, a_n) = M_{f(a_1, \ldots, a_n)}^{a_1, \ldots, a_n}$.

   Note that f denotes, by name, a function, that is, a *potentially infinite table* of input/output pairs, where the input is always an n-tuple. On the other hand, $M_{\mathbf{y}}^{\mathbf{x_1}, \ldots, \mathbf{x_n}}$ goes a step further: It *finitely represents the table* f, being able to do so because it is a finite set of instructions that can be used to compute the output for each input where f is defined.

2.1.1.2 Definition. (Computable Functions) A function $f : \mathbb{N}^n \to \mathbb{N}$ of n variables $x_1, \ldots, x_n$ is called *partial computable* iff for some URM, M, we have $f = M_{\mathbf{y}}^{\mathbf{x_1}, \ldots, \mathbf{x_n}}$. The set of all partial computable functions is denoted by $\mathcal{P}$. The set of all the *total* functions in $\mathcal{P}$—that is, those that are defined on *all inputs* from $\mathbb{N}$—is the set of *computable* functions and is denoted by $\mathcal{R}$. The term *recursive* is used in the literature synonymously with the term *computable*.   □

2.1.1.3 Remark. Note that since a URM is a *theoretical*, rather than practical, model of computation we do *not* include *human-computer interface* considerations in the computation. Thus, the "input" and "output" phases just happen during initialization—they are *not* part of the computation. That is why we have dispensed with both **read** and **write** instructions and speak instead of *initialization* in (1) of 2.1.1.1. This approach to input/output is entirely analogous with the input/output convention for the other well-known model of computation, the Turing machine [cf. Davis (1958); Hopcroft *et al.* (2007); Lewis and Papadimitriou (1998); Sipser (1997); Tourlakis (1984)].   □

2.1.1.4 Example. Let M be the program

$$1 : \mathbf{x} \leftarrow \mathbf{x} + 1$$
$$2 : \mathbf{stop}$$

Then $M_{\mathbf{x}}^{\mathbf{x}}$ is the function f given, for all $x \in \mathbb{N}$, by $f(x) = x + 1$, the *successor* function. $\square$

2.1.1.5 Remark. (λ **Notation**) To avoid saying verbose things such as "$M_{\mathbf{x}}^{\mathbf{x}}$ is the function f given, for all $x \in \mathbb{N}$, by $f(x) = x + 1$", we will often use Church's λ-notation and write instead "$M_{\mathbf{x}}^{\mathbf{x}} = \lambda x.x + 1$".

In general, the notation "$\lambda \cdots .$" marks the beginning of a sequence of input variables "$\cdots$" by the symbol "λ", and the end of the sequence by the symbol "$.$" What comes after the period "$.$" is the "rule" that indicates how the output relates to the input. The template for λ-notation thus is

$$\lambda \text{"input"}.\text{"output-rule"}$$

Relating to the above example, we note that $f = \lambda x.x + 1 = \lambda y.y + 1 = \lambda z.f(z)$ is correct—although the rule "$y + 1$" is more informative than "$f(z)$". To the left and right of each "$=$" we have (a symbol for) the table of a function, and we are saying that *all these tables are the same*. Note that x, y, z are "apparent" variables ("dummy" or bound) and are not free (for substitution). In particular, $f = f(x)$ is incorrect as we have distinct data types to the left and right of "$=$", namely, a table on one hand and a number on the other (albeit an unspecified number).

Pause. Why bother with these notational acrobatics?◄

Because well-chosen notation protects against meaningless statements, such as

$$M_{\mathbf{x}}^{\mathbf{x}} = \mathbf{x} + 1 \tag{1}$$

that one might make in the context of the above example. As remarked before, "$M_{\mathbf{x}}^{\mathbf{x}}$" is not a term, nor are the occurrences of x in it free (for substitution). For example, "$M_3^3 = 4$" [obtained by substituting 3 for x throughout in (1)] is totally meaningless, as it says

$$\begin{matrix} 1 : 3 \leftarrow 3 + 1 \\ 2 : \mathbf{stop} \end{matrix} = 4$$

However, $M_{\mathbf{x}}^{\mathbf{x}} = \lambda \mathbf{x}.\mathbf{x} + 1$ *does* make syntactic *and* semantic sense; indeed it is true, as two tables are compared and are found to be equal! Since $\lambda \mathbf{x}.\mathbf{x} + 1 = \lambda y.y + 1$,

the following three tables are identical:[52]

$\mathbf{x} \to \boxed{M}$	$\boxed{M} \to \mathbf{x}$
0	1
1	2
2	3
3	4
$\vdots$	$\vdots$

Input	Output
$\mathbf{x}$	$\mathbf{x}+1$
0	1
1	2
2	3
3	4
$\vdots$	$\vdots$

Input	Output
y	$y+1$
0	1
1	2
2	3
3	4
$\vdots$	$\vdots$

In programming circles, the distinction between *function definition* or *declaration*, $\lambda \vec{x}.f(\vec{x})$, and *function invocation* (or *call*, or *application*, or "use")—what we call a *term*, $f(\vec{x})$, in logic parlance—is well established. The definition part, in programming, uses various notations depending on the programming language and corresponds to writing a program that implements the function, just as we did with M here.

However, there is a double standard in notation when it comes to relations. Extensionally, a relation R is a table (i.e., set) of n-tuples. Its counterpart in formal logic is a formula (cf. the discussion following Definition 1.2.0.4). But where in formal logic we rather infrequently write a formula A as $A[x]$—doing so only if we want to draw attention to our interest in its (free) variable x—in the metatheory we most frequently write a relation R as $R(\vec{x}_n)$, *without* employing λ notation, to draw attention to its "input slots", which here are $x_1, \ldots, x_n$ (i.e., its "free variables").

Since stating "$R(\vec{a}_n)$", by convention, is short for stating "$\langle \vec{a}_n \rangle \in R$", we have two notations for a relation: *Logical* or *relational*, i.e., $R(\vec{x}_n)$, and *set-theoretic*, i.e., $\langle \vec{x}_n \rangle \in R$, both without the benefit of λ notation. There are exceptions to this practice, for example, when we define one relation from another one via the process of "freezing" some of the original relation's inputs. For example, writing $x < y$ (the standard "less than" on $\mathbb{N}$) means that *both* x and y are meant to be inputs; we have a table of ordered pairs. However, we will write $\lambda x.x < y$ to convey that y is fixed and that the input is just x. Clearly, a different relation arises for each y; we have an infinite family of tables: For $y = 0$ we have the empty table; for $y = 1$ one that contains just 0; for $y = 2$ one that contains just $0, 1$; etc.   □

We wrote on p. 41

> Thus, the relation (table) establishes a *one-to-many* input/output correspondence. Contrary to our viewpoint with formulae $\mathscr{A}(x, y)$—where the *input* variables are all the free variables, here x and y—in the case of relations we are allowed *two points of view*; one being the one presented above, and the other where *both* x and y are the inputs of the relation $R(x, y)$. The context will fend for us!

---

[52]$\mathbf{x} \to \boxed{M}$ means "$\mathbf{x}$ is input to M" and $\boxed{M} \to \mathbf{x}$ indicates "$\mathbf{x}$ is output from M".

Our attitude in computability—the "context"—will be to view n-ary relations not as one-to-many correspondences

$$\langle \vec{x}_{n-1} \rangle \longrightarrow \boxed{\quad R \quad} \longrightarrow x_n$$

with $n-1$ input-variables and one output variable, but rather as formulae $R(\vec{x}_n)$ of n free variables. This is already implicit in Remark 2.1.1.5 above, where we said "writing $x < y$ (the standard "less than" on $\mathbb{N}$) means that *both* x and y are meant to be inputs" and "we will write $\lambda x.x < y$ to convey that y is fixed and that the input is just x".

2.1.1.6 Example. Let M be the program

$$1 : \mathbf{x} \leftarrow \mathbf{x} \mathbin{\dot-} 1$$
$$2 : \mathbf{stop}$$

Then $M_{\mathbf{x}}^{\mathbf{x}}$ is the function $\lambda x.x \mathbin{\dot-} 1$, the *predecessor* function. The operation $\dot-$ is called "proper subtraction" and is in general defined by

$$x \mathbin{\dot-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

It ensures that subtraction (as modified) does not take us out of the set of the so-called *number-theoretic* functions, which are those with inputs from $\mathbb{N}$ and outputs in $\mathbb{N}$.

$\square$

Pause. Why are we restricting computability theory to number-theoretic functions? Surely, in practice we can compute with negative numbers, rational numbers, and with nonnumerical entities, such as graphs, trees, etc. Theory ought to reflect, and explain, our practices, no?◄

It does. Negative numbers and rational numbers can be coded by natural number pairs. Computability of number-theoretic functions can handle such pairing (and unpairing or decoding). Moreover, finite objects such as graphs, trees, and the like that we manipulate via computers can be also coded (and decoded) by natural numbers. After all, the internal representation of all data in computers is, at the lowest level, via natural numbers represented in binary notation. Computers cannot handle infinite objects such as (irrational) real numbers. But there is an extensive "higher type" computability theory [which originated with the work of Kleene (1943)] that can handle such numbers as inputs and also compute with them. However, this theory is beyond our scope.

2.1.1.7 Example. Let M be the program

$$1 : \mathbf{x} \leftarrow 0$$
$$2 : \mathbf{stop}$$

Then $M_{\mathbf{x}}^{\mathbf{x}}$ is the function $\lambda x.0$, the *zero function*. □

In Definition 2.1.1.2 we spoke of partial computable and total computable functions. We retain the qualifiers *partial* and *total* for all number-theoretic functions, even for those that may not be computable. Indeed, *total* vs. *nontotal* (no hyphen) has been defined with respect to some assumed left field for any relation, single-valued or not (cf. p. 42).

The set union of all total *and* nontotal number-theoretic functions is the set of all *partial (number-theoretic) functions*. Thus *partial* is *not* synonymous with *nontotal*.

2.1.1.8 Example. The *unconditional* **goto** instruction, namely, "L : **goto** L'", can be simulated by L : **if** $\mathbf{x} = 0$ **goto** L' **else goto** L'. □

2.1.1.9 Example. Let M be the program segment

$$
\begin{aligned}
&k - 1 : \mathbf{x} \leftarrow 0 \\
&k : \quad\;\; \mathbf{x} \leftarrow \mathbf{x} + 1 \\
&k + 1 : \mathbf{z} \leftarrow \mathbf{z} \dot{-} 1 \\
&k + 2 : \textbf{if } \mathbf{z} = 0 \textbf{ goto } k + 3 \textbf{ else goto } k \\
&k + 3 : \dots
\end{aligned}
$$

What it does, by the time the computation reaches instruction $k + 3$, is to have set the value of $\mathbf{z}$ to 0, and to make the value of $\mathbf{x}$ equal to the value that $\mathbf{z}$ had when instruction $k - 1$ was current. In short, the above sequence of instructions simulates the following sequence

$$
\begin{aligned}
&L : \quad\;\; \mathbf{x} \leftarrow \mathbf{z} \\
&L + 1 : \mathbf{z} \leftarrow 0 \\
&L + 2 : \dots
\end{aligned}
$$

where the semantics of L : $\mathbf{x} \leftarrow \mathbf{z}$ are standard in programming: They require that upon execution of the instruction the value of $\mathbf{z}$ is copied into $\mathbf{x}$, but the value of $\mathbf{z}$ remains unchanged. □

2.1.1.10 Exercise. Write a program segment that simulates precisely L : $\mathbf{x} \leftarrow \mathbf{z}$; that is, copy the value of $\mathbf{z}$ into $\mathbf{x}$ without causing $\mathbf{z}$ to change as a side-effect. □

Because of the above, without loss of generality, one may assume that any input variable, $\mathbf{x}$, of a program M is *read-only*. This means that its value remains invariant throughout any computation of the program. Indeed, if $\mathbf{x}$ is not so, a new input variable, $\mathbf{x}'$, can be introduced as follows to relieve $\mathbf{x}$ from its input role: Add at the very beginning of M the (derived) instruction $1 : \mathbf{x} \leftarrow \mathbf{x}'$ of Exercise 2.1.1.10, where $\mathbf{x}'$ is a variable that does not occur in M. Adjust all the following labels consistently, including, of course, the ones referenced by if-statements—a tedious but straightforward task. Call M' the so-obtained URM. Clearly, $M'\,{}_{\mathbf{z}}^{\mathbf{x}',\mathbf{y}_1,\dots,\mathbf{y}_n} = M_{\mathbf{z}}^{\mathbf{x},\mathbf{y}_1,\dots,\mathbf{y}_n}$, and M' does not change $\mathbf{x}'$.

2.1.1.11 Example. (Composing Computable Functions) Suppose that $\lambda x \vec{y}. f(x, \vec{y})$ and $\lambda \vec{z}. g(\vec{z})$ are partial computable, and say $f = F_{\mathbf{u}}^{\mathbf{x}, \vec{\mathbf{y}}}$ while $g = G_{\mathbf{x}}^{\vec{\mathbf{z}}}$.

Since we can rewrite any program, renaming its variables at will, we assume without loss of generality that $\mathbf{x}$ is the only variable common to F and G. Thus, if we concatenate the programs G and F in that order, and (1) remove the last instruction of G (k : **stop**, for some k)—call the program segment that results from this G', and (2) renumber the instructions of F as $k, k+1, \ldots$ (and, as a result, the references that if-statements of F make) in order to give $(G' F)$ the correct program structure, then, $\lambda \vec{y} \vec{z}. f(g(\vec{z}), \vec{y}) = (G' F)_{\mathbf{u}}^{\vec{\mathbf{y}}, \vec{\mathbf{z}}}$. Note that all non-input variables of F will still hold 0 as soon as the execution of $(G' F)$ makes the first instruction of F current *for the first time*. This is because none of these can be changed by G' under our assumption, thus ensuring that F works as designed.   □

Thus, we have, by repeating the above a finite number of times:

2.1.1.12 Proposition. *If $\lambda \vec{y}_n. f(\vec{y}_n)$ and $\lambda \vec{z}. g_i(\vec{z})$, for $i = 1, \ldots, n$, are partial computable, then so is $\lambda \vec{z}. f(g_1(\vec{z}), \ldots, g_n(\vec{z}))$.*

We can rephrase 2.1.1.12, saying simply that $\mathcal{P}$ *is closed under composition*. For the record, we will define *composition* to mean the somewhat rigidly defined operation used in 2.1.1.12, that is:

2.1.1.13 Definition. Given any partial functions (computable or not) $\lambda \vec{y}_n. f(\vec{y}_n)$ and $\lambda \vec{z}. g_i(\vec{z})$, for $i = 1, \ldots, n$, we say that $\lambda \vec{z}. f(g_1(\vec{z}), \ldots, g_n(\vec{z}))$ is the result of their *composition*.   □

We characterized the definition as "rigid". Indeed, note that it requires *all* the arguments of f to be substituted by some $g_i(\vec{z})$—unlike Example 2.1.1.11, where we *substituted* a function invocation (cf. terminology in 2.1.1.5) only in *one* variable of f there, and did nothing with the variables $\vec{y}$—and for each application $g_i(\ldots)$ the argument list, "$\ldots$", *must be the same*, for example $\vec{z}$. That is, in computability we only say, technically, that "we do composition" if—in the sense of 1.2.0.18—we take an $f : \mathbb{N}^n \to \mathbb{N}$ and the n-tuple-valued function $\langle g_1, \ldots, g_n \rangle : \mathbb{N}^m \to \mathbb{N}^n$ [53] and form $\langle g_1, \ldots, g_n \rangle \circ f$.

As we will show in examples in the next subsection (2.1.2), this rigidity is only apparent.

Composing a number of times that *depends on the value of an input variable*—or as we may say, a variable number of times—is *iteration*. The general case of iteration is called *primitive recursion*.

2.1.1.14 Definition. (Primitive Recursion) A number-theoretic function f is defined by *primitive recursion* from given functions $\lambda \vec{y}. h(\vec{y})$ and $\lambda x \vec{y} z. g(x, \vec{y}, z)$ pro-

---

[53]For each m-tuple $\vec{z}$, $\langle g_1, \ldots, g_n \rangle (\vec{z}) = \langle g_1(\vec{z}), \ldots, g_n(\vec{z}) \rangle$.

vided, *for all* $x, \vec{y}$, its values are given by the two equations below:

$$f(0, \vec{y}) \quad \simeq h(\vec{y})$$
$$f(x+1, \vec{y}) \simeq g(x, \vec{y}, f(x, \vec{y}))$$

h is the *basis function*, while g is the *iterator*. In 1.7.0.27 we saw that a unique f that satisfies the above schema exists. Moreover, if both h and g are total, then so is f and the $\simeq$ in the schema may be replaced by $=$ (cf. the exercise following 1.7.0.27).

It will be useful to use the notation $f = prim(h, g)$ to indicate in shorthand that f is defined as above from h and g (note the order). □

Note that $f(1, \vec{y}) \simeq g(0, \vec{y}, h(\vec{y}))$, $f(2, \vec{y}) \simeq g(1, \vec{y}, g(0, \vec{y}, h(\vec{y})))$, $f(3, \vec{y}) \simeq g(2, \vec{y}, g(1, \vec{y}, g(0, \vec{y}, h(\vec{y}))))$, etc. Thus the "$x$-value", 0, 1, 2, 3, etc., equals the number of times we compose g with itself (i.e., the number of times we iterate g).

2.1.1.15 Example. (Iterating Computable Functions) Suppose that $\lambda x \vec{y} z.g(x, \vec{y}, z)$ and $\lambda \vec{y}.h(\vec{z})$ are partial computable, and, say, $g = G_{\mathbf{z}}^{\mathbf{i}, \vec{\mathbf{y}}, \mathbf{z}}$ while $h = H_{\mathbf{z}}^{\vec{\mathbf{y}}}$.

By earlier remarks we may assume:

(i) The only variables that H and G have in common are $\mathbf{z}, \vec{\mathbf{y}}$.

(ii) The variables $\vec{\mathbf{y}}$ are read-only in both H and G.

(iii) $\mathbf{i}$ is read-only in G.

(iv) $\mathbf{x}$ does not occur in any of H or G.

We can now argue that the following program, let us call it F, computes f defined as in 2.1.1.14 from h and g, where $\boxed{H'}$ is program H with the **stop** instruction removed, $\boxed{G'}$ is program G that has the **stop** instruction removed, and instructions renumbered (and if-statements adjusted) as needed:

$$\boxed{H'}$$

$$
\begin{array}{ll}
r : & \mathbf{i} \leftarrow 0 \\
r + 1 : & \text{if } \mathbf{x} = 0 \text{ goto } k + m + 2 \text{ else goto } r + 2 \\
r + 2 : & \mathbf{x} \leftarrow \mathbf{x} \mathbin{\dot{-}} 1
\end{array}
$$

$$\boxed{G'}$$

$$
\begin{array}{ll}
k : & \mathbf{i} \leftarrow \mathbf{i} + 1 \\
k + 1 : & \mathbf{w}_1 \leftarrow 0 \\
\vdots & \\
k + m : & \mathbf{w}_m \leftarrow 0 \\
k + m + 1 : & \text{goto } r + 1 \\
k + m + 2 : & \text{stop}
\end{array}
$$

The instructions $\mathbf{w}_i \leftarrow 0$ set explicitly to zero all the variables of G' other than $\mathbf{i}, \mathbf{z}, \vec{\mathbf{y}}$ to ensure correct behavior of G'. Note that the $\mathbf{w}_i$ are *implicitly* initialized to zero *only* the first time G' is executed. Clearly, $f = F_{\mathbf{z}}^{\mathbf{x}, \vec{\mathbf{y}}}$. □

We have at once:

2.1.1.16 Proposition. *If f, g, h relate as in Definition 2.1.1.14 and h and g are in $\mathcal{P}$, then so is f. We say that $\mathcal{P}$ is* closed under primitive recursion.

2.1.1.17 Example. (Unbounded Search) Suppose that $\lambda x \vec{y}.g(x, \vec{y})$ is partial computable, and, say, $g = G_{\mathbf{z}}^{\mathbf{x}, \vec{\mathbf{y}}}$. By earlier remarks we may assume that $\vec{\mathbf{y}}$ and $\mathbf{x}$ are read-only in G and that $\mathbf{z}$ is *not* one of them.

Consider the following program F, where $\boxed{G'}$ is program G with the **stop** instruction removed, and instructions have been renumbered (and if-statements adjusted) as needed so that its first command has label 2.

$$1:\qquad \mathbf{x} \leftarrow 0$$

$$\boxed{G'}$$

$k:\qquad$ if $\mathbf{z} = 0$ **goto** $k + l + 3$ **else goto** $k + 1$
$k+1:\qquad \mathbf{w}_1 \leftarrow 0$ {**Comment.** Setting all non-input variables to 0; cf. 2.1.1.15.}
$\vdots$
$k+l:\qquad \mathbf{w}_l \leftarrow 0$ {**Comment.** Setting all non-input variables to 0; cf. 2.1.1.15.}
$k+l+1: \mathbf{x} \leftarrow \mathbf{x} + 1$
$k+l+2: \textbf{goto } 2$
$k+l+3: \textbf{stop}$

Let us set $f = F_{\mathbf{x}}^{\vec{\mathbf{y}}}$. Note that, for any $\vec{a}$, $f(\vec{a}) \downarrow$ precisely if the URM F, initialized with $\vec{a}$ as the input values in $\vec{\mathbf{y}}$, ever reaches **stop**. This condition becomes true as long as the two conditions, (1) and (2) below, are fulfilled:

(1) Instruction k just found that $\mathbf{z}$ holds 0. This value of $\mathbf{z}$ is the result of an execution of G (i.e., G' with the **stop** instruction added) with input values $\vec{a}$ in $\vec{\mathbf{y}}$ and, say, b in $\mathbf{x}$, the latter being the *iteration counter*—$0, 1, 2, \ldots$—that indicates how many times instruction 2 becomes current.

(2) In none of the previous iterations (with $\mathbf{x}$-value $< b$) did G' (essentially, G) get into a non-ending computation (*infinite loop*).

Correspondingly, the computation of F will never halt for an input $\vec{a}$ if either G loops for ever at some step, or, if it halts in every iteration b, but nevertheless it never exits with a $\mathbf{z}$-value of 0.

Thus, for all $\vec{a}$,

$$f(\vec{a}) = \min \left\{ x : g(x, \vec{a}) = 0 \wedge (\forall y)\big(y < x \rightarrow g(y, \vec{a}) \downarrow \big) \right\} \qquad \square$$

2.1.1.18 Definition. The operation on partial functions g given for all $\vec{a}$ by

$$\min \left\{ x : g(x, \vec{a}) = 0 \wedge (\forall y)\big(y < x \rightarrow g(y, \vec{a}) \downarrow \big) \right\}$$

is called *unbounded search* (along the variable x) and is denoted by the symbol $(\mu x)g(x, \vec{a})$. The function $\lambda \vec{y}.(\mu x)g(x, \vec{y})$ is defined precisely when the minimum exists. $\qquad \square$

The result of Example 2.1.1.17 yields at once:

2.1.1.19 Proposition. $\mathcal{P}$ *is closed under unbounded search; that is, if* $\lambda x\vec{y}.g(x, \vec{y})$ *is in* $\mathcal{P}$, *then so is* $\lambda \vec{y}.(\mu x)g(x, \vec{y})$.

Why "unbounded" search? Because we do not know *a priori* how many times we have to go around the loop. This depends on the behavior of g.

2.1.1.20 Example. Is the function $\lambda \vec{x}_n.x_i$, where $1 \le i \le n$, in $\mathcal{P}$? Yes, and here is a program, M, for it:

$$1 : \qquad \mathbf{w}_1 \leftarrow 0$$
$$\vdots$$
$$i : \qquad \mathbf{z} \leftarrow \mathbf{w}_i \ \{\textbf{Comment. } \text{Cf. Exercise 2.1.1.10}\}$$
$$\vdots$$
$$n : \qquad \mathbf{w}_n \leftarrow 0$$
$$n + 1 : \textbf{stop}$$

$\lambda \vec{x}_n.x_i = M_{\mathbf{z}}^{\vec{\mathbf{w}}_n}$. To ensure that M indeed *has* the $\mathbf{w}_i$ as variables we reference them in instructions at least once, in any manner whatsoever. $\qquad\square$

2.1.2 Primitive Recursive Functions

Exercises 2.1.1.4, 2.1.1.7, and 2.1.1.20 show that the successor, the zero, and the *generalized identity* functions respectively—which we will often name S, Z and U_i^n respectively—are in $\mathcal{P}$; thus, not only are they "intuitively computable", but they are so in a precise mathematical sense. We have also shown that "computability" of functions is preserved by the operations of composition, primitive recursion, and unbounded search. In this subsection we will explore the properties of the important set of functions known as *primitive recursive*. Most people introduce them by derivations just as one introduces the theorems of logic, as in the definition below.

Note that the "U" (u = unit) in U_i^n is suggested by the behavior of the function as the identity or *unit* function[54] that, essentially, takes x to x, but has additional "non active" or "don't care" inputs. These unit functions are also called *projection functions* and are indeed the the p_i^n of Exercise 1.8.23, but here we employ them in the special setting where $A_j = \mathbb{N}$, for $j = 1, \ldots, n$. Unfortunately, the term "projection function" in computability clashes with the names of the specialized K and L functions associated with pairing functions (cf. 2.1.4.1).

2.1.2.1 Definition. ($\mathcal{PR}$-**derivations;** $\mathcal{PR}$-**functions**) A $\mathcal{PR}$-derivation is a finite sequence of number-theoretic functions that obeys, in its step-by-step construction, the following requirements. At each step we may write:

---

[54]The identity function behaves with respect to composition like the number 1 does with respect to multiplication; as a "unit". Cf. 1.8.25.

(1) Any one of Z, S, U_i^n (for any $n > 0$ and any $0 < i \leq n$).

(2) $\lambda \vec{z}.f(g_1(\vec{z}), \ldots, g_n(\vec{z}))$, *provided* each of $f, g_1, \ldots, g_n$ has already been written.

(3) $prim(h, g)$, provided appropriate h and g have already been written. Note that h and g are "appropriate" (cf. 2.1.1.14) as long as g has two more arguments than h.

A function f is *primitive recursive*, or a $\mathcal{PR}$-function, iff it occurs in some $\mathcal{PR}$-derivation. The set of functions allowed in step (1) are called *initial functions*. We will denote this set by $\mathcal{I}$. The set of *all* $\mathcal{PR}$-functions will be denoted by $\mathcal{PR}$.   □

2.1.2.2 Remark. The above definition defines essentially what Dedekind (1888) called "recursive" functions. Subsequently they have been renamed *primitive recursive* allowing the unqualified term *recursive* to be synonymous with *computable* and apply to the functions of $\mathcal{R}$ (cf. 2.1.1.2).

The concept of a $\mathcal{PR}$-derivation is entirely analogous with that of proof. Vis-à-vis proofs, derivations have the following analogous elements: initial functions (vs. axioms) and the operations composition and primitive recursion (vs. the rules of inference).

As it is the case with proofs, we can cut the tail off a derivation and still have a derivation. The reason is immediate: If the legitimacy of appearance of a function f in some $\mathcal{PR}$-derivation is based on (2) or (3) of 2.1.2.1, then the presence or absence of the "tail" after f in said derivation is totally irrelevant to f's legitimacy of appearance. Thus, we deduce at once that a $\mathcal{PR}$-function is one that appears at the *end* of a $\mathcal{PR}$-derivation.

Properties of primitive recursive functions can be proved by induction on derivation length, just as properties of theorems can be proved by induction on the length of proofs.

That a certain function is primitive recursive can be proved by exhibiting a derivation for it, just as is done for the certification of a theorem: We exhibit a proof. However, in proving theorems we accept the use of known theorems in proofs. Similarly, if we know that certain functions are primitive recursive, then we immediately infer that so is one obtained from them by an allowed operation (composition, primitive recursion, or yet-to-be-introduced derived operations). For example, if h and g are in $\mathcal{PR}$ and $prim(h, g)$ makes sense according to 2.1.1.14, then the latter is in $\mathcal{PR}$, too, since we can concatenate derivations of h and g and add $prim(h, g)$ to the right end.

In analogy to the case of theorem proving, where we benefit from powerful derived *rules*, in the same way the certification of functions as primitive recursive is greatly facilitated by the introduction of derived *operations* on functions beyond the two that we assumed as given outright (*primary operations*) in Definition 2.1.2.1.   □

The reader will recognize at once that Definition 2.1.2.1 is a "concrete" or special instance of the abstract (or "general") Definitions 1.6.0.6 and 1.6.0.9. Moreover, we must emphasize the fundamental (and practically important) relationship between iterative and recursive definitions of sets (cf. 1.6.0.14). The extremely useful principle of "*induction on* (with respect to) *a closure*" (or structural induction) is based, as

the reader will recall from 1.6.0.13, on the characterization of a set of objects as a closure. The earlier Remark 1.6.0.15 notes when we benefit from the iterative and when from the recursive point of view. It also provides an alternative reason for the correctness of the observation above:

> *Similarly, if we know that certain functions are primitive recursive, then we immediately infer that so is one obtained from them by an allowed operation* ...

Thus we may state at once:

2.1.2.3 Theorem. $\mathcal{PR}$ *is the closure of* $\mathcal{I}$ *under primitive recursion and composition.*

2.1.2.4 Remark. (Induction over $\mathcal{PR}$) For the general principle, cf. 1.6.0.13. We can do *induction over* $\mathcal{PR}$ toward proving a property $\mathscr{P}(f)$ for all $f \in \mathcal{PR}$. We prove:

(1) (*Basis*) All of Z, S, U_i^n (for any $n > 0$ and any $0, i \le n$) has the property $\mathscr{P}$.

(2) $\lambda \vec{z}.f(g_1(\vec{z}), \ldots, g_n(\vec{z}))$ has the property, *provided* each of $f, g_1, \ldots, g_n$ do.

(3) $prim(h, g)$ has the property, provided h and g do.

The above procedure is more elegant (and more widely used) than induction on the length of $\mathcal{PR}$-derivation. □

2.1.2.5 Example. If $\lambda xyw.f(x, y, w)$ and $\lambda z.g(z)$ are in $\mathcal{PR}$, how about $\lambda xzw.f(x, g(z), w)$? It is in $\mathcal{PR}$ since

$$\lambda xzw.f(x, g(z), w) = \lambda xzw.f(U_1^3(x, z, w), g(U_2^3(x, z, w)), U_3^3(x, z, w))$$

and the U_i^n are primitive recursive. The reader will see at once that to the right of "=" we have correctly formed compositions as expected by 2.1.1.13.

Similarly, for the same functions above,

(1) $\lambda yw.f(2, y, w)$ is in $\mathcal{PR}$. Indeed, this function can be obtained by composition, since

$$\lambda yw.f(2, y, w) = \lambda yw.f\Big(SSZ(U_1^2(y, w)), y, w\Big)$$

where I wrote "$SSZ(\ldots)$" as short for $S(S(Z(\ldots)))$ for visual clarity. Clearly, using $SSZ(U_2^2(y, w))$ above works as well.

(2) $\lambda xyw.f(y, x, w)$ is in $\mathcal{PR}$. Indeed, this function can be obtained by composition, since

$$\lambda xyw.f(y, x, w) = \lambda xyw.f\Big(U_2^3(x, y, w), U_1^3(x, y, w), U_3^3(x, y, w)\Big)$$

In this connection, note that while $\lambda xy.g(x, y) = \lambda yx.g(y, x)$, yet $\lambda xy.g(x, y) \ne \lambda xy.g(y, x)$ in general. For example, $\lambda xy.x \dotminus y$ asks that we subtract the second input (y) from the first (x), but $\lambda xy.y \dotminus x$ asks that we subtract the first input (x) from the second (y).

(3) $\lambda xy.f(x,y,x)$ is in $\mathcal{PR}$. Indeed, this function can be obtained by composition, since

$$\lambda xy.f(x,y,x) = \lambda xy.f\big(U_1^2(x,y), U_2^2(x,y), U_1^2(x,y)\big)$$

(4) $\lambda xyzwu.f(x,y,w)$ is in $\mathcal{PR}$. Indeed, this function can be obtained by composition, since

$$\lambda xyzwu.f(x,y,w) =$$
$$\lambda xyzwu.f(U_1^5(x,y,z,w,u), U_2^5(x,y,z,w,u), U_4^5(x,y,z,w,u))$$

$\square$

The above examples are summarized, named, and generalized in the following straightforward exercise:

2.1.2.6 Exercise. (The Grzegorczyk (1953) Substitution Operations) $\mathcal{PR}$ is closed under the following operations:

(i) *Substitution of a function invocation for a variable*:
From $\lambda \vec{x}y\vec{z}.f(\vec{x}, y, \vec{z})$ and $\lambda \vec{w}.g(\vec{w})$ obtain $\lambda \vec{x}\vec{w}\vec{z}.f(\vec{x}, g(\vec{w}), \vec{z})$.

(ii) *Substitution of a constant for a variable*:
From $\lambda \vec{x}y\vec{z}.f(\vec{x}, y, \vec{z})$ obtain $\lambda \vec{x}\vec{z}.f(\vec{x}, k, \vec{z})$.

(iii) *Interchange of two variables*:
From $\lambda \vec{x}y\vec{z}w\vec{u}.f(\vec{x}, y, \vec{z}, w, \vec{u})$ obtain $\lambda \vec{x}y\vec{z}w\vec{u}.f(\vec{x}, w, \vec{z}, y, \vec{u})$.

(iv) *Identification of two variables*:
From $\lambda \vec{x}y\vec{z}w\vec{u}.f(\vec{x}, y, \vec{z}, w, \vec{u})$ obtain $\lambda \vec{x}y\vec{z}\vec{u}.f(\vec{x}, y, \vec{z}, y, \vec{u})$.

(v) *Introduction of "don't care" variables*:
From $\lambda \vec{x}.f(\vec{x})$ obtain $\lambda \vec{x}\vec{z}.f(\vec{x})$. $\square$

By 2.1.2.6 composition can simulate the Grzegorczyk operations if the initial functions $\mathcal{I}$ are present. Of course, (i) alone can in turn simulate composition. With these comments out of the way, we see that the "rigidity" of Definition 2.1.1.13 is gone.

2.1.2.7 Example. The definition of primitive recursion is also rigid, but this rigidity is removable as well. For example, natural and simple recursions such as $p(0) = 0$ and $p(x+1) = x$—this one defining $p = \lambda x.x \mathbin{\dot{-}} 1$—do not fit the schema of Definition 2.1.1.14. This is because it requires the defined function to have one more variable than the basis, so no one-variable function can be directly defined! We can get around this. Define first $\widetilde{p} = \lambda xy.x \mathbin{\dot{-}} 1$ as follows: $\widetilde{p}(0,y) = 0$ and $\widetilde{p}(x+1,y) = x$. Now this can be dressed up according to the syntax of the schema in 2.1.1.14,

$$\begin{aligned}\widetilde{p}(0,y) &= Z(y)\\ \widetilde{p}(x+1,y) &= U_1^3(x,y,\widetilde{p}(x,y))\end{aligned}$$

that is, $\widetilde{p} = prim(Z, U_1^3)$. Then we can get p by (Grzegorczyk) substitution: $p = \lambda x.\widetilde{p}(x, 0)$. Incidentally, this shows that both p and $\widetilde{p}$ are in $\mathcal{PR}$.

Another rigidity in the definition of primitive recursion is that, *apparently*, one can use only the first variable as the iterating variable. Consider, for example, $sub = \lambda xy.x \,\dot-\, y$. Clearly, $sub(x, 0) = x$ and $sub(x, y + 1) = p(sub(x, y))$ is correct semantically, but the format is wrong: We are not supposed to iterate along the second variable! Well, define instead $\widetilde{sub} = \lambda xy.y \,\dot-\, x$:

$$\begin{aligned}
\widetilde{sub}(0, y) &= U_1^1(y) \\
\widetilde{sub}(x + 1, y) &= p\big(U_3^3(x, y, \widetilde{sub}(x, y))\big)
\end{aligned}$$

Then, using variable swapping [Grzegorczyk operation (iii)], we can get sub: $sub = \lambda xy.\widetilde{sub}(y, x)$. Clearly, both $\widetilde{sub}$ and sub are in $\mathcal{PR}$. With practice, one gets used to accepting at once simplified recursions like the one for p and sub. One needs to make them conform to the format of 2.1.1.14 only if the instructor insists! $\square$

2.1.2.8 Exercise. Prove that $\lambda xy.x + y$ and $\lambda xy.x \times y$ are primitive recursive. Of course, we will usually write multiplication $x \times y$ in "implied notation", xy. $\square$

2.1.2.9 Example. The very important "*switch*" (or "if-then-else") function $sw = \lambda xyz.$if $x = 0$ then y else z is primitive recursive. It is directly obtained by primitive recursion on initial functions: $sw(0, y, z) = y$ and $sw(x + 1, y, z) = z$. $\square$

2.1.2.10 Exercise. Dress up the recursion $sw(0, y, z) = y$ and $sw(x + 1, y, z) = z$ to bring it into the format required by Definition 2.1.1.14. $\square$

2.1.2.11 Exercise. Prove by induction first on derivation lengths, and then "over $\mathcal{PR}$", that all functions in $\mathcal{PR}$ are total. $\square$

2.1.2.12 Proposition. $\mathcal{PR} \subseteq \mathcal{R}$.

Proof. By 2.1.1.12, 2.1.1.16, and 2.1.2.3, $\mathcal{PR} \subseteq \mathcal{P}$. But all the functions in $\mathcal{PR}$ are total (cf. 2.1.2.11 and Definition 2.1.1.2). $\square$

Indeed, the above inclusion is proper, as we will see in Subsection 2.4. We also state for the record:

2.1.2.13 Proposition. $\mathcal{R}$ *is closed under both composition and primitive recursion.*

Proof. Because $\mathcal{P}$ is, and both operations conserve totalness. $\square$

2.1.2.14 Example. Consider the function ex given by

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

Thus, if $x = 0$, then $ex(x, 0) = 1$, but $ex(x, y) = 0$ for all $y > 0$. On the other hand, if $x > 0$, then $ex(x, y) = x^y$ for all y.

Note that x^y is "mathematically" undefined when $x = y = 0$.[55] Thus, by Exercise 2.1.2.11 the exponential cannot be a primitive recursive function!

This is rather silly, since the computational process for the exponential is so straightforward; thus it is a shame to declare the function non-$\mathcal{PR}$. After all, we know *exactly where and how it is undefined* and we can remove this undefinability by *redefining "x^y" to mean $ex(x, y)$ for all inputs.*

Clearly $ex \in \mathcal{PR}$. In computability we do this kind of redefinition a lot in order to remove easily recognizable points of "non definition" of calculable functions. We will see further examples, such as the remainder, quotient, and logarithm functions.

Caution! We cannot always remove points of non definition of a calculable function and still obtain a computable function. That is, there are functions $f \in \mathcal{P}$ that *have no recursive extensions*. This we will show in Subsection 2.7.   □

2.1.2.15 Definition. A relation $R(\vec{x})$ is (*primitive*) *recursive* iff its *characteristic function*,

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

is (primitive) recursive. The set of all primitive recursive (respectively, recursive) *relations* is denoted by $\mathcal{PR}_*$ (respectively, $\mathcal{R}_*$).   □

Computability theory practitioners often call relations *predicates*. It is clear that one can go from relation to characteristic function and back in a unique way, since $R(\vec{x}) \equiv \chi_R(\vec{x}) = 0$. Thus, we may think of relations as "0-1 valued" functions. The concept of relation simplifies the further development of the theory of primitive recursive functions.

The following is useful:

2.1.2.16 Proposition. $R(\vec{x}) \in \mathcal{PR}_*$ iff some $f \in \mathcal{PR}$ exists such that, for all $\vec{x}$, $R(\vec{x}) \equiv f(\vec{x}) = 0$.

Proof. For the *if*-part, I want $\chi_R \in \mathcal{PR}$. This is so since $\chi_R = \lambda \vec{x}.1 \dotminus (1 \dotminus f(\vec{x}))$ (using Grzegorczyk substitution and $\lambda xy.x \dotminus y \in \mathcal{PR}$; cf. 2.1.2.7). For the *only if*-part, $f = \chi_R$ will do.   □

2.1.2.17 Corollary. $R(\vec{x}) \in \mathcal{R}_*$ iff some $f \in \mathcal{R}$ exists such that, for all $\vec{x}$, $R(\vec{x}) \equiv f(\vec{x}) = 0$.

Proof. By the above proof, 2.1.2.12, and 2.1.2.13.   □

2.1.2.18 Corollary. $\mathcal{PR}_* \subseteq \mathcal{R}_*$.

Proof. By the above corollary and 2.1.2.12.   □

---

[55] In first-year university calculus we learn that "0^0" is an "indeterminate form".

2.1.2.19 Theorem. $\mathcal{PR}_*$ *is closed under the Boolean operations.*

Proof. It suffices to look at the cases of $\neg$ and $\vee$, since $R \rightarrow Q \equiv \neg R \vee Q$, $R \wedge Q \equiv \neg(\neg R \vee \neg Q)$ and $R \equiv Q$ is short for $(R \rightarrow Q) \wedge (Q \rightarrow P)$.

($\neg$) Say, $R(\vec{x}) \in \mathcal{PR}_*$. Thus (2.1.2.15), $\chi_R \in \mathcal{PR}$. But then $\chi_{\neg R} \in \mathcal{PR}$, since $\chi_{\neg R} = \lambda \vec{x}.1 \dot{-} \chi_R(\vec{x})$, by Grzegorczyk substitution and $\lambda xy.x \dot{-} y \in \mathcal{PR}$.

($\vee$) Let $R(\vec{x}) \in \mathcal{PR}_*$ and $Q(\vec{y}) \in \mathcal{PR}_*$. Then $\lambda \vec{x}\vec{y}.\chi_{R \vee Q}(\vec{x}, \vec{y})$ is given by

$$\chi_{R \vee Q}(\vec{x}, \vec{y}) = \text{if } R(\vec{x}) \text{ then } 0 \text{ else } \chi_Q(\vec{y})$$

and therefore is in $\mathcal{PR}$. □

2.1.2.20 Remark. Alternatively, for the $\vee$ case above, note that $\chi_{R \vee Q}(\vec{x}, \vec{y}) = \chi_R(\vec{x}) \times \chi_Q(\vec{y})$ and invoke 2.1.2.8. □

It is common practice to use $R(\vec{x})$ and $\chi_R(\vec{x})$ (almost) interchangeably. For example, "if $R(\vec{x})$ then …" is the same as "if $\chi_R(\vec{x}) = 0$ then …". The latter more directly shows that a (Grzegorczyk) substitution was effected into an argument of the if-then-else (2.1.2.9) function:

$$\begin{array}{c} \chi_R(\vec{x}) \\ \downarrow \\ \text{if} \quad x \quad = 0 \text{ then } \dots \end{array}$$

thus establishing the primitive recursiveness of the resulting function.

2.1.2.21 Corollary. $\mathcal{R}_*$ *is closed under the Boolean operations.*

Proof. As above, mindful of 2.1.2.12, and 2.1.2.13. □

2.1.2.22 Example. The relations $x \leq y$, $x < y$, $x = y$ are in $\mathcal{PR}_*$. See 2.1.1.5 for a refresher on our conventions regarding lambda notation and relations.

With this out of the way, note that $x \leq y \equiv x \dot{-} y = 0$ and invoke 2.1.2.16. Finally invoke Boolean closure and note that $x < y \equiv \neg y \leq x$ while $x = y$ is equivalent to $x \leq y \wedge y \leq x$. □

2.1.2.23 Proposition. *If* $R(\vec{x}, y, \vec{z}) \in \mathcal{PR}_*$ *and* $\lambda \vec{w}.f(\vec{w}) \in \mathcal{PR}$, *then* $R(\vec{x}, f(\vec{w}), \vec{z})$ *is in* $\mathcal{PR}_*$.

Proof. Let $Q(\vec{x}, \vec{w}, \vec{z})$ denote $R(\vec{x}, f(\vec{w}), \vec{z})$. Then $\chi_Q(\vec{x}, \vec{w}, \vec{z}) = \chi_R(\vec{x}, f(\vec{w}), \vec{z})$. □

2.1.2.24 Proposition. *If* $R(\vec{x}, y, \vec{z}) \in \mathcal{R}_*$ *and* $\lambda \vec{w}.f(\vec{w}) \in \mathcal{R}$, *then* $R(\vec{x}, f(\vec{w}), \vec{z})$ *is in* $\mathcal{R}_*$.

Proof. Similar to that of 2.1.2.23. □

2.1.2.25 Corollary. *If* $f \in \mathcal{PR}$ *(respectively, in* $\mathcal{R}$*), then its* graph, $z = f(\vec{x})$ *is in* $\mathcal{PR}_*$ *(respectively, in* $\mathcal{R}_*$*).*

Proof. Using the relation $z = y$ and 2.1.2.23. □

The following converse of 2.1.2.25, "if $z = f(\vec{x})$ is in $\mathcal{PR}_*$ and f is total, then $f \in \mathcal{PR}$" is *not* true. A counterexample is provided by the Ackermann function. However, "if $z = f(\vec{x})$ is in $\mathcal{R}_*$ and f is total, then $f \in \mathcal{R}$" *is* true. More on the Ackermann function in Section 2.4.

2.1.2.26 Exercise. Using unbounded search, prove that if $z = f(\vec{x})$ is in $\mathcal{R}_*$ and f is total, then $f \in \mathcal{R}$. □

2.1.2.27 Definition. (Bounded Quantifiers) The abbreviations $(\forall y)_{<z} R(z, \vec{x})$ and $(\exists y)_{<z} R(z, \vec{x})$ stand for $(\forall y)(y < z \rightarrow R(z, \vec{x}))$ and $(\exists y)(y < z \wedge R(z, \vec{x}))$, respectively, and similarly for the nonstrict inequality "$\leq$". □

2.1.2.28 Theorem. $\mathcal{PR}_*$ *is closed under bounded quantification.*

Proof. By 2.1.2.19 it suffices to look at the case of $(\exists y)_{<z}$ since $(\forall y)_{<z} R(y, \vec{x}) \equiv \neg(\exists y)_{<z} \neg R(y, \vec{x})$.

Let then $R(y, \vec{x}) \in \mathcal{PR}_*$ and let us give the name $Q(z, \vec{x})$ to $(\exists y)_{<z} R(y, \vec{x})$. We note that $Q(0, \vec{x})$ is false (why?) and $Q(z + 1, \vec{x}) \equiv Q(z, \vec{x}) \vee R(z, \vec{x})$. Thus,

$$\chi_Q(0, \vec{x}) = 1$$
$$\chi_Q(z + 1, \vec{x}) = \chi_Q(z, \vec{x})\chi_R(z, \vec{x})$$
□

2.1.2.29 Corollary. $\mathcal{R}_*$ *is closed under bounded quantification.*

2.1.2.30 Exercise. The operations of *bounded summation* and *bounded multiplication* are quite handy. These are applied on a function f and yield the functions $\lambda z\vec{x}.\sum_{i<z} f(i, \vec{x})$ and $\lambda z\vec{x}.\prod_{i<z} f(i, \vec{x})$, respectively, where $\sum_{i<0} f(i, \vec{x}) = 0$ and $\prod_{i<0} f(i, \vec{x}) = 1$ *by definition.* Prove that $\mathcal{PR}$ and $\mathcal{R}$ are closed under both operations; i.e., if f is in $\mathcal{PR}$ (respectively, in $\mathcal{R}$), then so are $\lambda z\vec{x}.\sum_{i<z} f(i, \vec{x})$ and $\lambda z\vec{x}.\prod_{i<z} f(i, \vec{x})$. □

2.1.2.31 Definition. (Bounded Search) Let f be a total number-theoretic function of $n + 1$ variables. The symbol $(\mu y)_{<z} f(y, \vec{x})$, for all $z, \vec{x}$, stands for

$$\begin{cases} \min\{y : y < z \wedge f(y, \vec{x}) = 0\} & \text{if } (\exists y)_{<z} f(y, \vec{x}) = 0 \\ z & \text{otherwise} \end{cases}$$

We define "$(\mu y)_{\leq z}$" to mean "$(\mu y)_{<z+1}$". □

2.1.2.32 Theorem. $\mathcal{PR}$ *is closed under the bounded search operation* $(\mu y)_{<z}$. *That is, if* $\lambda y\vec{x}.f(y, \vec{x}) \in \mathcal{PR}$, *then* $\lambda z\vec{x}.(\mu y)_{<z} f(y, \vec{x}) \in \mathcal{PR}$.

Proof. Set $g = \lambda z\vec{x}.(\mu y)_{<z} f(y, \vec{x})$. Then the following primitive recursion settles it:

$$g(0, \vec{x}) = 1$$

$$g(z+1, \vec{x}) = \text{if } g(z, \vec{x}) < z \text{ then } g(z, \vec{x})$$
$$\text{else if } f(z, \vec{x}) = 0 \text{ then } z$$
$$\text{else } z+1 \qquad \square$$

2.1.2.33 Exercise. Mindful of the comment following 2.1.2.19, dress up the primitive recursion that defined g above so that it conforms to the Definition 2.1.1.14. $\qquad \square$

2.1.2.34 Corollary. $\mathcal{PR}$ *is closed under the bounded search operation* $(\mu y)_{\leq z}$.

2.1.2.35 Exercise. Prove the corollary. $\qquad \square$

2.1.2.36 Corollary. $\mathcal{R}$ *is closed under the bounded search operations* $(\mu y)_{<z}$ *and* $(\mu y)_{\leq z}$.

Consider now a set of *mutually exclusive* relations $R_i(\vec{x})$, $i = 1, \ldots, n$, that is, $R_i(\vec{x}) \wedge R_j(\vec{x})$ is *false* for each $\vec{x}$ as long as $i \neq j$.

Then we can define a function f *by cases* R_i from given functions f_j by the requirement (for all $\vec{x}$) given below:

$$f(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } R_1(\vec{x}) \\ f_2(\vec{x}) & \text{if } R_2(\vec{x}) \\ \ldots & \ldots \\ f_n(\vec{x}) & \text{if } R_n(\vec{x}) \\ f_{n+1}(\vec{x}) & \text{otherwise} \end{cases}$$

where, as is usual in mathematics, "if $R_j(\vec{x})$" is short for "if $R_j(\vec{x})$ is true" and "otherwise" is the condition $\neg(R_1(\vec{x}) \vee \cdots \vee R_n(\vec{x}))$. We have the following result:

2.1.2.37 Theorem. (Definition by Cases) *If the functions* f_i, $i = 1, \ldots, n+1$ *and the relations* $R_i(\vec{x})$, $i = 1, \ldots, n$ *are in* $\mathcal{PR}$ *and* $\mathcal{PR}_*$, *respectively, then so is* f *above.*

Proof. By repeated use (composition) of if-then-else. Alternatively, by noting—being mindful of 2.1.2.19—that

$$f(\vec{x}) = f_1(\vec{x})(1 \dot{-} \chi_{R_1}(\vec{x})) + \cdots + f_n(\vec{x})(1 \dot{-} \chi_{R_n}(\vec{x})) +$$
$$f_{n+1}(\vec{x})(1 \dot{-} \chi_{\neg(R_1 \vee \cdots \vee R_n)}(\vec{x})) \qquad \square$$

2.1.2.38 Corollary. *Same statement as above, replacing* $\mathcal{PR}$ *and* $\mathcal{PR}_*$ *by* $\mathcal{R}$ *and* $\mathcal{R}_*$, *respectively.*

The tools we now have at our disposal allow easy certification of the primitive recursiveness of some very useful functions and relations. But first a definition:

2.1.2.39 Definition. $(\mu y)_{<z} R(y, \vec{x})$ means $(\mu y)_{<z} \chi_R(y, \vec{x})$. $\qquad \square$

Thus, if $R(y, \vec{x}) \in \mathcal{PR}_*$ (resp. $\in \mathcal{R}_*$), then $\lambda z \vec{x}.(\mu y)_{<z} R(y, \vec{x}) \in \mathcal{PR}$ (resp. $\in \mathcal{R}$), since $\chi_R \in \mathcal{PR}$ (resp. $\in \mathcal{R}$).

2.1.2.40 Example. The following are in $\mathcal{PR}$ or $\mathcal{PR}_*$ as appropriate:

(1) $\lambda xy. \lfloor x/y \rfloor$[56] (the quotient of the division x/y). This is another instance of a nontotal function with an "obvious" way to remove the points where it is undefined (cf. 2.1.2.14). Thus the symbol is extended to *mean* $(\mu z)_{\leq x}\big((z+1)y > x\big)$ for all x, y. It follows that, for $y > 0$, $\lfloor x/y \rfloor$ is as expected in "normal math", while $\lfloor x/0 \rfloor = x + 1$.

(2) $\lambda xy.rem(x, y)$ (the remainder of the division x/y). $rem(x, y) = x \dotdiv y\lfloor x/y \rfloor$.

(3) $\lambda xy.x|y$ (x divides y). $x|y \equiv rem(y, x) = 0$. Note that if $y > 0$, we cannot have $0|y$—a good thing!—since $rem(y, 0) = y$. Our redefinition of $\lfloor x/y \rfloor$ yields, however, $0|0$, but we can live with this in practice.

(4) $Pr(x)$ (x is a prime). $Pr(x) \equiv x > 1 \wedge (\forall y)_{\leq x}(y|x \rightarrow y = 1 \vee y = x)$.

(5) $\pi(x)$ (the number of primes $\leq x$).[57] The following primitive recursion certifies the claim: $\pi(0) = 0$, and $\pi(x + 1) = $ if $Pr(x + 1)$ then $\pi(x) + 1$ else $\pi(x)$.

(6) $\lambda n.p_n$ (the nth prime). First note that the graph $y = p_n$ is primitive recursive: $y = p_n \equiv Pr(y) \wedge \pi(y) = n + 1$. Next note that, for all n, $p_n \leq 2^{2^n}$ (see Exercise 2.1.2.42 below), thus $p_n = (\mu y)_{\leq 2^{2^n}} (y = p_n)$, which settles the claim.

(7) $\lambda nx. \exp(n, x)$ (*the* exponent of p_n in the prime factorization of x). $\exp(n, x) = (\mu y)_{\leq x} \neg(p_n^{y+1}|x)$.

(8) $Seq(x)$ (x's prime number factorization contains at least one prime, but no gaps) $Seq(x) \equiv x > 1 \wedge (\forall y)_{\leq x}(\forall z)_{\leq x}(Pr(y) \wedge Pr(z) \wedge y < z \wedge z|x \rightarrow y|x)$.   □

2.1.2.41 Remark. What makes $\exp(n, x)$ "*the* exponent of p_n in the prime factorization of x", rather than *an* exponent, is Euclid's prime number factorization theorem: *Every number $x > 1$ has a unique factorization—within permutation of factors—as a product of primes.* See Exercise 40 of Section 1.8.   □

2.1.2.42 Exercise. Prove by induction on n, that for all n we have $p_n \leq 2^{2^n}$.

Hint. Consider, as Euclid did,[58] $p_0 p_1 \cdots p_n + 1$. If this number is prime, then it is greater than or equal to p_{n+1} (why?). If it is composite, then none of the primes up to p_n divide it. So any prime factor of it is greater than or equal to p_{n+1} (why?).   □

---

[56]The symbol "$\lfloor x \rfloor$" is called the *floor* of x. It succeeds in the literature (with the same definition) the so-called "greatest integer function, $[x]$", i.e., the *integer part* of the real number x.

[57]The π-function plays a central role in number theory, figuring in the so-called *prime number theorem*. See, for example, LeVeque (1956).

[58]In his proof that there are infinitely many primes.

2.1.2.43 Exercise. Prove that $\lambda x.\lfloor \log_2 x \rfloor \in \mathcal{PR}$. Remove the undefinedness at $x = 0$ in some convenient manner. For example, arrange that $\lfloor \log_2 0 \rfloor = 0$ □

2.1.2.44 Definition. (Coding Sequences) Any sequence of numbers, $a_0, \ldots, a_n$, $n \geq 0$, is *coded* by the number $[a_0, \ldots, a_n]$ defined as

$$\prod_{i \leq n} p_i^{a_i + 1}$$ □

For *coding* to be useful, we need a simple *decoding* scheme. By Remark 2.1.2.41 there is no way to have $z = [a_0, \ldots, a_n] = [b_0, \ldots, b_m]$, *unless*

  (i) $n = m$

     and

  (ii) For $i = 0, \ldots, n$, $a_i = b_i$.

Thus, it makes sense to correspondingly define the *decoding expressions*:

  (i) $lh(z)$ (pronounced "length of z") as shorthand for $(\mu y)_{\leq z} \neg (p_y | z)$

  (ii) $(z)_i$ as shorthand for $\exp(i, z) \dotdiv 1$

Note that

  (a) $\lambda i z.(z)_i$ and $\lambda z.lh(z)$ are in $\mathcal{PR}$.

  (b) If $Seq(z)$, then $z = [a_0, \ldots, a_n]$ for some $a_0, \ldots, a_n$. In this case, $lh(z)$ equals the number of distinct primes in the decomposition of z, that is, the length $n + 1$ of the coded sequence. Then $(z)_i$, for $i < lh(z)$, equals a_i. For larger i, $(z)_i = 0$. Note that if $\neg Seq(z)$ then $lh(z)$ need not equal the number of distinct primes in the decomposition of z. For example, 10 has 2 primes, but $lh(10) = 1$.

The symbol $[\ldots]$ for *numerical sequence coding* is not standard. Usually, $\langle \ldots \rangle$ is used in the literature on theory of computation. But this clashes with the use of the same symbol in set theory, used to denote ordered tuples of objects.

   The tools lh, $Seq(z)$, and $\lambda i z.(z)_i$ are sufficient to perform *decoding*, primitive recursively, once the truth of $Seq(z)$ is established. This coding/decoding scheme is essentially that of Gödel (1931), and we will use it throughout this volume.

   We conclude this subsection with a flexible extension of primitive recursion *on total functions*. Simple primitive recursion defines a function "at $n+1$" in terms of its value "at n". However we also have examples of "recursions" (or "recurrences"), one of the best known perhaps being the *Fibonacci sequence*, $0, 1, 1, 2, 3, 5, 8, \ldots$, that is given by $F_0 = 0$, $F_1 = 1$ and (for $n \geq 1$) $F_{n+1} = F_n + F_{n-1}$, where the value at $n+1$ depends on both the values at n and $n-1$. This generalizes to recursions where the value at $n + 1$ depends, in general, on the entire *history*, or *course-of-values*, of the function values at inputs $n, n-1, n-2, \ldots, 1, 0$. Cf. 1.7.0.29. The easiest way

to represent the history of values of a *total* number-theoretic function f "at (input) x", namely, $\{f(0,\vec{y}), f(1,\vec{y}), \ldots, f(x,\vec{y})\}$, is to code it by a single number!

2.1.2.45 Definition. (Course-of-Values Recursion) We say that f, of $n + 1$ arguments, is defined from two total functions—namely, the *basis* function $\lambda \vec{y}_n . b(\vec{y}_n)$ and the *iterator* $\lambda x \vec{y}_n z . g(x, \vec{y}_n, z)$—by *course-of-values recursion* if for all $x, \vec{y}_n$ the following equations hold:

$$\begin{cases} f(0, \vec{y}_n) & = b(\vec{y}_n) \\ f(x+1, \vec{y}_n) & = g(x, \vec{y}_n, H(x, \vec{y}_n)) \end{cases} \tag{1}$$

where $\lambda x \vec{y}_n . H(x, \vec{y}_n)$ is the *history function*, which is given "at x" (for all $\vec{y}_n$) by

$$[f(0,\vec{y}), f(1,\vec{y}), \ldots, f(x,\vec{y})] \qquad \square$$

Compare with the general case in 1.7.0.29. Here, totalness allows a neat coding of the "history" $\{f(0,\vec{y}), f(1,\vec{y}), \ldots, f(x,\vec{y})\}$ as a single number that depends on x and $\vec{y}$.[59] Of course, 1.7.0.29 guarantees the existence of an f satisfying the schema (1) above.

2.1.2.46 Exercise. Prove that f given by (1) is total.
   Hint. Use strong induction on x. $\qquad \square$

The major result here is:

2.1.2.47 Theorem. $\mathcal{PR}$ *is closed under course-of-values recursion.*

Proof. So, let b and g be in $\mathcal{PR}$. We will show that $f \in \mathcal{PR}$. It suffices to prove that the history function H is primitive recursive, for then $f = \lambda x \vec{y}_n . \big(H(x, \vec{y}_n)\big)_x$ and we are done by Grzegorczyk substitution. To this end, the following equations—true for all $x, \vec{y}_n$—settle the case:

$$H(0, \vec{y}_n) = [b(\vec{y}_n)]$$
$$H(x+1, \vec{y}_n) = H(x, \vec{y}_n) p_{x+1}^{g(x, \vec{y}_n, H(x, \vec{y}_n))+1} \qquad \square$$

The same proof with trivial adjustments yields:

2.1.2.48 Corollary. $\mathcal{R}$ *is closed under course-of-values recursion.*

2.1.2.49 Example. The Fibonacci sequence, $(F_n)_{n \geq 0}$, is given by

$$F_0 = 0$$
$$F_1 = 1$$

   otherwise,

---

[59] The inclusion of $f(i, \vec{y})$ in $\Pi_{i \leq x} p_i^{f(i,\vec{y})+1}$ renders all of $H(x, \vec{y})$ undefined if $f(i, \vec{y}) \uparrow$. In a set setting, the latter condition simply means that $f(i, \vec{y})$ is not included in the set $\{f(0,\vec{y}), f(1,\vec{y}), \ldots, f(x,\vec{y})\}$.

$$F_{n+1} = F_n + F_{n-1}$$

It can be viewed as a function $\lambda n.F_n$. As such it is in $\mathcal{PR}$. Indeed, letting H_n be the history of the sequence at n—that is, $[F_0, \ldots, F_n]$—we have the following course-of-values recursion for $\lambda n.F_n$ in terms of functions known to be in $\mathcal{PR}$.

$$F_0 = 0$$
$$F_{n+1} = \text{if } n = 0 \text{ then } 1$$
$$\text{else } \big(H_n\big)_n + \big(H_n\big)_{n \dot- 1} \qquad \qquad \square$$

2.1.2.50 Remark. How important is totalness in course-of-values recursion?

Let us analyze the primitive recursion schema below, where b and g are known to be in $\mathcal{P}$. What can we learn from the *partial recursive H* that is defined as follows? (Note the use of "$\simeq$" this time.)

$$H(0, \vec{y}_n) \simeq [b(\vec{y}_n)]$$
$$H(x + 1, \vec{y}_n) \simeq H(x, \vec{y}_n) p_{x+1}^{g(x, \vec{y}_n, H(x, \vec{y}_n)) + 1} \qquad (1)$$

First off, the domain of $\lambda x.H(x, \vec{y}_n)$ is either $\mathbb{N}$, or a set of the form $\{i \in \mathbb{N} : i < k\}$ for some k (that depends on the chosen fixed $\vec{y}_n$). We call this latter set an *initial segment* of $\mathbb{N}$.

Let us verify this claim. We fix a $\vec{y}_n$. If $\lambda x.H(x, \vec{y}_n)$ is total, then there is nothing to prove. Otherwise, let $x = x_0$ be *smallest* such that $H(x, \vec{y}_n) \uparrow$. By induction on x we see that $H(x, \vec{y}_n) \uparrow$, for $x \geq x_0$. Indeed, the basis is from the choice of x_0. On the obvious I.H. let us look at $H(x + 1, \vec{y}_n)$. By the I.H., the product in the right hand side of (1) is undefined.

Next, we note that "if $x < x_0$ then $H(x, \vec{y}_n) = [a_0, \ldots, a_x]$, for appropriate a_i". Of course, if $x_0 = 0$ then the statement is vacuously true. So assume $x_0 > 0$. We do induction on x. The basis is settled by $H(0, \vec{y}_n) = [b(\vec{y}_n)]$ (why "="?). If $H(x, \vec{y}_n) = [a_0, \ldots, a_x]$ and $x + 1 < x_0$ (I.H.), then the second equation of (1) yields $H(x + 1, \vec{y}_n) = [a_0, \ldots, a_x, g(x, \vec{y}_n, H(x, \vec{y}_n))]$, as needed.

Let us now set

$$f(x, \vec{y}_n) \overset{\text{Def}}{\simeq} \big(H(x, \vec{y}_n)\big)_x \qquad (2)$$

Thus we have the validity of the course-of-values recurrence, for all x and $\vec{y}_n$

$$f(0, \vec{y}_n) \simeq b(\vec{y}_n)$$
$$f(x + 1, \vec{y}_n) \simeq g(x, \vec{y}_n, H(x, \vec{y}_n)) \qquad (3)$$

By (2) and the earlier remarks, $\text{dom}(f) = \text{dom}(H)$ and in particular, for each fixed $\vec{y}_n$, $\text{dom}(\lambda x.f(x, \vec{y}_n))$ is either $\mathbb{N}$ or a set of the form $\{i \in \mathbb{N} : i < k\}$ for some k (that depends on the chosen fixed $\vec{y}_n$).

By definition, (1) is the course-of-values recursion schema for partial functions. It defines a function H, but also a function f, the latter satisfying (3). We can now summarize: $\qquad \qquad \square$

2.1.2.51 Theorem. $\mathcal{P}$ *is closed under the schema of course-of-values recursion, that is, (1) of 2.1.2.50. For any fixed $\vec{y}_n$, the subsidiary function f defined by (2) is either total in x or is an initial segment of $\mathbb{N}$. Moreover it satisfies the recursion (3).*

2.1.3   Simultaneous Primitive Recursion

Taking $B = \mathbb{N}^k$ in 1.7.0.27, and letting h and g be *total* functions from $\mathbb{N}^n$ and $\mathbb{N}^{n+1} \times \mathbb{N}^k$ respectively to $\mathbb{N}^k$, and replacing $\simeq$ by $=$, (1) in said example becomes

$$\begin{cases} f(0, \vec{y}_n) & = h(\vec{y}_n) \\ f(x+1, \vec{y}_n) & = g(x, \vec{y}_n, f(x, \vec{y}_n)) \end{cases} \tag{1}$$

Now f, h, and g are k-tuple valued, so they are *not* number-theoretic functions (which *must* have a right field equal to $\mathbb{N}$, by definition; cf. 2.1.1.6). However, if we write them in terms of their components, for example,

$$f(x, \vec{y}_n) = \langle f_1(x, \vec{y}_n), \dots, f_k(x, \vec{y}_n) \rangle$$

then we can rewrite the recurrence equations (1) componentwise, to obtain the schema of *simultaneous (primitive) recursion* below, which first occurred in Hilbert and Bernays (1968).

$$\begin{cases} f_1(0, \vec{y}_n) & = h_1(\vec{y}_n) \\ \vdots & \\ f_k(0, \vec{y}_n) & = h_k(\vec{y}_n) \\ f_1(x+1, \vec{y}_n) & = g_1(x, \vec{y}_n, f_1(x, \vec{y}_n), \dots, f_k(x, \vec{y}_n)) \\ \vdots & \\ f_k(x+1, \vec{y}_n) & = g_k(x, \vec{y}_n, f_1(x, \vec{y}_n), \dots, f_k(x, \vec{y}_n)) \end{cases} \tag{2}$$

Hilbert and Bernays proved the following:

2.1.3.1 Theorem. *If all the h_i and g_i are in $\mathcal{PR}$ (resp. $\mathcal{R}$), then so are all the f_i obtained by the schema (2) of simultaneous recursion.*

Proof. Define, for all $x, \vec{y}_n$, and $\vec{z}_k$,

$$F(x, \vec{y}_n) \stackrel{\text{Def}}{=} [f_1(x, \vec{y}_n), \dots, f_k(x, \vec{y}_n)]$$

$$H(\vec{y}_n) \stackrel{\text{Def}}{=} [h_1(\vec{y}_n), \dots, h_k(\vec{y}_n)]$$

$$G(x, \vec{y}_n, \vec{z}_k) \stackrel{\text{Def}}{=} [g_1(x, \vec{y}_n, \vec{z}_k), \dots, g_k(x, \vec{y}_n, \vec{z}_k)]$$

We readily have that $H \in \mathcal{PR}$ (resp. $\in \mathcal{R}$) and $G \in \mathcal{PR}$ (resp. $\in \mathcal{R}$)—and, in particular, $\lambda x \vec{y}_n z.G(x, \vec{y}_n, (z)_0, \ldots, (z)_{k-1}) \in \mathcal{PR}$ (resp. $\in \mathcal{R}$)—depending on where we assumed the h_i and g_i to be. We can now rewrite schema (2) as

$$\begin{cases} F(0, \vec{y}_n) & = H(\vec{y}_n) \\[2mm] F(x+1, \vec{y}_n) & = G\left(x, \vec{y}_n, \left(F(x, \vec{y}_n) \right)_0, \ldots, \left(F(x, \vec{y}_n) \right)_{k-1} \right) \end{cases} \quad (3)$$

By the above remarks, $F \in \mathcal{PR}$ (resp. $\in \mathcal{R}$) depending on where we assumed the h_i and g_i to be. In particular, this holds for each f_i since, for all $x, \vec{y}_n, f_i(x, \vec{y}_n) = \left(F(x, \vec{y}_n) \right)_{i-1}$. $\qquad \square$

2.1.3.2 Example. We saw one way to justify that $\lambda x.rem(x, 2) \in \mathcal{PR}$ in 2.1.2.40. A direct way is the following. Setting $f(x) = rem(x, 2)$, for all x, we notice that the sequence of outputs (for $x = 0, 1, 2, \ldots$) of f is

$$0, 1, 0, 1, 0, 1 \ldots$$

Thus, the following primitive recursion shows that $f \in \mathcal{PR}$:

$$\begin{cases} f(0) & = 0 \\ f(x+1) & = 1 \dot{-} f(x) \end{cases}$$

Here is a way, via simultaneous recursion, to obtain a proof that $f \in \mathcal{PR}$, without using any arithmetic! Notice the infinite "matrix"

$$\begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 & \ldots \\ 1 & 0 & 1 & 0 & 1 & 0 & \ldots \end{matrix}$$

Let us call g the function that has as its sequence outputs the entries of the second row—obtained by shifting the first row by one position to the left. The first row still represents our f. Now

$$\begin{cases} f(0) & = 0 \\ g(0) & = 1 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) \end{cases} \quad (1)$$

$\qquad \square$

The reader may protest that the above example contains a bit of an overstatement, if not sophistry: After all, to legitimize simultaneous recursion as a "$\mathcal{PR}$ operation" not only did we use arithmetic, but quite a bit at that. Didn't we?

Well yes, but that is irrelevant to the point we are driving at: A very simple programming language (formalism), which we will discuss in the next section, is intimately connected with simultaneous recursion, and we will see that, within that formalism, we indeed can compute $rem(x, 2)$ doing no arithmetic at all!

How we will do this is *precisely* reflected by the recursion (1) above.

2.1.3.3 Example. We saw one way to justify that $\lambda x. \lfloor x/2 \rfloor \in \mathcal{PR}$ in 2.1.2.40. A direct way is the following.

$$\begin{cases} \left\lfloor \dfrac{0}{2} \right\rfloor & = 0 \\[2ex] \left\lfloor \dfrac{x+1}{2} \right\rfloor & = \left\lfloor \dfrac{x}{2} \right\rfloor + rem(x, 2) \end{cases}$$

where rem is in $\mathcal{PR}$ by 2.1.2.40 or by 2.1.3.2.

Alternatively, here is a way that can do it—via simultaneous recursion—and with only the knowledge of how to add 1. Consider the matrix

$$\begin{array}{ccccccccc} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & \ldots \\ 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & \ldots \end{array}$$

The top row represents $\lambda x. \lfloor x/2 \rfloor$, let us call it "$f$". The bottom row we call g and is, again, the result of shifting row one to the left by one position. Thus, we have a simultaneous recursion

$$\begin{cases} f(0) & = 0 \\ g(0) & = 0 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) + 1 \end{cases} \tag{2}$$

$\square$

2.1.4   Pairing Functions

Coding of sequences $a_0, a_1, \ldots, a_n$, for $n \geq 1$, has a special case; pairing functions, that is, the case of $n = 2$.

2.1.4.1 Definition. A total, 1-1 function $J : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is called a *pairing function*.

$\square$

2.1.4.2 Remark. By 1.2.0.25 there is an onto $g : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ such that $g\big(J(x, y)\big) = \langle x, y \rangle$ for all x, y in $\mathbb{N}$. g is *pair-valued* ("vector"-valued) thus not a number-theoretic function. As in the previous subsection, we may write, for all $z \in \text{dom}(g)$,

$$g(z) = \langle K(z), L(z) \rangle$$

that is, decompose g as a pair of functions $\langle K, L \rangle$, along the "x-" and "y-axes". We call K and L the *first* and *second projection* functions of the pairing function f, respectively. We have at once, for all x and y in $\mathbb{N}$,

$$K\big(J(x, y)\big) = x \tag{1}$$

and

$$L\big(J(x, y)\big) = y \tag{2}$$

One usually encounters the (capital) letters K, L in the literature as (generic) names for projection functions of some (generic) pairing function. In turn, the generic symbol for the latter is J rather than "f". We will conform to this notational convention in what follows.

We must emphasize that since Definition 2.1.4.1 does not require a pairing function J to be onto $\mathbb{N}$, we *cannot* expect

$$J(K(z), L(z)) = z, \text{ for all } z \tag{3}$$

to hold *in general*. In fact, (3) *requires* that J is onto (cf. 1.2.0.22).

Pause. What if in (3) we said "for all $z \in \text{dom}(\langle K, L\rangle)$" instead?◄

Hint. See Section 1.8, Exercise 26.

Also note that if the g of this remark is obtained precisely as in 1.2.0.25, then its domain equals the range of J—cf. 1.2.0.26. However, we can *extend* g to a total function $\tilde{g}$, for example, letting, for all $z \in \mathbb{N}$,

$$\tilde{g}(z) = \begin{cases} g(z) & \text{if } z \in \text{dom}(g) \\ \langle 0, 0\rangle & \text{otherwise} \end{cases}$$

Clearly, we still get $(\tilde{g}J) = 1_{\mathbb{N}\times\mathbb{N}}$, that is, $\tilde{g}(J(x,y)) = \langle x, y\rangle$ for all x and y in $\mathbb{N}$, simply because $J(x,y) \in \text{dom}(g)$. Of course, by preceding remarks, $(J\tilde{g}) = 1_{\mathbb{N}}$ *must* fail unless J is onto $\mathbb{N}$.   □

The set of "tools" consisting of a pairing function J and its two projections K and L is a coding/decoding scheme for sequences of length two. We want to have computable such schemes and indeed there is an abundance of *primitive recursive* pairing functions that also have primitive recursive projections. Some of those we will indicate in the examples below and others we will let the reader to discover in the exercises section.

2.1.4.3 Example. The function $J = \lambda xy.[x,y]$ is pairing. Its projections are $K = \lambda z.(z)_0$ and $L = \lambda z.(z)_1$. All three are already known to us as members of $\mathcal{PR}$.

This J is not onto. For example, $5 \notin \text{ran}(J)$. Nevertheless, K and L are total—because $\lambda iz.(z)_i$ is; indeed is in $\mathcal{PR}$—and unlike the explicit extension of g in the preceding remark, this fact did not require our intervention.   □

Pause. Does the coincidence of the non ontoness of the J above, and the totalness of K and L, contradict our calculation in 1.2.0.26?◄

2.1.4.4 Example. The function $J = \lambda xy.2^x 3^y$ is pairing. Its projections are $K = \lambda z.\exp(0, z)$ and $L = \lambda z.\exp(1, z)$ (cf. 2.1.2.40). All three are already known to us as members of $\mathcal{PR}$.

This J is not onto. Again, $5 \notin \text{ran}(J)$. Nevertheless, K and L are total—because $\lambda iz.\exp(i, z)$ is; indeed is in $\mathcal{PR}$—and unlike the explicit extension of g in the preceding remark, this fact did not require our intervention either.   □

Clearly, for any distinct primes p_i and p_j, the function $J = \lambda xy.p_i^x p_j^y$ is pairing, with projections $\lambda z.\exp(i, z)$ and $\lambda z.\exp(j, z)$.

2.1.4.5 Example. Note that every number $n \geq 1$ has a unique representation—i.e., the x and y are uniquely determined by n—of the form $2^x(2y+1)$. This, for $n \geq 2$ is just an abstraction of the unique factorization theorem, recognizing that the $3^a 5^{a'} 7^{a''} \cdots$ part of the factorization is an odd number. On the other hand, $1 = 2^0(2 \cdot 0 + 1)$.

Thus, the $J = \lambda xy.2^x(2y+1)$ of Grzegorczyk (1953) is pairing. This J is not onto either; it just misses one member of $\mathbb{N}$!

Its projections can be easily calculated in a manner that readily establishes their primitive recursiveness. $Kz = \exp(0, z)$ and

$$
Lz = \left\lfloor \frac{\left\lfloor \frac{z}{2^{Kz}} \right\rfloor \dot{-} 1}{2} \right\rfloor
\qquad \square
$$

The preceding example illustrates a notational convention that we will adhere to regarding projection functions: We will write "Kz", "Lz", "$KLLz$", etc., omitting brackets from around arguments. This is a visual improvement over "$K(z)$", "$L(z)$", "$K(L(L(z)))$", etc.

2.1.4.6 Example. $J = \lambda xy.2^{x+y+2} + 2^{y+1}$ is pairing. That it is in $\mathcal{PR}$ and hence total is trivial. Why is it 1-1? Well, we may prove this directly, but instead, what if I can find x and y *uniquely* in terms of z once I set $z = 2^{x+y+2} + 2^{y+1}$?

If that succeeds then I have defined two functions $\lambda z.Kz$ and $\lambda z.Lz$ that satisfy (1) and (2) of 2.1.4.2. This will prove that J is 1-1 by 1.2.0.22!

Let us do this, and start by setting $z = J(x, y)$. We will "solve" for (natural number-values of) x and y:[60] Notice that $2^{y+1} < 2^{x+y+2}$,[61] hence, $2^{x+y+2} \leq z < 2^{x+y+2} + 2^{x+y+2} = 2^{x+y+3}$. Thus, taking logarithms base-2, $x + y + 2 \leq \log_2 z < x + y + 3$. We obtain at once

$$
x + y + 2 = \lfloor \log_2 z \rfloor \tag{1}
$$

Now, $z = 2^{\lfloor \log_2 z \rfloor} + 2^{y+1}$, thus

$$
y = \left\lfloor \log_2 \left(z \dot{-} 2^{\lfloor \log_2 z \rfloor} \right) \right\rfloor \dot{-} 1
$$

We obtain $Lz = \left\lfloor \log_2 \left(z \dot{-} 2^{\lfloor \log_2 z \rfloor} \right) \right\rfloor \dot{-} 1$ and, from (1), $Kz = \lfloor \log_2 z \rfloor \dot{-} (Lz + 2)$. From Exercise 2.1.2.43 we conclude that K and L are indeed in $\mathcal{PR}$. $\square$

2.1.4.7 Example. In 2.1.4.6 we note that $J(x, y) \geq x$ and $J(x, y) \geq y$, for all x, y. Thus an alternative way to prove that the related K and L are in $\mathcal{PR}$ is to compute as follows:

$$
Kz = (\mu x)_{\leq z}(\exists y)_{\leq z}(J(x, y) = z) \tag{1}
$$

---

[60] An equation for which we seek *integer solutions only* is called a *Diophantine equation*.
[61] For example, take the logarithm base-2 for both sides.

and

$$Lz = (\mu y)_{\leq z}(\exists x)_{\leq z}(J(x,y) = z) \tag{2}$$

Equipped with theorems 2.1.2.28 and 2.1.2.32, and Definition 2.1.2.39, we see that (1) and (2) establish that K and L are in $\mathcal{PR}$.

But this approach—unlike that in the preceding example—does *not* prove that K and L exist or that the J is 1-1!

Pause. Why?◄ □

2.1.4.8 Example. Here is a pairing function that does not require exponentiation. Let $J(x,y) = (x+y)^2 + x$. Clearly, $J \in \mathcal{PR}$.

Let us use the methodology of 2.1.4.6. So let us set $z = (x+y)^2 + x$ and solve this "equation" for x and y (uniquely, hopefully). Well, $(x+y)^2 \leq z < (x+y+1)^2$. Thus $x + y \leq \sqrt{z} < x + y + 1$, hence

$$x + y = \lfloor \sqrt{z} \rfloor \tag{1}$$

Then, $z = \lfloor \sqrt{z} \rfloor^2 + x$ and therefore $Kz = z \dot{-} \lfloor \sqrt{z} \rfloor^2$. By (1), $Lz = \lfloor \sqrt{z} \rfloor \dot{-} Kz$.

As in 2.1.4.7, the J here satisfies $J(x,y) \geq x$ and $J(x,y) \geq y$. Thus, if we have an independent proof of the *existence* of K and L (say, by proving directly that J is 1-1; cf. Exercise 4 in Section 2.12), then their primitive recursiveness follows from the calculations $Kz = (\mu x)_{\leq z}(\exists y)_{\leq z}(J(x,y) = z)$ and $Lz = (\mu y)_{\leq z}(\exists x)_{\leq z}(J(x,y) = z)$. □

Why bother about pairing functions when we have the coding of sequences scheme of the previous subsection? Because prime-power coding is computationally very inefficient, while quadratic schemes such as that of the previous example allow us to significantly reduce the "computational complexity" of coding/decoding. But can we code arbitrary length sequences efficiently?

Yes, because any J, K, L scheme can lead to a coding/decoding scheme for sequences $a_1, \ldots, a_n$, $n \geq 2$, for both the cases of a fixed or variable n.

2.1.4.9 Definition. Given a primitive recursive pairing scheme J, K, L.

For any fixed $n \geq 2$ we define by recursion on n the symbol $[\![a_1, \ldots, a_n]\!]^{(n)}$: $[\![x, y]\!]^{(2)} = J(x,y)$; and $[\![x, y_1, \ldots, y_n]\!]^{(n+1)} = J(x, [\![y_1, \ldots, y_n]\!]^{(n)})$. □

2.1.4.10 Remark. It is a trivial exercise to verify that the 1-1ness of J implies that of $[\![y_1, \ldots, y_n]\!]^{(n)}$, for any $n \geq 2$. Moreover, if J is onto $\mathbb{N}$, then so is $[\![y_1, \ldots, y_n]\!]^{(n)}$ (cf. Exercise 34 in Section 1.8).

The primitive recursive K and L give rise to primitive recursive projections, denoted by Π_i^n for each fixed n, and each $1 \leq i \leq n$.

These will satisfy $\Pi_i^n [\![y_1, \ldots, y_n]\!]^{(n)} = y_i$, for $1 \leq i \leq n$.

The Π are defined by composition from K and L as follows (cf. Exercise 8 in Section 2.12). if $z = [\![\, y_1, \ldots, y_n \,]\!]^{(n)}$, then

$$\Pi_1^n = K$$
$$\Pi_2^n = KL$$
$$\Pi_3^n = KLL$$
$$\vdots$$
$$\Pi_i^n = KL^{i-1}$$
$$\vdots$$
$$\Pi_{n-1}^n = KL^{n-2}$$
$$\Pi_n^n = L^{n-1}$$

where we wrote L^i for

$$\overbrace{(LL \cdots L)}^{i}$$

for any fixed i ($i - 1$ compositions with itself). $\square$

We can also effect coding of variable length sequences, starting with a J, K, L primitive recursive scheme. However, we will not take this approach in this volume. Here is how:

2.1.4.11 Definition. Given a primitive recursive pairing scheme J, K, L, with a J that is onto.

A sequence $a_1, \ldots, a_n$, $n \geq 0$—$n = 0$ denoting the empty sequence—is coded by the expression $[\![\, 0, 0 \,]\!]^{(2)}$ if $n = 0$; $[\![\, 1, a_0 \,]\!]^{(2)}$ if $n = 1$; and $[\![\, n, a_1, \ldots, a_n \,]\!]^{(n+1)}$ otherwise. We denote by $lg(z)$ the sequence length, thus we set $lg = K$. $\square$

Ontones of J, and hence of $[\![\, a_1, \ldots, a_n \,]\!]^{(n)}$, for any $n \geq 2$, ensures that every z is a valid "code", so it makes sense to decode it once we know the code's *length*; we do not need a predicate analogous to "Seq" (Example 2.1.2.40, item 8).

The projections, denoted by $(\!(\, z \,)\!)_i$, for non zero $lg(z)$ and $1 \leq i \leq lg(z)$, are

$$(\!(\, z \,)\!)_i = \begin{cases} KL^i z & \text{if } 1 \leq i < lg(z) \\ L^i z & \text{if } i = lg(z) \end{cases}$$

The reader encounters here the symbol "L^i", where i is variable (an input), for the first time. What does it mean, and is $\lambda iz.L^i(z)$ primitive recursive?

This is a special case of primitive recursion, known as *iteration* (composing a function with itself a "variable number of times"), a concept that we will take up in

the next subsection. In the present case we note that

$$L^0 z = z$$
$$L^{x+1} z = L(L^x z)$$

Brackets were inserted in "$LL^x z$" above to make the iterator (2.1.1.14) "$L(\ldots)$" stand out. The above primitive recursion shows that $\lambda iz.L^i z \in \mathcal{PR}$.

2.1.5 Iteration

A very natural special case of primitive recursion, that of composing a function with itself a number of times that is "read" as an input—a "variable number of times"—has been studied by Robinson (1947) in the context of number-theoretic functions. In general, if $g : A \to A$ is some function, its (pure) iteration is the function $\lambda xy.g^x(y) : \mathbb{N} \times A \to A$ given by the following definition:

2.1.5.1 Definition. (Pure Iteration) We say that $\lambda xy.f(x,y) : \mathbb{N} \times A \to A$—where A is some set—is defined from $\lambda x.g(x) : A \to A$ by *pure iteration* iff, for all $x \in \mathbb{N}$ and $y \in A$, we have

$$f(0,y) = y$$
$$f(x+1,y) \simeq g(f(x,y))$$

We write $g^x(y)$ for $f(x,y)$, since a trivial induction on k shows that, for any $k \in \mathbb{N}$, $f(k,y) \simeq \underbrace{(gg\cdots g(y))}_{k-1 \text{ compositions}}$, while we adhere to the convention that $g^0 = \mathbf{1}_A$. □

2.1.5.2 Remark. Of course, we replace $\simeq$ above by $=$ if g is total. Thus every iteration is a special case of primitive recursion in the style of 2.1.1.14. Indeed, for any $g \in \mathcal{PR}$, $\lambda xy.g^x(y) \in \mathcal{PR}$. □

2.1.5.3 Example. Let $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ be given by $g(x,y) = \langle y, x \rangle$. What is $g^z(0,1)$? It is $\langle rem(z,2), rem(z+1,2) \rangle$. Cf. Example 2.1.3.2. □

2.1.5.4 Example. Let $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ be given by $g(x,y) = \langle y, x+1 \rangle$. What is $g^z(0,0)$? Consider the diagram below that indicates the values of $g^0, g^1, g^2, g^3, \ldots$ starting at input $\langle 0,0 \rangle$ (depicted as $\begin{smallmatrix}0\\0\end{smallmatrix}$ below). The arrows, labeled g, indicate the input/output relation of g applied to the input at the left of the arrow in each case.

$$0 \xrightarrow{g} 0 \xrightarrow{g} 1 \xrightarrow{g} 1 \xrightarrow{g} 2 \xrightarrow{g} 2 \xrightarrow{g} 3 \xrightarrow{g} 3 \cdots$$
$$0 \xrightarrow{g} 1 \xrightarrow{g} 1 \xrightarrow{g} 2 \xrightarrow{g} 2 \xrightarrow{g} 3 \xrightarrow{g} 3 \xrightarrow{g} 4 \cdots$$

Thus, $g^z(0,0) = \left\langle \left\lfloor \frac{z}{2} \right\rfloor, \left\lfloor \frac{z+1}{2} \right\rfloor \right\rangle$. Cf. Example 2.1.3.3. □

2.1.5.5 Example. Let $g : A^n \to A^n$ be such that its first component is the "identity" function, in the sense, $p_1^n(g(\langle a, \ldots \rangle)) = a$ for all $\langle a, \ldots \rangle \in \mathrm{dom}(g)$ (cf. Exercise 1.8.23). In other words, if $g = \langle g_1, \ldots, g_n \rangle$ is the decomposition of g into its components, then $g_1(\vec{x}_n) = x_1$ for all $\vec{x}_n$ in its domain.

We will show that the first component of $\lambda \vec{x}_n . g^z(\vec{x}_n)$ is also the identity.

Indeed, we can show this by induction on z: For $z = 0$, $g^0(\vec{x}_n) = \langle \vec{x}_n \rangle$, hence $p_1^n(g^0(\vec{x}_n)) = x_1$, for all $\vec{x}_n$. For fixed z, assume (I.H.) that

$$p_1^n(g^z(\vec{x}_n)) = x_1, \text{ for all } \vec{x}_n \text{ on which the left hand side of = is defined} \qquad (*)$$

that is, $g^z(\vec{x}_n) = \langle x_1, \ldots \rangle$. Thus,

$$g^{z+1}(\vec{x}_n) = g(x_1, \ldots) = {}^{62} \langle x_1, g_2(\langle x_1, \ldots \rangle), \ldots, g_n(\langle x_1, \ldots \rangle) \rangle \qquad \square$$

2.1.5.6 Remark. The preceding result clearly holds in the special cases on the two extremes:

(1) The induction step above essentially proves that *if the first component of each of* $g : A^n \to A^n$ *and* $f : A^n \to A^n$ *is the identity, then this holds for* (fg) *and* (gf) *as well*. Indeed, all that we used about g^z in the I.S. above was $(*)$; we might as well think of $\lambda \vec{x}_n . g^z(\vec{x}_n)$ as the "f" here.

(2) With g as in 2.1.5.5, let us set $f = \lambda \vec{x}_n z . g^z(\vec{x}_n)$. Then $f(\vec{x}_n, z) = \langle x_1, \ldots \rangle$ whenever it is defined.

It is clear from the arguments presented here and in 2.1.5.5 that x_1, the "first" variable, is not particularly privileged with respect to the foregoing reasoning, and that the results hold no matter which component of g and f is the identity. The "general" case starts with $g = \langle g_1, \ldots, g_n \rangle$ [and $f = \langle f_1, \ldots, f_n \rangle$ in (1)], where $g_k(\vec{x}_n) = x_k$ [and $f_k(\vec{x}_n) = x_k$ in (1)], for all $\vec{x}_n$ in the respective domains. $\square$

The main result in this subsection is Robinson's theorem, that in the presence of pairing functions we have a converse of Remark 2.1.5.2: that iteration can simulate primitive recursion of number-theoretic functions.

2.1.5.7 Theorem. *If a class of total number-theoretic functions is closed under iteration and composition and includes a pairing scheme J, K, L, then it is closed under the full primitive recursion of 2.1.1.14 as well.*

Proof. We follow Tourlakis (1984). We are given the number-theoretic g (iterator) and h (basis). We will simulate the schema (A) below, using pure iteration.

$$(A) \quad \begin{cases} f(0, \vec{y}_n) & = h(\vec{y}_n) \\ f(x+1, \vec{y}_n) & = g(x, \vec{y}_n, f(x, \vec{y}_n)) \end{cases}$$

---

[62] Recall that we write $g(x, y, \ldots)$ for $g(\langle x, y, \ldots \rangle)$; cf. p. 43.

Using the coding introduced in Definition 2.1.4.9 that is generated by the given here J, K, L, we define the function F by

$$F(x, \vec{y}_n) = [\![\, x, \vec{y}_n, f(x, \vec{y}_n)\,]\!]^{(n+2)}$$

We may write a primitive recursion for F:

$$(B) \begin{cases} F(0, \vec{y}_n) & = [\![\, 0, \vec{y}_n, h(\vec{y}_n)\,]\!]^{(n+2)} \\ F(x+1, \vec{y}_n) & = [\![\, x+1, \vec{y}_n, g(x, \vec{y}_n, f(x, \vec{y}_n))\,]\!] \end{cases}$$

Noting that (omitting bracketing) $\Pi_1^{n+2} F(x, \vec{y}_n) = x$, $\Pi_{i+1}^{n+2} F(x, \vec{y}_n) = y_i$, for $1 \leq i \leq n$; and $\Pi_{n+2}^{n+2} F(x, \vec{y}_n) = f(x, \vec{y}_n)$, we see that schema (B) has the form

$$(B') \begin{cases} F(0, \vec{y}_n) & = H(\vec{y}_n) \\ F(x+1, \vec{y}_n) & = G(F(x, \vec{y}_n)) \end{cases}$$

where

$$G = \lambda x \vec{y}_n z. [\![\, \Pi_1^{n+2} z + 1, \Pi_{1+i}^{n+2} z, \dots, \Pi_{1+n}^{n+2} z,$$
$$g(\Pi_1^{n+2} z, \Pi_{1+i}^{n+2} z, \dots, \Pi_{1+n}^{n+2} z, \Pi_{n+2}^{n+2} z)\,]\!]$$

and $H = \lambda \vec{y}_n. [\![\, 0, \vec{y}_n, h(\vec{y}_n)\,]\!]^{(n+2)}$.

A trivial induction on x shows that $F(x, \vec{y}_n) = G^x H(\vec{y}_n)$. Thus, we have reduced the schema (A) to pure iteration G^x and composition and can obtain f as the $(n+2)$-th projection of F, as already indicated. $\qquad\Box$

2.2 A PROGRAMMING FORMALISM FOR THE PRIMITIVE RECURSIVE FUNCTIONS

Loop programs were introduced by Meyer and Ritchie (1967) as a program-theoretic counterpart to the number-theoretic introduction of the set of primitive recursive functions $\mathcal{PR}$. This programming formalism is analogous to the URM formalism, but, unlike the latter, it corresponds—i.e., is able to "compute"—*precisely the primitive recursive functions*. It also provides a connection between the *definitional (or structural) complexity* of primitive recursive functions—that is, the complexity of their syntactic definition—with their (run time) computational complexity as we show in Chapter 5.

Loop programs are very similar to programs written in the old programming language FORTRAN, but have a number of simplifications—notably, they lack an unrestricted do-while instruction (equivalently, they lack a **goto** instruction).

While we have an infinite supply of (program) variables, each loop program—being a finite string—references (uses) only a finite number of variables. We will most often denote variables *metamathematically* by single letter names (upper or lower case is all right) with or without subscripts or primes.[63]

---

[63]The precise syntax of variables will be given shortly, but even after this fact we will continue using signs such as X, A, Z', Y_{34}'' *to stand for variables*—i.e., we will continue using metanotation.

Let us define by induction (cf. 1.6.0.12), at first somewhat loosely, the structure (syntax) of loop programs. To assist the definition, and our general discussion about loop programs, we will be using syntactic variables (or metavariables) P, Q, R (with or without primes) to stand for loop programs.

2.2.0.8 Tentative Definition. (The Set L of Loop Programs) A loop program is one of

(1) A single *instruction*, that is, a string of type (i)–(iv) below

(i) $X \leftarrow 0$

(ii) $X \leftarrow Y$

(iii) $X \leftarrow X + 1$

(iv) **Loop** X; P; **end**

where P is a loop program and ";" is a separator, just like the semicolon used in the C programming language. *There is no restriction on whether X may or may not occur in P.*

In (i)–(iv), X and Y are metasymbols that denote arbitrary variables, including, possibly, *two identical* variables.

(2) The *superposition* or concatenation of a loop program, P, with an instruction, Q,[64] in that order: That is, $P; Q$.

The set of all loop programs we denote by L.   □

Instruction (iv) is substantial. However, this situation is common in programming language definition. For example, an Algol or Pascal "if-then-else" instruction has the form "**if** condition **then** *instruction1* **else** *instruction2*", where each of *instruction1* and *instruction2* may be a so-called "begin-end block", which is a self-standing program.

By clause (2) above, a loop program is an *ordered sequence of instructions*, separated by semicolons. In informal discussions we *normally write a loop program vertically*, in which case the separator ";" becomes redundant and is omitted. Thus, rather than

$$P; Q$$

we write

$$P$$
$$Q$$

Rather than

Loop X; P; **end**

---

[64]Using the letter Q to stand for an instruction is consistent with clause (1).

we write

$$\textbf{Loop } X$$
$$P$$
$$\textbf{end}$$

The the syntactic construct **Loop** X; P; **end** is called a *loop closure* of P.

We will shortly offer a careful definition of loop program *semantics*, that is, exactly what loop programs "compute". It is instructive to do so informally at first. Thus, to begin with, as in the case of the URM formalism, the variables X, Y'', Z_{23}, etc., that we utilize in a loop program can hold any natural number. We assume that an appropriate "computing agent" (which may be human) understands and performs, or *executes*, a loop program's instructions.

In the course of this sequential instruction-execution the agent successively points to the *next instruction* that must be executed. Once the agent has completed executing the instruction, it will then point to the one immediately following it. It will do so for one instruction at a time, from the first toward the last.

Pause. But will the agent always reach the last instruction?◄

We will answer this affirmatively, shortly.
If there is no instruction to point to, then the agent *terminates* its computation.

2.2.0.9 Tentative Definition. (Informal Loop Program Semantics) The semantics, or prescribed behavior of the computing agent is as follows:

(1) The effect of executing any one of instructions (i)–(iv) by the agent is to

   (i) make the value of X equal to 0; and then point just below the instruction (where the next instruction is located, unless the executed instruction was the last one);

   (ii) copy the value of Y into X non destructively—i.e., the value of Y does not change—and then point just below the instruction;

   (iii) increment the current value of X by 1, and then point just below the instruction;

   (iv) according as X holds the value 0 or $k > 0$ *immediately prior* to the execution of the instruction: **Case where X holds 0**. *Do nothing* toward the instruction; then point just below the instruction. **Case where X holds $k > 0$**. *Macro expand* instruction (iv) as below, $(*)$, and execute the indicated sequence of instructions; then point just below the instruction.

Note that by recursion on the (tentative) definition of programs, the agent knows *how to execute in sequence all the instructions of (each copy of) P:*

$$
k \text{ copies } \left\{ \begin{array}{l} P \\ P \\ \vdots \\ P \end{array} \right. \tag{$*$}
$$

The semantics of (iv) require that the value of X immediately *before* the execution of instruction (iv) matters in determining the macro expansion. Any instructions of the type (i)–(iii) included in P may change X, but *not* the k in $(*)$.

(2) The effect of executing the program

$$
\begin{array}{c} P \\ Q \end{array}
$$

where P is a program, and Q is an instruction is to let the agent start by pointing at the first instruction of P and allowing it to begin execution. By recursion on the (tentative) definition of programs, *the agent knows* how to execute in sequence all the instructions of P.

If the last instruction of P eventually gets executed, then the agent points to instruction Q, which we execute according to part (1) of this definition.   □

As already noted, in a program P all instructions are *executed in sequence*, starting with the first instruction. The program *terminates* iff its *last instruction* has eventually been executed. The hedging in the preceding definition, "*If the last instruction of P eventually gets executed*", will now be removed:

2.2.0.10 Theorem. Every loop program P terminates.

Proof. By induction on the set L, as this is defined recursively in 2.2.0.8. We will also use 2.2.0.9.

First off, the basis looks into the case of programs consisting of a single instruction of types (i)–(iii). By 2.2.0.9 such a one-line program terminates, since the execution of the instruction can be clearly completed.

Assuming as I.H. that the claim holds in the case of a program P we prove that the execution of an instruction of type (iv)—**Loop** X; P; **end**—terminates. Indeed, this is trivial if X holds 0 initially. If, on the other hand, it holds $k > 0$, then the instruction that the agent actually executes from top to bottom is the macro expansion

$$
k \text{ copies } \left\{ \begin{array}{l} P \\ P \\ \vdots \\ P \end{array} \right.
$$

By the I.H., the agent reaches and concludes the last instruction in each copy of P, so it eventually does so with the k-th copy.

The last case to consider is when we have a program

$$R = \begin{cases} P \\ Q \end{cases}$$

where Q is an instruction. Starting the agent at the first instruction of R is the same as starting it at the first instruction of P. By the I.H. the agent will eventually point to Q. But as we saw, it will be able to conclude the execution of Q under each case (i)–(iv) of 2.2.0.8. $\square$

2.2.0.11 Example. What does

Loop X
$\qquad X \leftarrow X + 1$
end

do? For any initial value of X, a, the above is computed as

$$a \text{ copies } \begin{cases} X \leftarrow X + 1 \\ X \leftarrow X + 1 \\ \vdots \\ X \leftarrow X + 1 \end{cases}$$

Thus, the end-value of X is $2a$. This is correct for both $a = 0$ and $a > 0$; cf. 2.2.0.9. $\square$

2.2.0.12 Example. What does

Loop Y
$\qquad X \leftarrow X + 1$
end

do? Assuming that X and Y initially hold a and b, respectively, the final value of X is $a + b$, while the final value of Y is still b. $\square$

2.2.0.13 Example. What does

Loop Z
$\qquad W \leftarrow X$
$\qquad X \leftarrow Y$
$\qquad Y \leftarrow W$

<div align="center">**end**</div>

do if X, Y, and W are initially 0, 1 and 1 respectively (cf. 2.1.5.3)? Given that the three instructions inside the loop swap the X and Y values, then, if the original value of Z is k, the end-values of X, Y, Z, W are, in order, $rem(k, 2), rem(k + 1, 2), k, rem(k + 1, 2)$.

We could have programmed this also reusing Z in the place of W, not introducing a fourth variable. Then the end-values of X, Y, Z would be, in order, $rem(k, 2), rem(k + 1, 2)$, if $k = 0$ then 0 else $rem(k + 1, 2)$.   □

2.2.0.14 Example. What does

<div align="center">

Loop Z

$W \leftarrow X$

$X \leftarrow Y$

$Y \leftarrow W$

$Y \leftarrow Y + 1$

end

</div>

do if X, Y are both 0, initially 0 (cf. 2.1.5.4)? Given that the four instructions inside the loop swap the given X and Y values, but make Y hold a value that is equal to $X + 1$, the effect is

$$X \longrightarrow \quad Y$$

$$Y \longrightarrow \quad X + 1$$

Thus, if the original value of Z is k, the end-values of X, Y are, in order, $\lfloor \frac{k}{2} \rfloor$, $\lfloor \frac{k+1}{2} \rfloor$.

We could have programmed this also reusing Z in the place of W, not introducing a fourth variable.   □

2.2.0.15 Example. What does the following do if $Z = a$ originally?

<div align="center">

$X \leftarrow 0$

$Y \leftarrow 0$

Loop Z

$Y \leftarrow X$

$X \leftarrow X + 1$

end

</div>

Notice the result of each "pass" around the loop:

$$X \xrightarrow{\#0} 0 \xrightarrow{\#1} 1 \xrightarrow{\#2} 2 \xrightarrow{\#3} 3 \xrightarrow{\#4} 4 \xrightarrow{\#5} 5 \cdots$$

$$Y \xrightarrow{\#0} 0 \xrightarrow{\#1} 0 \xrightarrow{\#2} 1 \xrightarrow{\#3} 2 \xrightarrow{\#4} 3 \xrightarrow{\#5} 4 \cdots$$

Thus, the end-value of Y is $a \doteq 1$, while that of X is a. This is correct also in the case where $a = 0$ initially. $\square$

We now present a more careful inductive definition of the syntax of loop programs. This, in turn, will allow a more mathematical definition of semantics that will eventually enable us to prove that loop programs compute precisely the functions of $\mathcal{PR}$. We start with the finite alphabet of symbols:

$$\Sigma = \{v, 1, \leftarrow, +, 0, \textbf{Loop}, \textbf{end}, ; \} \tag{1}$$

We next define the exact strings that are the variables:

2.2.0.16 Definition. (Variables of Loop Programs) The set of all variables used in loop programs is the closure of the single-element set[65] $\{v1\}$ under the string operation that, given a string x, produces $x1$ (x concatenated with 1). $\square$

Thus, the variables form the set $\{v1, v11, v111, \ldots, v1^n, \ldots\}$, where 1^n, for $n > 0$, denotes the string that consists of n 1-symbols. By convention, for any string x, x^0 denotes ϵ, the empty string.

We will often use a variation on the notation $v1^n$, namely, v_i.

Indeed, even more often, we will use, as we have already done in the preceding examples, metasymbols such as $x, y, X, Y, Z''_{33}, Y'''$ to stand for variables. Clearly, the v_i are also metasymbols, but are closer to the "ontology" of loop program variables. We can now see why our earlier statement "*X and Y are metasymbols that denote arbitrary variables, including, possibly,* two identical *variables*" (p. 126) makes sense: For example, both X and Y might actually stand for $v11$.

We can now state the final form of Definition 2.2.0.8:

2.2.0.17 Definition. The set of loop programs, L, is the closure of the set of strings over Σ [(1), p. 131] below

$$\{v_i \leftarrow 0 : i \geq 1\} \cup \{v_i \leftarrow v_j : i \geq 1 \wedge j \geq 1\} \cup \{v_i \leftarrow v_i + 1 : i \geq 1\}$$

under the following string-operations:

(1) From any string P we may form the string

$$\textbf{Loop } v_i; P; \textbf{end}$$

for any $i \geq 1$.

(2) From any strings P and Q we may form the following strings,[66] for all $i \geq 1$ and $j \geq 1$:

---

[65] One calls single-element sets *singletons*.

[66] Of course, P and Q may—but don't have to—name the same string (loop program). The same holds for v_i and v_j: It *is* allowed to have $i = j$.

(a) $P; v_i \leftarrow 0$

(b) $P; v_i \leftarrow v_j$

(c) $P; v_i \leftarrow v_i + 1$

(d) $P; \textbf{Loop } v_j; Q; \textbf{end}$ □

The definitive semantics of loop programs describe *what the execution of such a program does to the values (or "contents") of its list—or* tuple *under some fixed ordering—of variables.* Cf. also 2.2.0.9

2.2.0.18 Definition. (Loop Program Semantics) Let $x_1, \ldots, x_n$ be a fixed ordering of variables that *includes all the variables that occur in a program* P. Each x_i, of course, stands for a $v1^k$, for some $k > 0$. We denote by $\lambda \vec{x}_n.p(\vec{x}_n)$ the *vector-valued* function from $\mathbb{N}^n \to \mathbb{N}^n$ computed by P. This function *is* the semantics of P, and is defined by recursion on the definition of P (cf. Section 1.7) using 2.2.0.17 as follows:

For all (values of) $x_1, \ldots, x_n$ *that are in effect immediately prior to executing* P:
If P is

(I) $x_i \leftarrow 0$:   Then $p(\vec{x}_n) = \langle x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n \rangle$.

(II) $x_i \leftarrow x_j$:   Then $p(\vec{x}_n) = \langle x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_n \rangle$.

(III) $x_i \leftarrow x_i + 1$:   Then $p(\vec{x}_n) = \langle x_1, \ldots, x_{i-1}, x_i + 1, x_{i+1}, \ldots, x_n \rangle$.

(IV) **Loop** $Z; Q; \textbf{end}$, where Z is, without loss of generality, the variable x_1 (our ordering of the variables was arbitrary):   Then $p(\vec{x}_n) = q^{x_1}(\vec{x}_n)$.

(V) $R; S$, where S is a program of the types encountered in (I)–(IV) above: Then

$$p(x_1, \ldots, x_n) = s\Big(r(x_1, \ldots, x_n)\Big)$$ □

We notice that in the case (IV) above we followed precisely the informal semantics of 2.2.0.9. The case establishes that if $Z = a$ then the effect of **Loop** $Z; Q; \textbf{end}$ is the same as that of the program

$$\underbrace{Q; Q; \cdots; Q}_{a \text{ copies of } Q}$$

namely,

$$\underbrace{qq \cdots q}_{a \text{ copies of } q}(\vec{x}_n), \text{ where the } \vec{x}_n \text{ hold the values in effect } before \text{ the loop-start.}$$

The number of iterations, a, is predetermined *before we enter the loop* and regardless of what may be happening to Z (i.e., x_1) inside Q. That is, if Z is changed by Q this does not affect the number of copies of Q in the above macro expansion. This number depends only on the value that Z held *just prior to entering the loop*. In

particular, if $Z = 0$ the loop is in effect *skipped*, a fact captured by $q^0(\vec{x}_n) = \langle \vec{x}_n \rangle$, for all $\vec{x}_n$ (identity function from $\mathbb{N}^n \to \mathbb{N}^n$).

2.2.0.19 Example. Intuitively, a variable x_k that does *not occur* in a loop program P will not have its value changed by the execution of P. This expectation is upheld by our loop program semantics. That is, the semantics $\lambda \vec{x}_n.p(\vec{x}_n)$ of P—where $x_1, \ldots, x_n$ includes all the variables that occur in P—will, under our assumption, satisfy

$$p(\vec{x}_n) = \langle p_1(\vec{x}_n), \ldots, p_{k-1}(\vec{x}_n), x_k, p_{k+1}(\vec{x}_n) \ldots, p_n(\vec{x}_n) \rangle \qquad (1)$$

for all $\vec{x}_n$. We can prove (1) by induction on loop programs, with the help of Definition 2.2.0.18.

If P has the forms (I)–(III) (*basis*), then k is neither i nor j, therefore x_k stays unchanged.

If P has the form (IV), then as x_k does not occur in it, it is neither Z, nor occurs in Q. By the I.H. we have that the semantics of Q, i.e., q, satisfies for all $\vec{x}_n$,

$$q(\vec{x}_n) = \langle q_1(\vec{x}_n), \ldots, q_{k-1}(\vec{x}_n), x_k, q_{k+1}(\vec{x}_n) \ldots, q_n(\vec{x}_n) \rangle$$

By Remark 2.1.5.6, if we set $f = \lambda \vec{x}_n.q^{x_1}(\vec{x}_n)$, where, without loss of generality, Z is x_1, we will have $f(\vec{x}_n) = \langle \ldots, x_k, \ldots \rangle$, for all $\vec{x}_n$.

Finally, if P is $R; S$, then the I.H. applies to the semantics s and q, that is,

$$r(\vec{x}_n) = \langle r_1(\vec{x}_n), \ldots, r_{k-1}(\vec{x}_n), x_k, r_{k+1}(\vec{x}_n) \ldots, r_n(\vec{x}_n) \rangle$$

and

$$s(\vec{x}_n) = \langle s_1(\vec{x}_n), \ldots, s_{k-1}(\vec{x}_n), x_k, s_{k+1}(\vec{x}_n) \ldots, s_n(\vec{x}_n) \rangle$$

for all $\vec{x}_n$. Once more, invoking 2.1.5.6, we see that $s(r(\vec{x}_n)) = \langle \ldots, x_k, \ldots \rangle$, for all $\vec{x}_n$. □

2.2.0.20 Remark. The preceding example removes any ambiguity in the semantics that may be implied by our ability to arbitrarily add (in the definition) variables that may not occur in a program. The values of such variables remain invariant in the semantics. Or as we say, "during the loop program computation".[67] □

Let us turn once more to case (IV) of 2.2.0.18. We aim to write the iteration that defines p as a primitive recursion, and to this end we consider two cases according as x_1 occurs, or does not occur, in Q. Thus, for all $\vec{x}_n$, in the first case we have

$$q(\vec{x}_n) = \langle q_1(\vec{x}_n), q_2(\vec{x}_n) \ldots, q_n(\vec{x}_n) \rangle \qquad (1)$$

while, by 2.2.0.19, in the second case we have the decomposition

$$q(\vec{x}_n) = \langle x_1, q_2(\vec{x}_n) \ldots, q_n(\vec{x}_n) \rangle \qquad (2)$$

---

[67] We will define URM *computations* explicitly in Section 2.3. In the case of loop programs we have defined *semantics* that will characterize the set of computed functions without the need to define mathematically the concept of "computation" explicitly.

The primitive recursion for $F = \lambda a \vec{y}_n . q^a(\vec{y}_n)$ is given below (cf. 2.1.5.1):

$$\begin{cases} F(0, y_1, \ldots, y_n) & = \langle y_1, \ldots, y_n \rangle \\ F(a+1, y_1, \ldots, y_n) & = q(F(a, y_1, \ldots, y_n)) \end{cases} \qquad (3)$$

One invokes ("calls") F with the argument $x_1, x_1, x_2 \ldots, x_n$ to obtain the semantics p of (IV) above. That is, $p = \lambda \vec{x}_n . F(x_1, \vec{x}_n)$.

Thus, if we represent F in terms of its components, for all $a, \vec{y}_n$,

$$F(a, \vec{y}_n) = \langle F_1(a, \vec{y}_n), F_2(a, \vec{y}_n) \ldots, F_n(a, \vec{y}_n) \rangle$$

the recurrence (3) yields the following simultaneous recursion [in the style of (2) on p. 116)], if representation (1) holds for q,

$$\begin{aligned} F_i(0, \vec{y}_n) &= y_i \\ F_i(a+1, \vec{y}_n) &= q_i \Big(F_1(a, \vec{y}_n), \ldots, F_n(a, \vec{y}_n) \Big) \end{aligned} \qquad (3')$$

while it gives the following, if representation (2) holds.

$$\begin{aligned} F_i(0, \vec{y}_n) &= y_i \\ F_1(a+1, \vec{y}_n) &= F_1(a, \vec{y}_n) \\ &\text{and, for } i = 2, \ldots, n, \\ F_i(a+1, \vec{y}_n) &= q_i \Big(F_1(a, \vec{y}_n), \ldots, F_n(a, \vec{y}_n) \Big) \end{aligned} \qquad (3'')$$

Incidentally, a trivial induction on a, using the recurrence for F_1 in $(3'')$, rediscovers the result of 2.1.5.6: $F_1(a, \vec{y}_n) = y_1$ for all $a, \vec{y}_n$. In particular, this confirms our intuitive expectation that the variable Z of P remains unchanged by the execution of "instruction" (IV), if it does not occur in Q.

Thus, if all the component-functions, q_i, of q—the $\lambda \vec{x}_n . q_i(\vec{x}_n)$ above—are in $\mathcal{PR}$, then so are all the F_i. It follows at once by identification of variables (cf. 2.1.2.6) that all the p_i—the components of the semantics of P in (IV),

$$p_i = \lambda \vec{x}_n . F(x_1, \vec{x}_n)$$

are in $\mathcal{PR}$ as well.

Similar comments apply to the case (V) above: With the same notation as before, it is immediate that expressing the semantics p of $P = R; S$ component-wise

$$\langle \lambda \vec{x}_n . p_1(\vec{x}_n), \ldots, \lambda \vec{x}_n . p_n(\vec{x}_n) \rangle$$

we have the component-wise identities, for $i = 1, \ldots, n$:

$$p_i = \lambda \vec{x}_n . s_i \Big(r_1(\vec{x}_n), \ldots, r_n(\vec{x}_n) \Big)$$

Thus, if all the r_i and all the s_k are in $\mathcal{PR}$, then so are all the p_i functions.

We have done all the work that allows us now to state

2.2.0.21 Theorem. *For any loop program P whose variables are among $\vec{x}_n$, each of the components of the* semantics *function p—$\lambda \vec{x}_n.p_i(\vec{v}_n)$, for $i = 1, \ldots, n$—is in* $\mathcal{PR}$.

Proof. Induction on programs P. If P corresponds to (I)–(III) in Definition 2.2.0.18, then each of $\lambda \vec{x}_n.p_i(\vec{x}_n)$ is initial or is obtained from an initial function by adding "don't care" variables (cf. 2.1.2.6).

For case (IV), we are done by the I.H.—which applies to Q—and the preceding analysis of the semantics of the loop-instruction.

For case (V):

- If S is of type (I)–(III), then we are done by the basis case, and the remarks about (V) made before the theorem statement. We also invoke the I.H. that applies to R.

- If S is of type (IV), then we rely on the remarks made prior to the theorem about (IV): Accordingly, $s_i \in \mathcal{PR}$ by the I.H. that applies to Q, while the I.H. that applies to R concludes the case by remarks regarding (V). □

2.2.1 $\mathcal{PR}$ vs. $\mathcal{L}$

We next define what it means for a program P—whose list of variables *is* (rather than "includes") $x_1, \ldots, x_n$—to compute a number-theoretic function. This will be in exact analogy with the corresponding definition for URM programs (cf. 2.1.1.1). Of course, here we have no undefined/no-termination cases, since loop programs compute only total functions, indeed, exclusively primitive recursive functions as we just saw.

As in the case with the URM, we first decide which ones among the $\vec{x}_n$ we want to be the *input variables*; say,[68]

$$x_1, \ldots, x_r, \text{where } r \leq n \tag{1}$$

We also decide on *one output variable*; say, x_k.

The "agent" that executes a loop program (human, or machine) will first *implicitly—i.e., **not** via instructions that are contained in the program—initialize* the computation as follows:

(a) Sets the variables in the list (1) to hold the inputs $a_1, \ldots, a_r$.

(b) Sets the variables $x_{r+1}, \ldots, x_n$ to hold 0.

The agent will then execute program P according to the semantics of 2.2.0.18.

---

[68]More generally, we could have chosen $x_{i_1}, \ldots, x_{i_r}$ for input. Since renaming of variables is up to us we can avoid the ugly notational acrobatics that this choice entails.

The function computed by P with the given input-output choice is (with notation as in 2.2.0.18)

$$\lambda \vec{x}_n . U_k^n \Big(p(\vec{x}_r, \underbrace{0, 0, \dots, 0}_{n-r \text{ zeros}}) \Big)$$

where $U_k^n(p(\vec{y}))$ is short for the composition (2.1.1.13) $U_k^n(p_1(\vec{y}), \dots, p_n(\vec{y}))$ that is written in terms of the components p_j of p.

Intuitively, after "termination" of the the execution of P we read off what x_k holds—this is the output caused by input $\vec{x}_r$. We can thus record:

2.2.1.1 Definition. For a loop program program P whose variable list is $\vec{x}_n$—these variables are precisely those that occur in P—we define the symbol $P_{x_k}^{\vec{x}_r}$, where $1 \leq r, k \leq n$, to mean the computed by P function

$$\lambda \vec{v}_r . U_k^n \Big(p(\vec{x}_r, \underbrace{0, \dots, 0}_{n-r \text{ zeros}}) \Big)$$

The set of all loop program computable functions $P_{x_k}^{\vec{x}_r}$ we denote by $\mathscr{L}$. $\square$

2.2.1.2 Remark. Note that we have a totally syntactic definition of $\mathscr{L}$:

$$\mathscr{L} = \{P_{x_k}^{\vec{x}_r} : P \in L \wedge \text{ the } \vec{x}_r \text{ and } x_k \text{ occur in } P\}$$

$\square$

By 2.2.0.21, we have at once

2.2.1.3 Theorem. $\mathscr{L} \subseteq \mathcal{PR}$.

The converse is true.

2.2.1.4 Theorem. $\mathcal{PR} \subseteq \mathscr{L}$.

Proof. By induction on $\mathcal{PR}$ and brute-force programming:

Basis: $\lambda x.x + 1$ is P_X^X where P is $X \leftarrow X + 1$. Similarly, $\lambda \vec{x}_n.x_i$ is $P_{X_i}^{\vec{X}_n}$ where P is

$$X_1 \leftarrow X_1; X_2 \leftarrow X_2; \dots; X_n \leftarrow X_n$$

The case of $\lambda x.0$ is as easy.

How does one compute $\lambda x.f(g(x))$ if g is G_X^X and f is F_X^X? One uses

$$\begin{pmatrix} G' \\ F \end{pmatrix}_X^X$$

where G' is G, but modified to avoid side-effects: One must ensure that all the variables of G other than X are set to 0 upon exit from G, because F expects all

these variables to be 0 in order to compute f correctly. G' does that by placing at the end of G several statements of the type $Y \leftarrow 0$.

The general case $\lambda \vec{x}.f\big(g_1(\vec{x}), \ldots, g_n(\vec{x})\big)$ is programmed similarly.

Finally, we indicate in "pseudo code", as we say in programming courses, how to compute $f(x, \vec{y}_n)$ where

$$f(0, \vec{y}_n) = h(\vec{y}_n)$$
$$f(x + 1, \vec{y}_n) = g(x, \vec{y}_n, f(x, \vec{y}_n))$$

assuming we have loop programs H and G for h and g, rspectively. The pseudo code is

$$z \leftarrow h(\vec{y}_n)$$
$$i \leftarrow 0$$

   Loop x
   $$z \leftarrow g(i, \vec{y}_n, z)$$
   $$i \leftarrow i + 1$$
   end

Let then $h = H_z^{\vec{y}_n}$ and $g = G_z^{i, \vec{y}_n, z}$, where we have been careful to ensure that H and G do not have side-effects that affect adversely the semantics of the loop. That is,

(1) $\vec{y}_n$ are not changed by either H or G—they are "read-only".

(2) Neither H nor G contain the variable x.

(3) i is read-only in G and does not occur in H.

(4) H and G have no variables in common, other than $z, \vec{y}_n$.

   Let G additionally contain the variables $Z_1, \ldots, Z_m$ and H contain, additionally, $W_1, \ldots, W_r$.

(5) Program G explicitly sets the Z_j to 0 via $Z_j \leftarrow 0$ instructions—first thing, up in front—rather than wait for the "agent" to do so (cf. 2.2.1.1).

Thus, the pseudo code above transforms into

$$P \begin{cases} H \\ i \quad \leftarrow 0 \end{cases}$$

$$Q \begin{cases} \textbf{Loop } x \\ \quad G' \begin{cases} G \\ i \leftarrow i + 1 \end{cases} \\ \textbf{end} \end{cases}$$

The program $P; Q$ computes f. $\qquad \qquad \square$

2.2.1.5 Corollary. $\mathscr{L} = \mathcal{PR}$.

2.2.1.6 Example. It is instructive to follow Definition 2.2.0.18 to actually prove that $P; Q$ above indeed computes f of 2.2.1.4 in the sense of 2.2.1.1. Let us (arbitrarily) *order* all the variables of $P; Q$ as

$$z, x, i, \vec{y}_n, \vec{Z}_m, \vec{W}_r$$

Given our assumptions (1)–(5) above, about the variables, and about what H and G compute (cf. 2.2.1.1), the semantics (vector-valued function; cf. 2.2.0.18) of H, *diagrammatically* is

$$\left\langle 0, x, i, \vec{y}_n, \vec{Z}_m, \vec{0}_r \right\rangle \xrightarrow{\text{program } H} \left\langle h(\vec{y}_n), x, i, \vec{y}_n, \vec{Z}_m, \text{don't care} \right\rangle$$
$$\xrightarrow{\text{program } i \leftarrow 0} \left\langle h(\vec{y}_n), x, 0, \vec{y}_n, \vec{Z}_m, \text{don't care} \right\rangle \tag{1}$$

where "don't care" indicates our indifference to what happens to the $\vec{W}_r$ since they do not occur in G, and therefore the computation of G and G' is independent of the W_i-values. The sequential action from left to right indicated by the above arrows is the semantics of P.

Now, the semantics of $G' = \big(G; i \leftarrow i + 1\big)$ is

$$\left\langle z, x, i, \vec{y}_n, \vec{Z}_m, \vec{W}_r \right\rangle \xrightarrow{\text{program } G} \left\langle g(i, \vec{y}_n, z), x, i, \vec{y}_n, \cdots, \vec{W}_r \right\rangle$$
$$\xrightarrow{\text{program } i \leftarrow i + 1} \left\langle g(i, \vec{y}_n, z), x, i + 1, \vec{y}_n, \cdots, \vec{W}_r \right\rangle \tag{2}$$

Recall that G sets the Z_i to 0 explicitly, first thing, up in front! The "$\cdots$" just after $\vec{y}_n$ indicate what happens to the $\vec{Z}_m$ as a side-effect of executing program G (about which variables we do not need to be specific—nor can we possibly know anything about, in this general setting).

The $\vec{W}_r$, as well as the $x, i, \vec{y}_n$ remain unchanged under our assumptions.   (3)

The result in the z-coordinate is justified by 2.2.1.1 and the assumption that G "programs" g.

By 2.2.0.18, the semantics of Q is $q = (g')^x(z, x, i, \vec{y}_n, \vec{Z}_m, \vec{W}_r)$ and, therefore, that of $(P; Q)$ is [refer to (1) above]

$$(g')^x(h(\vec{y}_n), x, 0, \vec{y}_n, \vec{Z}_m, \text{don't care}) \tag{4}$$

The values in the W_i are "don't care" with respect to iterating g' since this function does not change the W_i by earlier remarks.

We now proceed to make some sense out of (4) toward our final goal, which is to verify that

$$f = \begin{pmatrix} P \\ Q \end{pmatrix}_z^{x, \vec{y}_n} \tag{5}$$

The diagram uses a new iteration variable, a, and depicts an inductive proof that *the arrow labeled "$(g')^a$" is "correct"*.

$$\Big\langle 0, x, i, \vec{y}_n, \vec{Z}_m, \vec{0}_r \Big\rangle \quad \xrightarrow[\text{cf. (1)}]{P} \qquad \Big\langle h(\vec{y}_n), x, 0, \vec{y}_n, \vec{Z}_m, \text{don't care} \Big\rangle$$

$$\xrightarrow[\text{I.H.}]{(g')^a} \qquad \langle f(a, \vec{y}_n), x, a, \vec{y}_n, \cdots, \text{don't care}\rangle$$

$$\xrightarrow[\text{I.S.}]{(g')\ [\text{cf. (2)}]} \Big\langle g\big(a, \vec{y}_n, f(a, \vec{y}_n)\big), x, a+1, \vec{y}_n, \cdots, \text{don't care} \Big\rangle$$

The induction diagrammed above goes like this: Assume correctness of the arrow labeled $(g')^a$ for some fixed $a \geq 0$ (I.H.) But then the arrow for $(g')^{a+1}$ is also correct for it is precisely the third arrow: Notice that $f(a+1, \vec{y}_n) = g\big(a, \vec{y}_n, f(a, \vec{y}_n)\big)$. Of course, $a = 0$ is the case

$$\Big\langle h(\vec{y}_n), x, 0, \vec{y}_n, \vec{Z}_m, \text{don't care} \Big\rangle \xrightarrow{(g')^0} \Big\langle f(0, \vec{y}_n), x, 0, \vec{y}_n, \vec{Z}_m, \text{don't care} \Big\rangle$$

which is correct: $f(0, \vec{y}_n) = h(\vec{y}_n)$.

To use the $(P; Q)$ semantics, (4) above, we simply set $a = x$. By 2.2.1.1, the right hand side of (5) is the first projection ("along z") of this semantics, and we have proved it to be $f(x, \vec{y}_n)$. We have proved (5). □

2.2.2 Incompleteness of $\mathcal{PR}$

We encounter in this subsection our first application of Cantor's diagonalization argument to computability. We will argue that $\mathcal{PR}$ cannot possibly contain all the *intuitively computable total* functions. We say that $\mathcal{PR}$ is *incomplete* with respect to the notion of "computable total function"—or, it is an incomplete *formalism* of computable functions.

A fully mathematical version of this fact will be revisited, and proved, first in Section 2.4 and later revisited in Section 2.9. The argument is easy in the presence of the result of Corollary 2.2.1.5.

We proceed as follows:

(A) The reader will readily accept that he can algorithmically tell whether a string P over the alphabet Σ on p. 131 is a well-formed loop program (syntactically speaking) or not.

(B) We can algorithmically build the list, $List_1$, of *all strings over* Σ—i.e., Σ^+ (p. 40). Here is how: We list strings by increasing length, and in each length group, we list them *lexicographically* (alphabetically).[69]

(C) Simultaneously with building $List_1$, we build $List_2$ as follows: For every string P generated in $List_1$, we copy it into $List_2$ iff $P \in L$ (which we can test algorithmically, by (A)).

---

[69]Fix the ordering of Σ as listed in (1) on p. 131. Lexicographic order is the resulting alphabetic order.

(D) Simultaneously with building $List_2$, we build $List_3$: For every P (necessarily a program) copied in $List_2$, we copy all the finitely many strings P_Y^X (for all choices of X and Y in P) alphabetically—we "linearize" the string P_Y^X as $P; X; Y$—into $List_3$.

At the end of all this we have an algorithmic list of *all* the functions $\lambda x.f(x)$ of $\mathcal{PR}$, listed by their aliases, the P_Y^X. Let us call this list

$$f_0, f_1, f_2, \ldots, f_x, \ldots$$

By Cantor's "diagonalization method" we define a new function d for all x as follows:

$$d(x) = f_x(x) + 1 \tag{1}$$

Two observations:

1. d is total (obvious, since each f_x is) and *intuitively* computable. Indeed, to compute $d(a)$ we generate the lists long enough until we find the a-th item (counting as in $0, 1, 2, \ldots, a$) in $List_3$. This item has the format P_Y^X—i.e., as a loop program with its designated (one) input and output variables. We execute this program with input value a (in X). Once it terminates, we add 1 to what Y holds and we are done! This is $d(a)$.

2. The function d is not in the list! For otherwise, $d = f_i$ for some $i \geq 0$. We get a contradiction as in 1.3.0.50:

$$f_i(i) \stackrel{\text{by } d=f_i}{=} d(i) \stackrel{\text{by (1) above}}{=} f_i(i) + 1$$

2.2.2.1 Remark. We elaborate somewhat on the claim we made in (A) above: "The reader will readily accept ...". There is not much that needs elaboration regarding recognition of instructions of types (I)–(III) of Definition 2.2.0.18. A word about checking that the words **Loop** and **end** *balance* each other (the reader will likely be familiar with this process from first year programming courses, where one often is asked to write programs that "recognize arithmetic expressions"): We will use a *stack*, that is, an *ordered set of data* where we can *add* or *delete* new data always at *the same end* of the ordered set, but nowhere else. This end is known as the *top of the stack*.

Pause. The reader probably already knows about stacks in the programming context. Thus, we do not need to define them carefully here [stacks will reappear in a formal context when we turn our attention to *pushdown automata* (PDA) later in this volume]. For now we simply observe that, as the word suggests, a stack of data acts like a stack of plates in a cafeteria!◄

Back to the task: We want to *verify* (or reject, as the case may be) the claim that a given string P over the alphabet Σ of p. 131 is a loop program.

To this end we scan the string P from left to right. We look for ";" as separators, as in 2.2.0.17. Of course, there is no start-separator nor end-separator.

Whenever we recognize an instruction among the types (I)–(III), we resume our scanning. Whenever we recognize a string of the type "**Loop** X" (for any X) we add it—or as the jargon has it, *push* it—into the stack. Whenever we recognize a string of the type "**end**" we do:

- If the top of the stack contains a string of the type "**Loop** X" (for any X), then delete it—or *pop* it, as we say—and resume scanning.

- If the condition fails—where clearly the stack is empty, as we push nothing else into it—then we stop the process and reject P

We also stop and reject P if we encounter some string (other than separators) which does not fit the categories (I)–(III), **Loop** X and **end**.

Otherwise, once we have scanned all of P, we accept it as a well-formed loop program. The push/pop part of the process ensures that the **Loop** X and **end** strings balance each other like left and right brackets in a well-formed mathematical expression. □

The reader may wish to experiment with actually programming a loop program "parser", using the above ideas.

2.3   URM COMPUTATIONS AND THEIR ARITHMETIZATION

We now return to the systematic development of the basic theory of partial recursive functions, with a view of gaining an insight in the inherent limitations of the computing processes. Instrumental to this study is a mathematical characterization of what is going on during a URM computation as well as a mathematical "coding", as a primitive recursive predicate, of the statement "*the URM M, when presented with input x has a terminating computation, coded by the number y*" —the so-called Kleene-predicate. We achieve this "mathematization" via a process that Gödel (1931) invented in his paper on incompleteness of arithmetic, namely, *arithmetization*. The arithmetization of URM computations is our first task in this section. This must begin with a mathematically precise definition of "URM computation".

As an "agent" executes some URM's, M, instructions, it generates at each step *instantaneous descriptions* (*IDs*)—intuitively, "snapshots"—of a computation. The information each such description includes is simply the values of each variable of M, and the label (instruction number) of *the instruction that is about to be executed next*—the so-called *current* instruction.

In this section we will *arithmetize* URMs and their computations—just as Gödel did in the case of formal arithmetic and its proofs (loc. cit.)—and prove a cornerstone result of computability, the "normal form theorem" of Kleene that, essentially, says that the URM programming language is rich enough to allow us write a *universal program for computable functions*. Such a program, U, receives two inputs: One is a URM description, M, and the other is "data", x. U then simulates M on the data, behaving exactly as M would on input x. Programmers may call such a universal program an *interpreter* or a *compiler*.

2.3.0.2 Definition. (Codes for Instructions) The instructions are coded—using prime-power coding as in Definition 2.1.2.44—as follows, where $X1^i$ is short for

$$X\overbrace{1\cdots1}^{i \text{ ones}}$$

(1) $L: X1^i \leftarrow a$ has code $[1, L, i, a]$.

(2) $L: X1^i \leftarrow X1^i + 1$ has code $[2, L, i]$.

(3) $L: X1^i \leftarrow X1^i \doteq 1$ has code $[3, L, i]$.

(4) $L:$ **if** $X1^i = 0$ **goto** P **else goto** R has code $[4, L, i, P, R]$.

(5) $L:$ **stop** has code $[5, L]$. □

The first component of each instruction code z, $(z)_0$, denotes the *instruction type*, the second—$(z)_1$—denotes the label, and the remaining components give enough information for us to uniquely know what precise instruction we are talking about. For example, in $z = [3, L, i]$ we read that we are talking about the "decrement by one" instruction $((z)_0 = 3)$ applied to $X1^i$ $((z)_2 = i)$, which is found at label L $((z)_1 = L)$.

In turn, we code a URM M as an ordered sequence of numbers, each being a code for an instruction. Thus given a code z [i.e., z codes *something*: $Seq(z)$ is true] we can determine algorithmically whether z codes some URM. This remark is made precise in Theorem 2.3.0.3 below.

2.3.0.3 Theorem. *The relation $URM(z)$ that holds precisely if z codes a URM is in* $\mathcal{PR}_*$.

Proof. In what follows we employ shorthand such as $(\exists z, w)_{<u}$ for $(\exists z)_{<u}(\exists w)_{<u}$, and similarly for longer quantifier groupings, as well as for $\forall$.

$$URM(z) \equiv Seq(z) \wedge (z)_{lh(z)\doteq1} = [5, lh(z)]\ ^{70} \wedge$$

$$(\forall L)_{<lh(z)}\Bigg(Seq((z)_L) \wedge (L \neq lh(z) \doteq 1 \rightarrow ((z)_L)_0 \neq 5) \wedge$$

$$\Big\{ (z)_L = [5, L+1] \vee$$

$$(\exists i, a)_{\leq z}(z)_L = [1, L+1, i+1, a] \vee$$

$$(\exists i)_{\leq z}\Big\{ (z)_L = [2, L+1, i+1] \vee$$

$$(z)_L = [3, L+1, i+1] \vee$$

---

[70]Note that $z = \Big[(z)_0, \ldots, (z)_{lh(z)\doteq1}\Big]$. Since labels are positive, the last label is $lh(z)$. A similar comment holds about "$(\exists i, a)_{\leq z}(z)_L = [1, L+1, i+1, a]$", etc. Why $i + 1$? Because the variables are $X1, X11, X111, \ldots$

$$(\exists M, R)_{<lh(z)}(z)_L$$

$$= [4, L + 1, i + 1, M + 1, R + 1]\big\}\big\}\Big)$$ □

2.3.0.4 Remark. (Normalizing Input/Output:) There is clearly no loss of generality (why?) in assuming that any URM that computes a function of $n \geq 1$ inputs does so using $X11$ through $X1^{n+1}$ as *input* variables and $X1$ as the *output* variable. Such a URM will have at least two instructions, since the **stop** instruction does not reference any variables. □

2.3.0.5 Definition. An ID of a computation of a URM M is an ordered sequence $L; a_1, \ldots, a_r$, where all of M's variables appear *among* the $X1, X11, \ldots X1^r$—the latter denoted in metanotation as $\mathbf{x_1}, \ldots, \mathbf{x_r}$—and a_i is the current value of $\mathbf{x_i}$ immediately before instruction L is executed. L points precisely to the *current instruction*, meaning the immediately next to be executed.

All IDs have the same length, and we say that ID $I_1 = L; a_1, \ldots, a_r$ *yields* ID $I_2 = P; b_1, \ldots, b_r$, in symbols $I_1 \vdash I_2$, *exactly when*

(i) L labels "$\mathbf{x_i} \leftarrow c$", and I_1 and I_2 are identical, except that $b_i = c$ and $P = L+1$.

(ii) L labels "$\mathbf{x_i} \leftarrow \mathbf{x_i} + 1$", and I_1 and I_2 are identical, except that $b_i = a_i + 1$ and $P = L + 1$.

(iii) L labels "$\mathbf{x_i} \leftarrow \mathbf{x_i} \dot{-} 1$", and I_1 and I_2 are identical, except that $b_i = a_i \dot{-} 1$ and $P = L + 1$.

(iv) L labels "if $\mathbf{x_i} = 0$ goto R else goto Q", and I_1 and I_2 are identical, except that $P = R$ if $a_i = 0$, while $P = Q$ otherwise.

(v) L labels "**stop**", and I_1 and I_2 are identical.

A *terminating computation* of M with input $a_1, \ldots, a_k$ is a sequence $I_0, \ldots, I_n$ such that for all $i < n$ we have $I_i \vdash I_{i+1}$ and for some $j \leq n$, I_j has as 0th member the label of **stop**. Moreover, I_0 is *initial*; that is,

$$I_0 = 1; 0, a_1, \ldots, a_k, \underbrace{0, \ldots, 0}_{r - k - 1 \text{ 0s}}$$

The above reflects the normalizing convention of 2.3.0.4, and the standard convention of *implicitly*—i.e., not as part of the computation—setting all the non-input variables to 0, i.e., the $\mathbf{x_{k+2}}, \ldots, \mathbf{x_r}$ and $\mathbf{x_1}$, *before* the computation "starts".

The *length* or *run time* of the computation is its number of *steps* $I_i \vdash I_{i+1}$: That is, n. □

We code an ID $I = L; a_1, \ldots, a_r$ as $code(I) = [L, a_1, \ldots, a_r]$ and a terminating computation $I_0, \ldots, I_n$ by $[code(I_0), \ldots, code(I_n)]$.

2.3.0.6 Theorem. *For any $n \geq 1$, the relation $Comp^{(n)}(z, y)$, which is true iff y codes a terminating computation of the n-input normalized URM coded by z is primitive recursive.*

Proof. By the remark on p. 143, which normalizes the input/output convention, it must be that $lh(y) \geq 2$. In the course of the proof we will want to keep our quantifiers bounded by some primitive recursive function so that the placement of $Comp^{(n)}(z, y)$ in $\mathcal{PR}$ can be achieved.

So, how long need an ID be? Given its format (2.3.0.5), it suffices that it is as long as the *largest* index j of of any variable X_j that occurs in the URM; *plus one*.

Since the maximum j is $\max\{((z)_i)_2 : i < lh(z)\}$ and $((z)_i)_2 < z$ we adopt the generous, but simple, bound $z + 1$. Observe next that

$$Comp(z, y) \equiv URM(z) \wedge Seq(y) \wedge (\forall i)_{< lh(y)} \big(Seq((y)_i) \wedge lh((y)_i) = z + 1\big) \wedge$$
$$lh(y) > 1 \wedge (\forall j)_{< lh(y) \dot{-} 1} yield(z, (y)_j, (y)_{j+1}) \wedge$$

{**Comment.** The last ID surely has the label of z's **stop**.} $((y)_{lh(y) \dot{-} 1})_0 = lh(z) \wedge$

{**Comment.** The initial ID.} $((y)_0)_0 = 1 \wedge ((y)_0)_1 = 0 \wedge$
$$(\forall i)_{\leq z} \big(n + 1 < i \rightarrow ((y)_0)_i = 0\big)$$

The relation "$yield(z, (y)_j, (y)_{j+1})$" above says "URM z causes $(y)_j \vdash (y)_{j+1}$". The notation "$yield(z, u, v)$" is thus shorthand that expands as follows (cf. 2.3.0.5 and 2.3.0.2):

$$yield(z, u, v) \equiv (\exists k)_{\leq z}(\exists L)_{< lh(z)}\Big(L + 1 = (u)_0 \wedge k > 0 \wedge \Big\{$$

$$(\exists a)_{\leq z}\big((z)_L = [1, L+1, k, a] \wedge v = 2p_k^{a+1} \left\lfloor u/p_k^{\exp(k, u)} \right\rfloor \big) \text{[71]}) \vee$$

$$((z)_L = [2, L+1, k] \wedge v = 2p_k u) \vee$$

$$((z)_L = [3, L+1, k] \wedge v = 2(\text{if } (u)_k = 0 \text{ then } u \text{ else } \lfloor u/p_k \rfloor)) \vee$$

$$(\exists P, R)_{\leq lh(z)}\big((z)_L = [4, L+1, k, P, R] \wedge P > 0 \wedge R > 0 \wedge$$

$$v = \text{if } (u)_k = 0 \text{ then } \left\lfloor u/2^{L+2} \right\rfloor 2^{P+1}$$
$$\text{else } \left\lfloor u/2^{L+2} \right\rfloor 2^{R+1}\big) \vee$$

$$((z)_L = [5, L+1] \wedge v = u)\Big\}\Big) \qquad \square$$

2.3.0.7 Corollary. (The Kleene T-predicate) *For each $n \geq 1$, the Kleene predicate $T^{(n)}(z, \vec{x}_n, y)$ that is true precisely when the n-input URM z with input $\vec{x}_n$ has a terminating computation y, is primitive recursive.*

Proof. By earlier remarks, $T^{(n)}(z, \vec{x}_n, y) \equiv Comp^{(n)}(z, y) \wedge ((y)_0)_2 = x_1 \wedge ((y)_0)_3 = x_2 \wedge \ldots \wedge ((y)_0)_{n+1} = x_n$. $\qquad \square$

---

[71]The effect of "$L + 1 : X1^k \leftarrow a$" on ID $u = \langle L+1, \ldots \rangle$ is to change $L+1$ to $L+2$ (effected by the factor 2) and change the current value of $X1^k$, i.e., $(u)_k$—stored in the ID as a factor $p_k^{\exp(k, u)}$, a factor that we remove by dividing u by it—to a, this being stored in v as a factor p_k^{a+1}.

Recalling that for any predicate $R(y, \vec{x})$, $(\mu y) R(y, \vec{x})$ is alternative notation for $(\mu y) \chi_R(y, \vec{x})$—cf. 2.1.2.39—we have:

2.3.0.8 Corollary. (The Kleene Normal Form Theorem)

(1) *For any input/output normalized URM M of code z (2.3.0.2) and n inputs, we have that $M_{X1}^{X11,\dots X1^{n+1}}$ is defined on the input $\vec{x}_n$ iff $(\exists y) T^{(n)}(z, \vec{x}_n, y)$.*

(2) *There is a primitive recursive function d such that for any $\lambda \vec{x}_n. f(\vec{x}_n) \in \mathcal{P}$ there is a number z and we have for all $\vec{x}_n$:*

$$f(\vec{x}_n) \simeq d\big((\mu y) T^{(n)}(z, \vec{x}_n, y)\big)$$

Proof. Statement (1) is immediate as "$(\exists y) T^{(n)}(z, \vec{x}_n, y)$" says that there is a terminating computation of M (coded as z) on input $\vec{x}_n$.

For (2), let $f = M_{X1}^{X11,\dots X1^{n+1}}$, where M is a normalized URM of code z. The role of d is to extract, from a terminating computation's *last* ID, its 1st component. Thus, for all y, we let $d(y) = \big((y)_{lh(y) \dot- 1}\big)_1$. □

In what follows, the term *computation* will stand for *terminating computation*. Note that the "complete" equality (cf. 1.2.0.11) in the corollary, (2), becomes standard equality, =, iff we do have a (terminating) computation.

2.3.0.9 Definition. (ϕ-Notation of Rogers (1967)) We denote by $\phi_z^{(n)}$ the partial recursive n-ary function computed by a URM of code z, as $M_{X1}^{X11,\dots,X1^{n+1}}$. That is, $\phi_z^{(n)} = \lambda \vec{x}_n. d\big((\mu y) T^{(n)}(z, \vec{x}_n, y)\big)$. We usually write ϕ_z for $\phi_z^{(1)}$ and $T(z, x, y)$ for $T^{(1)}(z, x, y)$. □

Pause. Why did I not write "$\simeq$" above? (Cf. 1.8.32.)◄

2.3.0.10 Remark. If $f = \phi_i^{(n)}$, for some URM code i, then we call i a ϕ-index of f. □

We now readily obtain the very important number-theoretic characterization of $\mathcal{P}$, a class that was originally defined in 2.1.1.2 via the URM formalism. This result is a direct consequence of 2.3.0.8 and is the direct analog of Theorem 2.2.1.5, about $\mathcal{PR}$.

2.3.0.11 Corollary. (Number-Theoretic Characterization of $\mathcal{P}$) *$\mathcal{P}$ is the closure of the same $\mathcal{I}$ that we used for $\mathcal{PR}$, under composition, primitive recursion, and unbounded search.*

Proof. If we temporarily call $\widetilde{\mathcal{P}}$ the closure that we mentioned in the corollary, then since $\mathcal{P}$ contains $\mathcal{I}$ and is closed under the three stated operations (cf. 2.1.1.12, 2.1.1.16, and 2.1.1.19), we immediately have $\widetilde{\mathcal{P}} \subseteq \mathcal{P}$.

Conversely, ignoring closure under (μy) for a moment, we get $\mathcal{PR} \subseteq \widetilde{\mathcal{P}}$. Thus $\lambda \vec{x}_n. d((\mu y) T^{(n)}(z, \vec{x}_n, y)) \in \widetilde{\mathcal{P}}$, for all z. This shows $\mathcal{P} \subseteq \widetilde{\mathcal{P}}$. □

The preceding corollary provides an alternative formalism—that is, a *syntactic, finite description* other than via URM programs—for the functions of $\mathcal{P}$: Via $\mathcal{P}$-derivations, which can be defined totally analogously with the case of 2.1.2.1, by adding the operation of unbounded search. Both types of derivation are special cases of the general case of 1.6.0.6.

2.3.0.12 Remark. (1) The *normal form theorem* says, in particular, that every *unary* function that is computable in the technical sense of 2.1.1.2—or, equivalently, 2.3.0.11—can be expressed as an unbounded search followed by a composition, *using a toolbox of just* two *primitive recursive functions(!)*: d and $\lambda zxy.\chi_T(z,x,y)$. This representation, or "normal form", is parametrized by z, which denotes a URM M that computes the function in a normalized manner: as M_{X1}^{X11}. *Thus what we had set out to do at the beginning of this section is now done:* The two-input URM $\mathbf{U}$ that computes $\lambda zx.d\big((\mu y)T(z,x,y)\big)$—a computable function by 2.3.0.11—*is* universal, in precise the same way that *compilers*[72] of practical computing are: The universal URM $\mathbf{U}$ accepts two inputs—a program M, coded as a number z, *and* data for said program, x. It then "interprets" and acts exactly as program z would on x, i.e., as M_{X1}^{X11}.

(2) From Definition 2.3.0.2 it is clear that *not* every $z \in \mathbb{N}$ represents a URM. *Nevertheless, "$\lambda x.d\big((\mu y)T(z,x,y)\big)$" in Definition 2.3.0.9 is meaningful for* all *natural numbers z regardless of whether they code a URM or not,*

and is in $\mathcal{P}$, by the latter's closure properties.

Thus, if z is *not* a URM code, then $T(z,x,y)$ will simply be false, for all x, and all y; thus we will have $\phi_z(x) \uparrow$ for all x. *This is perfectly fine*! Indeed, it is consistent with the phenomenon where a real-life computer program that is *not* syntactically correct (like our z here) will not be translated by the compiler and thus will not run. Therefore, for any input it will decline to offer an output; the corresponding function will be totally undefined.

Due to these considerations we *extend* the concept of ϕ-index to all of $\mathbb{N}$, and correspondingly remove the hedging from Definition 2.3.0.9: "computed by a URM of code z, as $M_{X1}^{X11,\ldots,X1^{n+1}}$".

We now say: *For all $z \in \mathbb{N}$, $\phi_z^{(n)}$ denotes the function $\lambda \vec{x}_n.d\big((\mu y)T^{(n)}(z,\vec{x}_n,y)\big)$.*

(3) In view of the above redefinition, Definition 2.1.1.2 can now be rephrased as "$\lambda \vec{x}_n.f(\vec{x}_n) \in \mathcal{P}$ iff, for some $z \in \mathbb{N}$, $f = \phi_z^{(n)}$"—not just "for some z that is a URM code". $\square$

2.3.0.13 Exercise. Prove that every function of $\mathcal{P}$ has infinitely many ϕ-indices.

Hint. There are infinitely many ways to modify a program and yet have all programs so obtained compute the same function. $\square$

---

[72]A "compiler" translates "high level" programs written in C, Pascal, etc., into machine language so they can be "understood" by a computer, and therefore be executed on given input data.

2.3.0.14 Example. The nowhere-defined function, $\emptyset$, is in $\mathcal{P}$, as it can be obtained from any invalid code. For example,

$$\emptyset = \lambda x.d\big((\mu y)T(0, x, y)\big)$$

Pause. Why is 0 not a URM code?◄

It can, however, also be obtained from a program that compiles all right. Setting $\widetilde{S} = \lambda yx.x + 1$ we note:
(1) $\lambda x.(\mu y)\widetilde{S}(y, x) \in \mathcal{P}$ by 2.1.2.12 and 2.1.1.19.
(2) By the techniques of 2.1.1.17 we can write a program for $\emptyset = \lambda x.(\mu y)\widetilde{S}(y, x)$.

As a side-effect we have that $\mathcal{PR} \neq \mathcal{P}$ and $\mathcal{R} \neq \mathcal{P}$.   □

2.4   A DOUBLE RECURSION THAT LEADS OUTSIDE THE PRIMITIVE RECURSIVE FUNCTION CLASS

We saw in Subsection 2.2.2 that there are intuitively computable total functions that are not in $\mathcal{PR}$. This means that this class is an inadequate, or *incomplete*, formalization of the concept "total computable function". While the proof that our counterexample function is not in $\mathcal{PR}$ can be trivially completed into a mathematical proof, the part about it being "intuitively" computable was informal by virtue of the imprecision in the term "intuitively computable". The current section revisits this issue in a totally mathematical fashion. First, we produce a total function that is *provably* not in $\mathcal{PR}$. Second, we mathematically establish that this function is a member of $\mathcal{R}$, showing therefore that $\mathcal{PR} \subset \mathcal{R}$. This says more than "there exists an *intuitively* computable" $f \notin \mathcal{PR}$, since we produce a provably computable such f, by placing it within our $\mathcal{P}$ formalism.

We should note that it is customary in computability to talk of a "formalism" (such as that of $\mathcal{PR}$, for example)—and we utilized this jargon several times already. The term "formalism" entails a *syntactic* (= *formal*) method of *describing* our (mathematical) objects of study, which are then studied by rigorous mathematical methods.[73] Thus, either of the approaches, via loop programs, or via *derivations* (or *closures*; cf. 2.1.2.3) for the definition of $\mathcal{PR}$ is a formalized approach in this sense. The same holds true of either the URM-based or the $\mathcal{P}$-as-a-closure (cf. 2.3.0.11) approaches for the introduction of $\mathcal{P}$.

We cannot say this for $\mathcal{R}$ though! If we *could* have a syntactic definition of $\mathcal{R}$, then we could repeat the diagonalization argument of Subsection 2.2.2 to find a total "intuitively computable" $g \notin \mathcal{R}$. At the present state of knowledge it does

---

[73] A pure unadulterated formalism also employs the purely syntactic (or formal) application of logic, based on appropriately chosen nonlogical axioms. In such mode of application of logic, meaning or semantics is redundant, although well-chosen mathematical/logical *argot*, as in Bourbaki (1966) and Tourlakis (2003a), may create the feeling that one argues pretty much as in "everyday mathematics". As we have noted already in the Preface, while our use of logic is mathematically rigorous and correct, we do *not* insist on basing our reasoning on nonlogical axioms, e.g., such as those of Peano arithmetic.

not appear that such a g exists! Indeed, Church's conjecture—famously known as "Church's Thesis"—states categorically that *no intuitively computable total function exists outside $\mathcal{R}$*!

Quite apart from the empirical question of whether "Church's Thesis" is correct or not, we note that *within the formalization of $\mathcal{P}$*, in which the functions ϕ_z of $\mathcal{R}$ have *finite descriptions* $z \in \mathbb{N}$, *it is impossible to have an enumeration of all such descriptions* via a *recursive function* of $\mathcal{R}$.

So, at least, within the $\mathcal{P}$-formalism *we cannot diagonalize out of $\mathcal{R}$* by an argument like the one given in 2.2.2. This observation will be proved mathematically without invoking any "beliefs". Cf. 2.5.0.29

Pause. Why can't diagonalization of the type employed in 2.2.2 show the existence of partial, intuitively computable functions outside the syntactically defined $\mathcal{P}$, thus handily rejecting Church's Thesis?◄

Because $f_i(i) \simeq f_i(i) + 1$ is not necessarily a contradiction! (why?)

2.4.0.15 Remark. Can we tell, given z, whether ϕ_z is total (hence in $\mathcal{R}$)? No because if we could then we could build an enumeration of all such z—that we promised, above, to prove impossible. Indeed, if we could so test (computably), then for each $z = 0, 1, 2, 3, \ldots$, if z defines a total function, then add it to a list "List". This "List" would be an enumeration that we said we cannot have. □

2.4.1 The Ackermann Function

The "Ackermann function" was proposed, naturally, by Ackermann. The version here is a simplification offered by Robert Ritchie.

What the function does is to provide us with an example of a number-theoretic *intuitively computable, total* function that is not in $\mathcal{PR}$. But this function is more than just *intuitively* computable! It *is* computable—no hedging—as we will show by showing it to be a member of $\mathcal{R}$.

Another thing it does is that it provides us with an example of a function $\lambda \vec{x}.f(\vec{x})$ that is "hard to compute" ($f \notin \mathcal{PR}$) but whose *graph*—that is, the predicate $\lambda y \vec{x}.y = f(\vec{x})$—is nevertheless "easy to compute" ($\in \mathcal{PR}_*$).[74]

2.4.1.1 Definition. The Ackermann function, $\lambda nx.A_n(x)$, is given, for all $n \geq 0, x \geq 0$ by the equations

$$A_0(x) = x + 2$$
$$A_{n+1}(x) = A_n^x(2)$$

---

[74]Here the colloquialisms "easy to compute" and "hard to compute" are aliases for "primitive recursive" and "not primitive recursive", respectively. This is a hopelessly coarse rendering of *easy/hard* and a much better gauge for the runtime complexity of a problem is on which side of $O(2^n)$ it lies. However, our gauge will have to do for now: All I want to leave you with is that for some functions it is *easier* to compute the graph—to the quantifiable extent that it is in $\mathcal{PR}_*$—than the function itself, to the extent that it fails being primitive recursive.

where h^x is function iteration (cf. 2.1.5.1).   □

The λ-notation makes it clear that both n and x are *arguments* of the Ackermann function. While we could have written $A(n, x)$ instead, it is notationally less challenging to use the chosen notation. We refer to the n as the *subscript argument*, and to x as the *inner argument*.

2.4.1.2 Remark. An alternative way to define the Ackermann function, extracted directly from Definition 2.4.1.1, is as follows:

$$A_0(x) = x + 2$$
$$A_{n+1}(0) = 2$$
$$A_{n+1}(x + 1) = A_n(A_{n+1}(x))$$ □

2.4.2 Properties of the Ackermann Function

We present a sequence of less than earth-shattering—but useful—theorems. So we will just call them lemmata.

2.4.2.1 Lemma. *For each $n \geq 0$, $\lambda x.A_n(x) \in \mathcal{PR}$.*

Proof. Induction on n: For the basis, clearly $A_0 = \lambda x.x + 2 \in \mathcal{PR}$. Assume now the case for (arbitrary, fixed) n—i.e., $A_n \in \mathcal{PR}$—and go to that for $n+1$. Immediate from Definition 2.4.1.2, last two equations.   □

It turns out that the function blows up in size far too fast with respect to the argument n. We now quantify this remark.

The following unassuming lemma is the key to proving the growth properties of the Ackermann function. It is also the least straightforward to prove, as it requires a double induction—at once on n and x—as dictated by the fact that the "recursion" of Definition 2.4.1.2 does not leave any argument fixed.

2.4.2.2 Lemma. *For each $n \geq 0$ and $x \geq 0$, $A_n(x) > x + 1$.*

Proof. We start an induction on n:
  n-Basis. $n = 0$: $A_0(x) = x + 2 > x + 1$; true.
  n-I.H.[75] For all x and a fixed (but unspecified) n, *assume $A_n(x) > x + 1$.*
  n-I.S.[76] For all x and the above fixed (but unspecified) n, we must *prove $A_{n+1}(x) > x + 1$.*
  We do the n-I.S. by induction on x:
    x-Basis. $x = 0$: $A_{n+1}(0) = 2 > 1$; true.

---

[75] To be precise, what we are proving is "$(\forall n)(\forall x)A_n(x) > x + 1$". Thus, as we start on an induction on n, its I.H. is "$(\forall x)A_n(x) > x + 1$" for a fixed unspecified n.
[76] To be precise, the step is to prove—from the basis and I.H.—"$(\forall x)A_{n+1}(x) > x + 1$" for the n that we fixed in the I.H. It turns out that this is best handled by induction on x.

x-I.H. For the above fixed n, we now fix an x (but leave it unspecified) for which we *assume* $A_{n+1}(x) > x + 1$.

x-I.S. For the above fixed (but unspecified) n and x, *prove* $A_{n+1}(x + 1) > x + 2$.

Well,

$$
\begin{aligned}
A_{n+1}(x + 1) &= A_n(A_{n+1}(x)) \quad \text{by Def. 2.4.1.2} \\
&> A_{n+1}(x) + 1 \quad \text{by } n\text{-I.H.} \\
&> x + 2 \quad \text{by } x\text{-I.H.} \qquad\qquad \square
\end{aligned}
$$

2.4.2.3 Lemma. $\lambda x. A_n(x) \nearrow$.

"$\lambda x. f(x) \nearrow$" means that the (total) function f is *strictly increasing*, that is, $x < y$ implies $f(x) < f(y)$, for any x and y. Clearly, to establish the property one just needs to check for the arbitrary x that $f(x) < f(x + 1)$.

Proof. We handle two cases separately.

A_0: $\lambda x. x + 2 \nearrow$; immediate.

A_{n+1}: $A_{n+1}(x+1) = A_n(A_{n+1}(x)) > A_{n+1}(x)+1$—the ">" by Lemma 2.4.2.2. $\qquad \square$

2.4.2.4 Lemma. $\lambda n. A_n(x + 1) \nearrow$.

Proof. $A_{n+1}(x + 1) = A_n(A_{n+1}(x)) > A_n(x + 1)$—the ">" by Lemmata 2.4.2.2 (left argument > right argument) and 2.4.2.3. $\qquad \square$

The "$x + 1$" in Lemma 2.4.2.4 is important since $A_n(0) = 2$ for all n. Thus $\lambda n. A_n(0)$ is increasing but *not* strictly (constant).

2.4.2.5 Lemma. $\lambda y. A_n^y(x) \nearrow$.

Proof. $A_n^{y+1}(x) = A_n(A_n^y(x)) > A_n^y(x)$—the ">" by Lemma 2.4.2.2. $\qquad \square$

2.4.2.6 Lemma. $\lambda x. A_n^y(x) \nearrow$.

Proof. Induction on y: For $y = 0$ we want that $\lambda x. A_n^0(x) \nearrow$, that is, $\lambda x. x \nearrow$, which is true. We next take as I.H. that

$$A_n^y(x + 1) > A_n^y(x) \tag{1}$$

We want

$$A_n^{y+1}(x + 1) > A_n^{y+1}(x) \tag{2}$$

But (2) follows from (1) and Lemma 2.4.2.3, by applying A_n to both sides of ">". $\qquad \square$

2.4.2.7 Lemma. *For all* n, x, y, $A_{n+1}^y(x) \geq A_n^y(x)$.

Proof. Induction on y: For $y = 0$ we want that $A_{n+1}^0(x) \geq A_n^0(x)$, that is, $x \geq x$, which is true. We now take as I.H. that

$$A_{n+1}^y(x) \geq A_n^y(x)$$

We want

$$A_{n+1}^{y+1}(x) \geq A_n^{y+1}(x)$$

This is true because

$$A_{n+1}^{y+1}(x) = A_{n+1}\left(A_{n+1}^y(x)\right)$$
$$\text{by Lemma 2.4.2.4}$$
$$\geq A_n\left(A_{n+1}^y(x)\right)$$
$$\text{Lemma 2.4.2.3 and I.H.}$$
$$\geq A_n^{y+1}(x) \qquad\qquad \square$$

2.4.2.8 Definition. Given a predicate $P(\vec{x})$, we say that $P(\vec{x})$ *is true almost everywhere*—in symbols "$P(\vec{x})$ a.e."—iff the set of (vector) inputs that make the predicate *false* is *finite*. That is, the set $\{\vec{x} : \neg P(\vec{x})\}$ is finite.

A statement such as "$\lambda xy.Q(x, y, z, w)$ a.e." can also be stated, less formally, as "$Q(x, y, z, w)$ a.e. *with respect to x and y*". $\qquad\qquad \square$

2.4.2.9 Lemma. $A_{n+1}(x) > x + l$ *a.e. with respect to x.*

Thus, in particular, $A_1(x) > x + 10^{350000}$ a.e.

Proof. In view of Lemma 2.4.2.4 and the note following it, it suffices to prove

$$A_1(x) > x + l \text{ a.e. with respect to } x$$

Well, since

$$A_1(x) = A_0^x(2) = \overbrace{(\cdots(((y+2)+2)+2)+\cdots+2)}^{x\ 2\text{'s}}\,\|_{\text{evaluated at } y\,=\,2} = 2 + 2x$$

we ask: Is $2 + 2x > x + l$ a.e. with respect to x? It is so for all $x > l - 2$ (only $x = 0, 1, \ldots, l - 2$ fail). $\qquad\qquad \square$

2.4.2.10 Lemma. $A_{n+1}(x) > A_n^l(x)$ *a.e. with respect to x.*

Proof. If one (or both) of l and n is 0, then the result is trivial. For example,

$$A_0^l(x) = \overbrace{(\cdots(((x+2)+2)+2)+\cdots+2)}^{l\ 2\text{'s}} = x + 2l$$

We are done by Lemma 2.4.2.9.

Let us then assume that $l \geq 1$ and $n \geq 1$. We note that (straightforwardly, via Definition 2.4.1.1)

$$A_n^l(x) = A_n(A_n^{l-1}(x))$$

$$= A_{n-1}^{A_n^{l-1}(x)}(2) = A_{n-1}^{A_{n-1}^{A_n^{l-2}(x)}(2)}(2) = A_{n-1}^{A_{n-1}^{A_{n-1}^{A_n^{l-3}(x)}(2)}(2)}(2)$$

The straightforward observation that *we have a "ladder" of k A_{n-1}'s precisely when the topmost exponent is $l - k$* can be ratified by induction on k (left to the reader). Thus we state

$$A_n^l(x) = {}^{k\ A_{n-1}}\left\{ A_{n-1}^{\cdot^{\cdot^{A_{n-1}^{A_n^{l-k}(x)}(2)}}}(2) \right.$$

In particular, taking $k = l$,

$$A_n^l(x) = {}^{l\ A_{n-1}}\left\{ A_{n-1}^{\cdot^{\cdot^{A_{n-1}^{A_n^{l-l}(x)}(2)}}}(2) \right. = {}^{l\ A_{n-1}}\left\{ A_{n-1}^{\cdot^{\cdot^{A_{n-1}^{x}(2)}}}(2) \right. \qquad (*)$$

Let us now take $x > l$.

Thus, by $(*)$,

$$A_{n+1}(x) = A_n^x(2) = {}^{x\ A_{n-1}}\left\{ A_{n-1}^{\cdot^{\cdot^{A_{n-1}^{2}(2)}}}(2) \right. \qquad (**)$$

By comparing $(*)$ and $(**)$ we see that the first "ladder" is topped (after l A_{n-1} "steps") by x and the second is topped by

$$ {}^{x-l\ A_{n-1}}\left\{ A_{n-1}^{\cdot^{\cdot^{A_{n-1}^{2}(2)}}}(2) \right. $$

Thus—in view of the fact that $A_n^y(x)$ increases with respect to each of the arguments n, x, y—we conclude by asking . . .

$$\text{"Is}\quad {}^{x-l\ A_{n-1}}\left\{ A_{n-1}^{\cdot^{\cdot^{A_{n-1}^{2}(2)}}}(2) > x \text{ a.e. with respect to } x?"\right.$$

. . . and answering, "Yes", because by $(**)$ this is the same question as "is $A_{n+1}(x - l) > x$ a.e. with respect to x?", which we answered affirmatively in 2.4.2.9. □

2.4.2.11 Lemma. *For all n, x, y, $A_{n+1}(x + y) > A_n^x(y)$.*

Proof.

$$A_{n+1}(x + y) = A_n^{x+y}(2)$$

$$= A_n^x \Big(A_n^y(2) \Big)$$
$$= A_n^x \Big(A_{n+1}(y) \Big)$$
$$> A_n^x(y) \quad \text{by Lemmata 2.4.2.2 and 2.4.2.6} \qquad \square$$

2.4.3   The Ackermann Function Majorizes All the Functions of $\mathcal{PR}$

We say that a function f *majorizes* another function, g, iff $g(\vec{x}) \leq f(\vec{x})$ for all $\vec{x}$. The following theorem states precisely in what sense "*the Ackermann function majorizes all the functions of $\mathcal{PR}$*".

2.4.3.1 Theorem. *For every function $\lambda\vec{x}.f(\vec{x}) \in \mathcal{PR}$ there are numbers n and k, such that for all $\vec{x}$ we have $f(\vec{x}) \leq A_n^k(\max(\vec{x}))$.*

Proof. The proof is by induction with respect to $\mathcal{PR}$. Throughout I use the abbreviation $|\vec{x}|$ for $\max(\vec{x})$ as this is notationally friendlier.

For the basis, f is one of:

- *Basis.*

    Basis 1. $\lambda x.0$. Then $A_0(x)$ works ($n = 0, k = 1$).

    Basis 2. $\lambda x.x + 1$. Again $A_0(x)$ works ($n = 0, k = 1$).

    Basis 3. $\lambda\vec{x}.x_i$. Once more $A_0(x)$ works ($n = 0, k = 1$): $x_i \leq |\vec{x}| < A_0(|\vec{x}|)$.

- *Propagation with composition.* Assume as I.H. that

$$f(\vec{x}_m) \leq A_n^k(|\vec{x}_m|) \tag{1}$$

    and

$$\text{for } i = 1, \ldots, m, \; g_i(\vec{y}) \leq A_{n_i}^{k_i}(|\vec{y}|) \tag{2}$$

    Then

$$\begin{aligned}
f(g_1(\vec{y}), \ldots, g_m(\vec{y})) &\leq A_n^k(|g_1(\vec{y}), \ldots, g_m(\vec{y})|), \text{ by } (1) \\
&\leq A_n^k(|A_{n_1}^{k_1}(|\vec{y}|), \ldots, A_{n_m}^{k_m}(|\vec{y}|)|), \text{ by 2.4.2.6 and } (2) \\
&\leq A_n^k\Big(|A_{\max n_i}^{\max k_i}(|\vec{y}|)|\Big), \text{ by 2.4.2.6 and 2.4.2.7} \\
&\leq A_{\max(n, n_i)}^{k + \max k_i}(|\vec{y}|), \text{ by 2.4.2.7}
\end{aligned}$$

- *Propagation with primitive recursion.* Assume as I.H. that

$$h(\vec{y}) \leq A_n^k(|\vec{y}|) \tag{3}$$

    and

$$g(x, \vec{y}, z) \leq A_m^r(|x, \vec{y}, z|) \tag{4}$$

Let f be such that

$$f(0, \vec{y}) = h(\vec{y})$$
$$f(x+1, \vec{y}) = g(x, \vec{y}, f(x, \vec{y}))$$

I claim that

$$f(x, \vec{y}) \leq A_m^{rx}\left(A_n^k(|x, \vec{y}|)\right) \tag{5}$$

I prove (5) by induction on x:

For $x = 0$, I want $f(0, \vec{y}) = h(\vec{y}) \leq A_n^k(|0, \vec{y}|)$. This is true by (3) since $|0, \vec{y}| = |\vec{y}|$.

As an I.H. assume (5) for fixed x.

The case for $x + 1$:

$$
\begin{aligned}
f(x+1, \vec{y}) &= g(x, \vec{y}, f(x, \vec{y})) \\
&\leq A_m^r(|x, \vec{y}, f(x, \vec{y})|), \text{ by (4)} \\
&\leq A_m^r\left(\left|x, \vec{y}, A_m^{rx}\left(A_n^k(|x, \vec{y}|)\right)\right|\right), \text{ by the I.H. (5), and 2.4.2.6} \\
&= A_m^r\left(A_m^{rx}\left(A_n^k(|x, \vec{y}|)\right)\right), \text{ by 2.4.2.6 and } A_m^{rx}\left(A_n^k(|x, \vec{y}|)\right) \geq |x, \vec{y}| \\
&= A_m^{r(x+1)}\left(A_n^k(|x, \vec{y}|)\right)
\end{aligned}
$$

With (5) proved, let me set $l = \max(m, n)$. By Lemma 2.4.2.7 I now get

$$f(x, \vec{y}) \leq A_l^{rx+k}(|x, \vec{y}|) \underset{\text{Lemma 2.4.2.11}}{<} A_{l+1}(|x, \vec{y}| + rx + k) \tag{6}$$

Now, $|x, \vec{y}| + rx + k \leq (r+1)|x, \vec{y}| + k$ thus, (6) and 2.4.2.3 yield

$$f(x, \vec{y}) < A_{l+1}((r+1)|x, \vec{y}| + k) \tag{7}$$

To simplify (7) note that there is a number q such that

$$(r+1)x + k \leq A_1^q(x) \tag{8}$$

for all x. Indeed, this is so since (easy induction on y) $A_1^y(x) = 2^y x + 2^y + 2^{y-1} + \cdots + 2$. Thus, to satisfy (8), just take $y = q$ large enough to satisfy $r + 1 \leq 2^q$ and $k \leq 2^q + 2^{q-1} + \cdots + 2$.

By (8), the inequality (7) yields, via 2.4.2.3,

$$f(x, \vec{y}) < A_{l+1}(A_1^q(|x, \vec{y}|)) \leq A_{l+1}^{1+q}(|x, \vec{y}|)$$

(by Lemma 2.4.2.7) which is all we want.   □

2.4.3.2 Remark. Reading the proof carefully we note that the *subscript argument* of the *majorant*[77] is precisely the maximum depth of nesting of primitive recursion that occurs in a derivation of f.

Pause. In which derivation? There are infinitely many.◄

Indeed, the initial functions have a majorant with subscript 0; composition has a majorant with subscript no more than the maximum subscript of the component parts—*no increase*; primitive recursion has a majorant with a subscript that is bigger than the *maximum* subscript of the h- and g-majorants by precisely 1.   □

2.4.3.3 Corollary. $\lambda nx.A_n(x) \notin \mathcal{PR}$.

Proof. By contradiction: If $\lambda nx.A_n(x) \in \mathcal{PR}$ then also $\lambda x.A_x(x) \in \mathcal{PR}$ (identi-fication of variables; cf. 2.1.2.6). By the theorem above, for some n, k, $A_x(x) \leq A_n^k(x)$, for all x, hence, by 2.4.2.10

$$A_x(x) < A_{n+1}(x), \text{ a.e. with respect to } x \tag{1}$$

On the other hand, $A_{n+1}(x) < A_x(x)$ a.e. with respect to x—indeed for all $x > n+1$ by 2.4.2.4—which contradicts (1).   □

2.4.4   The Graph of the Ackermann Function is in $\mathcal{PR}_*$

How does one compute a yes/no answer to the question

$$\text{``}A_n(x) = z\text{?''} \tag{1}$$

Thinking "recursively" (in the programming sense of the word), we will look at the question by considering three cases, according to the definition in the Remark 2.4.1.2:

(a) If $n = 0$, then we will directly check (1) as "is $x + 2 = z$?".

(b) If $x = 0$, then we will directly check (1) as "is $2 = z$?".

(c) In all other cases, i.e., $n > 0$ and $x > 0$, for an appropriate w, we may naturally[78] ask *two* questions [both *must* be answerable "yes" for (1) to be true]: "Is $A_{n-1}(w) = z$?", and "is $A_n(x - 1) = w$?"

Assuming that we want to pursue this by pencil and paper or some other equivalent means, it is clear that the pertinent info that we are juggling are *ordered triples* of numbers such as n, x, z, or $n - 1, w, z$, etc. That is, the letter "A", the brackets, the equals sign, and the position of the arguments (subscript vs. inside brackets) are just ornamentation, and the string "$A_i(j) = k$", *in this section's context*, does not contain any more information than the *ordered* triple "$\langle i, j, k \rangle$".

---

[77]The function that does the majorizing.
[78]$A_n(x) = A_{n-1}(A_n(x - 1))$.

Thus, to "compute" an answer to (1) we need to write down enough triples, in stages (or steps), as needed to justify (1): At each stage we may write a triple $\langle i, j, k \rangle$ down just in case one of (i)–(iii) holds:

(i) $i = 0$ and $k = j + 2$

(ii) $j = 0$ and $k = 2$

(iii) $i > 0$ and $j > 0$, and for some w, we have already written down the two triples $\langle i - 1, w, k \rangle$ and $\langle i, j - 1, w \rangle$.

Pause. Since "$\langle i, j, k \rangle$" abbreviates "$A_i(j) = k$", Lemma 2.4.2.2 implies that $j < k$.◄

Our theory is more competent with numbers (than with pairs, triples, etc.) preferring to *code* tuples into single numbers. Thus if we were to carry out the pencil and paper algorithm *within our theory*, then we would be well advised to code all these triples, which we write down step by step, by single numbers: We will use our usual prime-power coding, $[i, j, k]$.

We note that our computation is "tree-like",[79] since a "complicated" triple such as that of case (iii) above *requires* two similar others to be already written down, each of which in turn will require two *earlier* similar others, etc., until we reach "leaves" [cases (i) or (ii)] that can be dealt with directly without passing the buck.

This "tree", just like the tree of a mathematical proof,[80] can be arranged in a sequence of coded triples $[i, j, k]$ so that the presence of a "$[i, j, k]$" implies that all its dependencies appear *earlier* (to its left).

We will code such a sequence by a single number, u, using the prime-power coding of sequences given in 2.1.2.44:

$$[a_0, \ldots, a_{z-1}] = \Pi_{i < z} p_i^{a_i + 1}$$

In effect, what we are doing is that we arithmetize our pencil-and-paper computation of the answer to question (1) above, a technique that we have already employed and learned in Section 2.3.

Now, given any number u, we can primitively recursively check whether or not it is a code of an Ackermann function computation:

2.4.4.1 Theorem. *The predicate*

$$Comp(u) \stackrel{\text{Def}}{=} u \text{ codes an Ackermann function computation}$$

is in $\mathcal{PR}_*$.

---

[79] A term the reader surely is familiar with, from programming and discrete math courses.

[80] Assuming that modus ponens is the only rule of inference, the proof a formula A depends, in general, on that of "earlier" formulae $X \to A$ and X, which in turn depend (require) earlier formulae each, and so on and so on, until we reach formulae that are axioms.

Proof. We will use some notation that will be useful to make the proof more intuitive. Thus we introduce two predicates: $\lambda vu.v \in u$ and $\lambda vwu.v <_u w$. The first says

$$u = [\ldots, v, \ldots]$$

and the second says

$$u = [\ldots, v, \ldots, w, \ldots]$$

Both are in $\mathcal{PR}_*$ since

$$v \in u \equiv Seq(u) \wedge (\exists i)_{<lh(u)}(u)_i = v$$

and

$$v <_u w \equiv Seq(u) \wedge (\exists i, j)_{<lh(u)}\big((u)_i = v \wedge (u)_j = w \wedge i < j\big)$$

We can now define $Comp(u)$ by a formula that makes it clear that it is in $\mathcal{PR}_*$:

$$Comp(u) \equiv Seq(u) \wedge (\forall v)_{\leq u}\Big(v \in u \rightarrow Seq(v) \wedge lh(v) = 3 \wedge$$

{**Comment:** Case (i), p. 156} $\Big\{(v)_0 = 0 \wedge (v)_2 = (v)_1 + 2 \vee$

{**Comment:** Case (ii)} $(v)_1 = 0 \wedge (v)_2 = 2 \vee$

{**Comment:** Case (iii)} $\big((v)_0 > 0 \wedge (v)_1 > 0 \wedge$

$$(\exists w)_{<v}\big([(v)_0 \,\dot-\, 1, w, (v)_2] <_u v \wedge [(v)_0, (v)_1 \,\dot-\, 1, w] <_u v)\big)\Big\}\Big)$$

The "Pause" on p. 156 justifies the bound on $(\exists w)$ above. Indeed, we could have used the tighter bound "$(v)_2$". Clearly $Comp(u) \in \mathcal{PR}_*$.   □

Thus $A_n(x) = z$ iff $[n, x, z] \in u$ for some u that satisfies $Comp$. For short

$$A_n(x) = z \equiv (\exists u)(Comp(u) \wedge [n, x, z] \in u) \tag{1}$$

If we succeed in finding a bound for u that is a primitive recursive function of n, x, z then we will have succeeded showing:

2.4.4.2 Theorem. $\lambda nxz.A_n(x) = z \in \mathcal{PR}_*$.

Proof. Let us focus on a computation u that as soon as it verifies $A_n(x) = z$ quits, that is, it only codes $[n, x, z]$ and just the needed *predecessor* triples, but no more. How big can such a u be?

Well,

$$u = \cdots p_r^{[i,j,k]+1} \cdots p_l^{[n,x,z]+1} \tag{2}$$

for appropriate l $(=lh(u) - 1)$. For example, if all we want is to verify $A_0(3) = 5$, then $u = p_0^{[0,3,5]+1}$.

Similarly, if all we want to verify is $A_1(1) = 4$, then—since the "recursive calls" here are to $A_0(2) = 4$ and $A_1(0) = 2$—two possible u-values work: $u = p_0^{[0,2,4]+1} p_1^{[1,0,2]+1} p_2^{[1,1,4]+1}$ or $u = p_0^{[1,0,2]+1} p_1^{[0,2,4]+1} p_2^{[1,1,4]+1}$.

How big need l be? No bigger than needed to provide distinct *positions* ($l + 1$ such) in the computation, for all the "needed" triples i, j, k. Since z is the largest possible output (and larger than any input) that is computed, there are no more than $(z + 1)^3$ triples possible, so $l + 1 \leq (z + 1)^3$. Therefore, (2) yields

$$u \leq \cdots p_r^{[z,z,z]+1} \cdots p_l^{[z,z,z]+1}$$

$$= \left(\Pi_{i \leq l} p_i\right)^{[z,z,z]+1}$$

$$\leq p_l^{(l+1)([z,z,z]+1)}$$

$$\leq p_{(z+1)^3}^{(z+1)^3([z,z,z]+1)}$$

Setting $g = \lambda z.p_{(z+1)^3}^{(z+1)^3([z,z,z]+1)}$ we have $g \in \mathcal{PR}$ and we are done by (1):

$$A_n(x) = z \equiv (\exists u)_{\leq g(z)}(Comp(u) \wedge [n, x, z] \in u) \qquad \square$$

Worth saying: If f is total and $y = f(\vec{x})$ is in $\mathcal{PR}_*$, then it does *not* necessarily follow that $f \in \mathcal{PR}$, as 2.4.4.2 exemplifies. On the other hand, if f is total and $y = f(\vec{x})$ is in $\mathcal{R}_*$, then, trivially, $f \in \mathcal{R}$ since $f = \lambda x.(\mu y)(y = f(\vec{x}))$.

What is missing from the preceding expression is a *primitive recursive bound* on the search (μy), and this absence does not allow us to conclude that f is primitive recursive even when its graph is. For example, such a bound is impossible in the Ackermann case as we know from its growth properties. Why is it, qualitatively, possible for a graph $y = f(\vec{x})$ to be "easier" to compute (say, primitive recursively, vs. recursively) than the function itself, at input $\vec{x}$? Because the complexity of the graph is not expressed in terms of $\vec{x}$ only; it is also expressed in terms of y; thus, we do not have to compute the "output" before we compare y and $f(\vec{x})$.

Lest the reader thinks that the foregoing justification is just a matter of creative accounting, it is not: Not all recursive functions f have an "easy" graph $y = f(\vec{x})$, as we will see in the Section 2.7. Functions that grow too fast (like the Ackermann function), are in a certain sense "honest",[81] that is, all the computational effort to compute their output goes toward building a very large output. Thus, e.g., having z already given and being asked to then test for $z = A_n(x)$ is a computation saver. On the other hand, if it takes an enormous amount of computation time to compute $g(\vec{x})$, but the output of g is always 0 or 1, then we have *no computational benefit* in knowing, say, $y = 0$ when being asked to verify (or reject) $y = g(\vec{x})$.

2.5 SEMI-COMPUTABLE RELATIONS; UNSOLVABILITY

We next define a $\mathcal{P}$-counterpart of $\mathcal{R}_*$ and $\mathcal{PR}_*$ and look into some of its closure properties.

---

[81] This is actually a technical term due to Blum (1967). See also Tourlakis (1984), Ch. 12.

2.5.0.3 Definition. (Semi-computable Relations) A relation $P(\vec{x})$ is called *semi-computable* or *semi-recursive* iff for some $f \in \mathcal{P}$, we have, for all $\vec{x}_n$,

$$P(\vec{x}_n) \equiv f(\vec{x}_n) \downarrow \tag{1}$$

The set of all semi-computable relations is denoted by $\mathcal{P}_*$.

If $f = \phi_a^{(n)}$ in (1) above, then we say that "a is *a semi-computable index* or just a *semi-index* of $P(\vec{x}_n)$". If $n = 1$ (thus $P \subseteq \mathbb{N}$) and a is one of the semi-indices of P, then we write $P = W_a$ [Rogers (1967)]. □

We are making the symbol $\mathcal{P}_*$ up, in complete analogy with the symbols $\mathcal{PR}_*$ and $\mathcal{R}_*$. It is not standard in the literature.

We have at once:

2.5.0.4 Theorem. (Normal Form Theorem for Semi-computable Relations)
$P(\vec{x}_n) \in \mathcal{P}_*$ *iff, for some* $a \in \mathbb{N}$, *we have (for all $\vec{x}_n$)* $P(\vec{x}_n) \equiv (\exists z)T^{(n)}(a, \vec{x}_n, z)$.

Proof. Only if-part. Let $P(\vec{x}_n) \equiv f(\vec{x}_n) \downarrow$, with $f \in \mathcal{P}$. By Remark 2.3.0.12, $f = \phi_a^{(n)}$ for some $a \in \mathbb{N}$. We conclude by Corollary 2.3.0.8(1).

If-part: By 2.5.0.3 and 2.3.0.12, the given equivalence translates into $P(\vec{x}_n) \equiv \phi_a^{(n)}(\vec{x}_n) \downarrow$. But $\phi_a \in \mathcal{P}$. □

Rephrasing the above theorem (hiding the ϕ-index "a", and remembering that $\mathcal{PR}_* \subseteq \mathcal{R}_*$), we have:

2.5.0.5 Corollary. (Strong Projection Theorem) $P(\vec{x}_n) \in \mathcal{P}_*$ *iff, for some recursive predicate $Q(\vec{x}_n, z)$, we have (for all $\vec{x}_n$)* $P(\vec{x}_n) \equiv (\exists z)Q(\vec{x}_n, z)$.

Proof. For the *only if*, take $Q(\vec{x}_n, z)$ to be $\lambda\vec{x}_n z.T^{(n)}(a, \vec{x}_n, z)$ for appropriate $a \in \mathbb{N}$. For the *if*, take $f = \lambda\vec{x}_n.(\mu z)Q(\vec{x}_n, z)$ (cf. 2.1.2.39). Then $f \in \mathcal{P}$ and $P(\vec{x}_n) \equiv f(\vec{x}_n) \downarrow$. □

2.5.0.6 Corollary. $P(\vec{x}_n) \in \mathcal{P}_*$ *iff, for some $\lambda\vec{x}_n.g(\vec{x}_n) \in \mathcal{P}$, we have (for all $\vec{x}_n$)* $P(\vec{x}_n) \equiv g(\vec{x}_n) = 0$.

Proof. The *only if* is immediate from 2.5.0.3: Let $f \in \mathcal{P}$ such that, for all $\vec{x}_n$, $P(\vec{x}_n) \equiv f(\vec{x}_n) \downarrow$. Take $g = \lambda\vec{x}_n.1 \dot{-} (1 \dot{-} f(\vec{x}_n))$.

For the *if*, note that $f = \phi_i^{(n)}$ for some i, thus

$$f(\vec{x}_n) = 0 \equiv (\exists z)\left(T^{(n)}(i, \vec{x}_n, z) \wedge d(z) = 0\right)$$

We are done by 2.5.0.5. □

The preceding corollary is the analogue of 2.1.2.16 and 2.1.2.17 for $\mathcal{P}_*$. It provides, among other things, easy proofs for the facts $\mathcal{PR}_* \subseteq \mathcal{P}_*$ and $\mathcal{R}_* \subseteq \mathcal{P}_*$. For example, say, $Q(\vec{x}) \in \mathcal{PR}_*$. Then, for some $g \in \mathcal{PR}$, $Q(\vec{x}) \equiv g(\vec{x}) = 0$, for all $\vec{x}$. But $g \in \mathcal{P}$ as well, and we can invoke 2.5.0.6.

2.5.0.7 Corollary. (Graphs of Partial Recursive Functions) $\lambda \vec{x}_n . f(\vec{x}_n) \in \mathcal{P}$ *iff* $y = f(\vec{x}_n)$ *is semi-recursive.*

Proof. For the *only if*, let $f = \phi_i^{(n)}$. Then

$$y = f(\vec{x}_n) \equiv (\exists z)\big(T^{(n)}(i, \vec{x}_n, z) \land d(z) = y\big)$$

We conclude by 2.5.0.5. For the if part, let (again, 2.5.0.5)

$$y = f(\vec{x}_n) \equiv (\exists z)Q(z, \vec{x}_n, y)$$

To compute $f(\vec{x}_n)$—given $\vec{x}_n$—we enumerate all pairs $\langle z, y \rangle$ and stop at the "first", if any, that satisfies $Q(z, \vec{x}_n, y)$; we output y. Mathematically,

$$f(\vec{x}_n) = \Big(\!(\mu w)Q\big((w)_0, \vec{x}_n, (w)_1\big)\!\Big)_1$$

Clearly $f \in \mathcal{P}$.   □

   Pause. Why not argue the *if* part more simply, in view of 2.5.0.6? Let $g \in \mathcal{P}$ such that
$$y = f(\vec{x}_n) \equiv g(y, \vec{x}_n) = 0$$
Then $f(\vec{x}_n) = (\mu y)g(y, \vec{x}_n)$, for all $\vec{x}_n$, and thus $f \in \mathcal{P}$.◄

   See Exercise 2.12.33.

2.5.0.8 Remark. (Deciders and Verifiers) A computable relation $P(\vec{x}_n)$ is, by definition, one for which $\chi_P \in \mathcal{R}$; thus it has an associated URM M that *decides membership* of any $\vec{a}_n$ in P *both ways*: "yes" (output 0) if it is in; "no" (output 1) if it is not. Thus this M is a *decider* for $P(\vec{x}_n)$.

   A semi-computable relation $Q(\vec{x}_m)$, on the other hand, comes equipped only with a *verifier*, i.e., a URM N that verifies $\vec{a}_m \in Q$, *if true*, by virtue of halting on input $\vec{a}_m$. A verifier gives no tangible information about the non membership cases, which cause it to enter a so-called "infinite loop" (it enters a *non terminating* computation).

   While, *mathematically* speaking, $\vec{a}_m \notin Q$ is also "verified" by virtue of *looping forever* on input $\vec{a}_m$, *algorithmically speaking* this is *no verification at all* as we do *not* have a way of knowing whether N is looping forever as opposed to simply being awfully slow, planning perhaps to halt in a couple of trillion years (cf. *halting problem* 2.5.0.16).

   In the algorithmic sense, a verifier (of a semi-computable set of m-tuples) *verifies only* the "yes" instances of questions such as "Is $\vec{a}_m \in Q$?"—hence its name.   □

Thus, the output of a verifier for a semi-computable relation $Q(\vec{x})$, when it halts, is irrelevant. It has verified membership of its input to Q *simply by virtue of terminating its computation.*

2.5.0.9 Definition. (Undecidable Problems) A *problem* is a *question* of the form $\vec{x} \in Q$. Synonymously, a question of the form $Q(\vec{x})$.
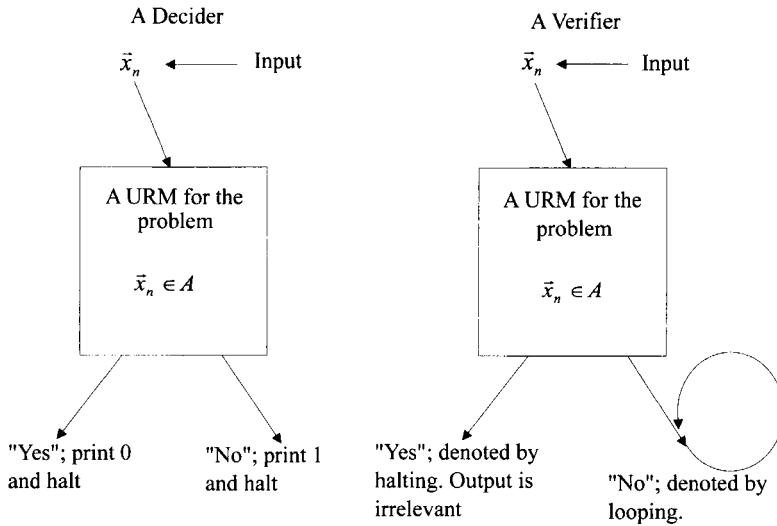
Figure 2.1 A decider and a verifier, pictorially, for handling the query, "$\vec{x}_n \in A$?", by a URM.

Thus, *a problem is a predicate.*

We say that a problem $Q(\vec{x})$ is *decidable* or (recursively)[82] *solvable*, iff there is a *decider* for it, which mathematically is expressed by "$Q(\vec{x}) \in \mathcal{R}_*$"—i.e., Q is recursive. In the opposite case we say that $Q(\vec{x})$ is *undecidable* or (recursively) *unsolvable*.

A problem $Q(\vec{x})$ is *semi-decidable* iff there is a a verifier for it, that is, iff $Q(\vec{x})$ is semi-computable. □

Intuitively, we see that if we have a verifier for a relation $Q(\vec{x}_n)$ *and* also have a verifier for its *complement* (negation) $\neg Q(\vec{x}_n)$, then we can build a decider for $Q(\vec{x}_n)$: On input $\vec{a}_n$ we simply run *both* verifiers simultaneously. If the one for Q halts, then we print 0 and stop the computation. If, on the other hand, the one for $\neg Q$ halts, then we print 1 and stop. Of course, one or the other *will* halt, since one of $Q(\vec{a}_n)$ or $\neg Q(\vec{a}_n)$ *is* true!

This process computes $\chi_Q(\vec{a}_n)$. Put more mathematically,

2.5.0.10 Proposition. *If $Q(\vec{x}_n)$ and $\neg Q(\vec{x}_n)$ are in $\mathcal{P}_*$, then both are in $\mathcal{R}_*$.*

Proof. Let i and j be semi-indices of Q and $\neg Q$ respectively, that is (2.5.0.4),

$$Q(\vec{x}_n) \equiv (\exists z)T^{(n)}(i, \vec{x}_n, z)$$
$$\neg Q(\vec{x}_n) \equiv (\exists z)T^{(n)}(j, \vec{x}_n, z)$$

[82]The parenthetical qualifier is usually omitted.

Define

$$g = \lambda \vec{x}_n.(\mu z)\big(T^{(n)}(i, \vec{x}_n, z) \vee T^{(n)}(j, \vec{x}_n, z)\big)$$

Intuitively, g implements (mathematically) the process in which we run the two verifiers simultaneously, (coded as) i and j, and look for one that halts, by looking for the smallest z that codes a computation[83] of i or j as the case may be.

Trivially, $g \in \mathcal{P}$. Hence, $g \in \mathcal{R}$, since it is total (why?). We are done by noticing that $Q(\vec{x}_n) \equiv T^{(n)}(i, \vec{x}_n, g(\vec{x}_n))$—cf. 2.1.2.24. By closure properties of $\mathcal{R}_*$ (2.1.2.21), $\neg Q(\vec{x}_n)$ is in $\mathcal{R}_*$, too. $\qquad\square$

2.5.0.11 Proposition. $\mathcal{R}_* \subseteq \mathcal{P}_*$.

Proof. Let $Q(\vec{x}) \in \mathcal{R}_*$ and y be a new variable (other than any of the $\vec{x}$). Since Q does not depend on y, we have $Q(\vec{x}) \equiv (\exists y)Q(\vec{x})$. By 2.5.0.5, $Q(\vec{x}) \in \mathcal{P}_*$. $\qquad\square$

2.5.0.12 Remark. An intuitive way to see the truth of the preceding proposition is this: Given a URM M that decides $Q(\vec{x})$, that is, computes χ_Q, modify it to compute instead

$$g = \lambda \vec{x}.\begin{cases} 0 & \text{if } \chi_Q(\vec{x}) = 0 \\ \uparrow & \text{i.e., "get into an infinite loop", otherwise} \end{cases}$$

How do we do this? Easy: Using M as a subprogram, we build a new URM, G, for g that

(1) Given the input $\vec{x}$, G first computes $\chi_Q(\vec{x})$—using M.

(2) If the computed answer is 0, then it halts (jumps to the **stop** instruction of G). If the answer is 1 then it enters an infinite loop.

How do we enter an infinite loop *deliberately*? Well, say that the output variable of M and G is Y. We just need the following program "logic" (as programmers say):

$$\vdots$$
Comment. Above $k - 1$ we have just M, with its **stop** removed
$k - 1 :$ **if** $Y = 0$ **goto** k **else goto** $k - 1$
$k :$ **stop**

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

2.5.0.13 Remark. (Undecidable Problems and Uncomputable Functions Exist)
We can readily show, albeit in a somewhat intangible manner, that *undecidable problems* and therefore *uncomputable total functions* (their characteristic functions) exist.

This readily follows from a so-called *cardinality argument*: By Kleene's Normal Form theorem, we have only a countable set of partial recursive functions

$$\{\phi_i : i \in \mathbb{N}\} \tag{1}$$

---

[83] A terminating one; cf. remark following the proof of 2.3.0.8.

Thus the *subset* (recall 1.3.0.44) of total (computable) 0-1-valued functions (and hence, decidable problems) is countable. However, by 1.3.0.50, the set of all total functions $f : \mathbb{N} \to \{0, 1\}$ is uncountable. So there must be many such functions that do not belong to the enumeration (1)! Each such function f not only provides an example of an uncomputable function, but being 0-1-valued provides an example of an undecidable problem, this one: $f(x) = 0$.

We called this an "intangible demonstration" of the existence of undecidable problems as it produced no specific meaningful problem that is undecidable. We remedy this below. □

2.5.0.14 Definition. (The Halting Problem) The *halting problem* has central significance in computability. It is the question whether "*program x will ever halt if it starts computing on input x*". That is, if we set $K = \{x : \phi_x(x) \downarrow\}$, then the halting problem is $x \in K$. We denote the complement of K by $\overline{K}$. □

2.5.0.15 Exercise. The halting problem $x \in K$ is semi-recursive.

Hint. The problem is "$\phi_x(x) \downarrow$". Now invoke the normal form theorem (2.3.0.8(1)). □

2.5.0.16 Theorem. (Unsolvability of the Halting Problem) *The halting problem is undecidable.*

Proof. In view of the preceding exercise (and 2.5.0.10), it suffices to show that $\overline{K}$ is not semi-computable. Suppose instead that i is a semi-index of this set. Thus, $x \in \overline{K} \equiv (\exists z)T(i, x, z)$, or, making the part $x \in \overline{K}$—that is, $\phi_x(x) \uparrow$—explicit:

$$\neg(\exists z)T(x, x, z) \equiv (\exists z)T(i, x, z) \tag{1}$$

Substituting i into x in (1) we get a contradiction. □

2.5.0.17 Remark. (1) By 2.5.0.3 a set $S \subseteq \mathbb{N}$ is semi-recursive iff "it is a W_i", that is, for some i, $S = W_i$. The above proof says that "$\overline{K}$ is not a W_i". Is this surprising? Well, no!

This goes back to the Cantor diagonalization that shows that $D \ (\subseteq \mathbb{N})$, below,

$$D = \{x : x \notin S_x\}$$

is not an S_i (cf. 1.3.0.52), where each S_i is a subset of $\mathbb{N}$. Indeed,

$$x \in W_i \overset{2.5.0.3}{\equiv} \phi_i(x) \downarrow \equiv (\exists y)T(i, x, y)$$

hence $x \notin W_i \equiv \neg(\exists y)T(i, x, y)$ and, in particular, $i \notin W_i \equiv \neg(\exists y)T(i, i, y)$. But the right hand side says "$\phi_i(i) \uparrow$", that is, $i \in \overline{K}$. Thus

$$\overline{K} = \{x : x \notin W_x\}$$

and Cantor's diagonalization argument shows that "$\overline{K}$ is not a W_i". *So the proof of 2.5.0.16 was a well-concealed diagonalization argument!*

(2) Since $K \in \mathcal{P}_*$, we conclude that the inclusion $\mathcal{R}_* \subseteq \mathcal{P}_*$ (2.5.0.11) is proper, i.e., $\mathcal{R}_* \subset \mathcal{P}_*$.

(3) The characteristic function of K provides an *example* of a *total* uncomputable function.

(4) In 2.1.2.14 we saw an example of how to remove "points of non definition" from a function so that it *remains computable* after it has been extended to a total function. Can we *always* do that?

No. For example, the function $f = \lambda x.\phi_x(x) + 1$ *cannot* be extended to a *total* computable function. Of course, by 2.3.0.8, $f \in \mathcal{P}$, since, for all x, $f(x) \simeq d\big((\mu y)T(x, x, y)\big) + 1$. Here is why: Suppose that $g \in \mathcal{R}$ extends f. Thus, $g = \phi_i$ for some i. Let us look at $g(i)$: We have

$$g(i) \underset{\text{by } g = \phi_i}{=} \phi_i(i) \underset{\text{both sides defined}}{\neq} \phi_i(i) + 1 \underset{\text{def. of } f}{=} f(i) \; ^{84}$$

But since $f(i) \downarrow$, we also have $g(i) = f(i)$ as g extends f, a contradiction.   □

Once we have built a class of functions or predicates, we next look at their closure properties.

2.5.0.18 Theorem. (Closure Properties of $\mathcal{P}_*$) $\mathcal{P}_*$ *is closed under* $\vee$, $\wedge$, $(\exists y)_{<z}$, $(\exists y)$, *and* $(\forall y)_{<z}$. *It is* not *closed under either* $\neg$ *or* $(\forall y)$.

Proof. Given semi-computable relations $P(\vec{x}_n)$, $Q(\vec{y}_m)$, and $R(y, \vec{u}_k)$ of semi-indices p, q, r, respectively. In each case we will express the relation we want to prove semi-computable as a strong projection (2.5.0.5):

$\vee$

$$P(\vec{x}_n) \vee Q(\vec{y}_m) \equiv (\exists z)T^{(n)}(p, \vec{x}_n, z) \vee (\exists z)T^{(m)}(q, \vec{y}_m, z)$$
$$\equiv (\exists z)\big(T^{(n)}(p, \vec{x}_n, z) \vee T^{(m)}(q, \vec{y}_m, z)\big)$$

$\wedge$

$$P(\vec{x}_n) \wedge Q(\vec{y}_m) \equiv (\exists z)T^{(n)}(p, \vec{x}_n, z) \wedge (\exists z)T^{(m)}(q, \vec{y}_m, z)$$
$$\equiv (\exists w)\big((\exists z)_{<w}T^{(n)}(p, \vec{x}_n, z) \wedge (\exists z)_{<w}T^{(m)}(q, \vec{y}_m, z)\big)$$

$(\exists y)_{<z}$

$$(\exists y)_{<z}R(y, \vec{u}_k) \equiv (\exists y)_{<z}(\exists w)T^{(k+1)}(r, y, \vec{u}_k, w)$$
$$\equiv (\exists w)(\exists y)_{<z}T^{(k+1)}(r, y, \vec{u}_k, w)$$

$(\exists y)$

$$(\exists y)R(y, \vec{u}_k) \equiv (\exists y)(\exists w)T^{(k+1)}(r, y, \vec{u}_k, w)$$
$$\equiv (\exists z)(\exists y)_{<z}(\exists w)_{<z}T^{(k+1)}(r, y, \vec{u}_k, w)$$

---

[84] We have used $=$ rather than $\simeq$ throughout, since all expressions used in this display are defined.

$(\forall y)_{<z}$

$$(\forall y)_{<z}R(y, \vec{u}_k) \equiv (\forall y)_{<z}(\exists w)T^{(k+1)}(r, y, \vec{u}_k, w)$$
$$\equiv (\exists v)(\forall y)_{<z}(\exists w)_{<v}T^{(k+1)}(r, y, \vec{u}_k, w)$$

As for possible closure under $\neg$ and $\forall y$, K provides a counterexample to $\neg$: $K \in \mathcal{P}_*$ (2.5.0.15) but $\overline{K} \notin \mathcal{P}_*$ (2.5.0.16). Closure under $\forall y$ is also untenable as $\neg T(x, x, y)$ provides a counterexample: Being primitive recursive, it is in $\mathcal{P}_*$. However, $(\forall y)\neg T(x, x, y)$ is not, since this is $\neg(\exists y)T(x, x, y)$—that is, $x \in \overline{K}$. $\quad\square$

2.5.0.19 Remark. The case for $\vee$ in the proof above is straightforward since $\exists$ distributes with $\vee$—$(\exists x)(\mathscr{A} \vee \mathscr{B}) \equiv (\exists x)\mathscr{A} \vee (\exists x)\mathscr{B}$.

However, it does not distribute with $\wedge$, hence the complication in the $\wedge$ case. We want to say that $P(\vec{x}_n) \wedge Q(\vec{y}_m)$ is true precisely when two independent computations z_1 and z_2 exist for the machines (coded by) p and q, if the inputs are $\vec{x}_n$ and $\vec{y}_m$, respectively.

One way to say this is just as we did, by saying that there is a number w, bigger than both of z_1 and z_2—for example $w = \max(z_1, z_2) + 1$ will do—and then use "$(\exists z)_{<w}$" in each case.

Pause. Why not forget about these acrobatics and just rest the case with the first line of the $\wedge$ case above?◄

But surely, *because* we want to express the left hand side as a formula of the form $(\exists w)S(w, \ldots)$—in order to invoke 2.5.0.5—that is, a formula *with a single existential quantifier up in front* and with a recursive S [confirmed due to bounded quantifications, "$(\exists z)_{<w}$", being employed].

An alternative way of saying "I have two computations z_1 and z_2" is to use coding:

$$P(\vec{x}_n) \wedge Q(\vec{y}_m) \equiv (\exists z)\big(T^{(n)}(p, \vec{x}_n, (z)_0) \wedge T^{(m)}(q, \vec{y}_m, (z)_1)\big)$$

For the $(\exists y)_{<z}$ case we used the commutativity betwee the two $\exists$.

Pause. But one is bounded! Can it commute with an unbounded one? (See Exercise 2.12.26)◄

In the double-$\exists$ case we reduce the two quantifiers to one, as needed for a form $(\exists w)S(w, \ldots)$ with recursive S, by using a $z > y, w$ (e.g., $z = \max(y, w) + 1$ will do). Alternatively, we can use $z = [y, w]$ and thus say

$$(\exists y)R(y, \vec{u}_k) \equiv (\exists z)T^{(k+1)}(r, (z)_0, \vec{u}_k, (z)_1)$$

Regarding the $(\forall y)_{<z}$ case, commuting $\exists$ and $\forall$ is not on.

Pause. Think of $(\forall x)(\exists y)x < y$ vs. $(\exists y)(\forall x)x < y$.◄

Yet, we wanted to bring

$$(\forall y)_{<z}(\exists w)T^{(k+1)}(r, y, \vec{u}_k, w) \tag{1}$$

in the form $(\exists w)S(w, \ldots)$ with a recursive S. Thus, we argued (implicitly) as follows: (1) says that for every $y = 0, 1, \ldots, z - 1$ there is a w-value—possibly dependent on

the y—that is, there is a sequence of numbers $w_0, w_1, \ldots, w_{z-1}$ that make the predicate $T^{(k+1)}(r, i, \vec{u}_k, w_i)$ true. If we set $u = \max\{w_0, w_1, \ldots, w_{z-1}\} + 1$ then we can capture this observation by making the quantifier prefix of (1) "$(\exists u)(\forall y)_{<z}(\exists w)_{<u}$". Incidentally, this expresses (1) in the form required: $(\exists w)S(w, \ldots)$ with a recursive S. This is precisely what we did in our proof above.

An alternative technique, using coding, is often encountered in the literature: (1) is equivalent to

$$(\exists u)\Big(Seq(u) \wedge (\forall y)_{<z} T^{(k+1)}(r, y, \vec{u}_k, (u)_y)\Big)$$

which has the proper form. □

2.5.0.20 Proposition. *If $\lambda\vec{x}.f(\vec{x}) \in \mathcal{P}$ and $Q(z, \vec{y}) \in \mathcal{P}_*$, then $Q(f(\vec{x}), \vec{y}) \in \mathcal{P}_*$.*

Proof. By 1.2.0.10,

$$Q(f(\vec{x}), \vec{y}) \equiv (\exists z)\Big(z = f(\vec{x}) \wedge Q(z, \vec{y})\Big)$$

By 2.5.0.7 and 2.5.0.18, the right hand side, and hence the left hand side, of $\equiv$ is semi-recursive. □

For a more direct proof, see Exercise 2.12.32.

2.5.0.21 Example. This is our first example of a *reduction argument*, a trivial one. We introduce a generalization, K_0, of the halting set K, by

$$K_0 \overset{\text{Def}}{=} \{\langle x, y\rangle : \phi_x(y) \downarrow\}$$

We show that the problem $\langle x, y\rangle \in K_0$ is undecidable, that is,

$$K_0 \notin \mathcal{R}_* \tag{1}$$

Suppose that (1) is false. Then the characteristic function, $\lambda xy.\chi_{K_0}(x, y)$ of K_0 is in $\mathcal{R}$. But then so is the function $f = \lambda x.\chi_{K_0}(x, x)$ obtained from χ_{K_0} by identification of variables (cf. 2.1.2.6). However, f is the characteristic function of the halting set, and we just have shown that the halting problem is decidable!

This contradiction shows that (1) is correct, after all.

We have just witnessed an instance of an argument that went like this: *If I have an algorithm that solves problem B,*[85] *then I know how to build another algorithm that uses the one for B and solves problem A.*[86]

That is, we *reduced problem A to problem B* (this makes A "more decidable" than B; and makes B "more undecidable" than A).

This reduction shows that *if we know that A is undecidable, then so must be B*.

---

[85]Here $\langle x, y\rangle \in K_0$.
[86]Here $x \in K$.

We will encounter many more reduction arguments in Section 2.7. $\qquad$ □

2.5.0.22 Example. (A Very Hard Problem) The *equivalence problem* is: *given two programs, decide if they compute the same function or not.*

A "program" here can be any *finite way* of describing a function. This finite way could be an actual program, such as a URM or a loop program. Or it could be a derivation (cf. 2.1.2.1 and Corollary 2.3.0.11), say, within $\mathcal{PR}$ or $\mathcal{P}$, which defines a function.

To fix ideas, let us focus attention on primitive recursive functions, *finitely defined via loop programs*. We ask: Is the problem of determining whether *two such functions* are equal decidable?

Well, if it is, then in particular so will be the special case of determining whether $\lambda y.1$ and $\lambda y.\chi_T(x, x, y)$—where T is the Kleene predicate—are the same function or not, for any given x. The reader may readily imagine—due to the primitive recursiveness of both functions—that they are given by loop programs.

The question, mathematically, is $(\forall y)(1 = \chi_T(x, x, y))$. In terms of T this says $(\forall y)\neg T(x, x, y)$, or $\neg(\exists y)T(x, x, y)$.

We recognize the last expression as $x \in \overline{K}$, which we know that is *not semi-computable* (2.5.0.16), let alone recursive!

Pause. Why "let alone"?◄

Thus the equivalence problem of primitive recursive functions is incredibly hard: There is not even a verifier for it! $\qquad$ □

2.5.0.23 Remark. (Computably Enumerable Sets) There is an interesting characterization of *non-empty* semi-computable sets that is found in all introductions to the theory of computation. These sets are precisely those that can be "enumerated effectively" or "computably", that is, we can prove that

A non-empty set $S \subseteq \mathbb{N}$ is semi-computable iff it is the range *(cf. Definition 1.2.0.5) of some $f \in \mathcal{PR}$.*

The enumeration is *not* required to be 1-1, so there may be repetitions. Notice that since the enumerating function is total, there will *necessarily* be repetitions in the case when S is finite.

What is the intuition for this? Well,

(1) Assume first that we have an algorithmic enumeration of all the members of S. Here is then how to verify (semi-decide) the question $x \in S$: Given x, start the algorithmic enumeration and keep an eye on what it "prints". If and when x is printed, then stop. We have verified $x \in S$. What if $x \notin S$? Well, then it will never be printed by the enumeration and we will never stop our process.

(2) Conversely, assume that we have a verifier, M (a URM), for $x \in S$. We write a new program N that behaves as follows: It *systematically generates* all pairs of numbers $\langle x, y \rangle$, one at a time.

For example, one can enumerate all numbers

$$0, 1, 2, 3, \ldots, z, \ldots$$

in turn, and, for each z generated, one can generate the pair $\langle (z)_0, (z)_1 \rangle$.

For each pair generated, N checks whether M, on input x, *halts within y computation steps.*[87] If so, x is printed (as it clearly belongs to S).

Pause. Why make it so complicated and not instead enumerate all numbers x in turn, and if M halts on x, then print x?◄

The technique in (2) above is called *dovetailing* several computations (of M) *for several inputs* "at once". Well, strictly speaking, not "at once". The method implements, indeed *sequentially* simulates, a "poor person's *parallelism*", because, in essence, it simulates the *in parallel* examination of several questions of the type "does M halt on x?".

The essential feature of parallelism is not the temporal simultaneity of testing the questions "does M halt on x?" for various x, but rather the fact that if an input $x = a$ causes M to run forever, this *does not affect, nor block, the testing of other inputs for which M halts*. A true parallel "environment" allocates one process to each input x. On the other hand, dovetailing captures this *key property of parallelism* and does so with *a single computation process* (or "single processor")!

The simulation of parallelism is effected by allowing to each question gradually more and more *time* (number of steps) to reach an answer. Notice that since there are infinitely many pairs $\langle a, y \rangle$ with first component a, if M ever halts on a—say, using $y = b$ computation steps—then this fact will be eventually verified in the process (2): It will happen precisely when we will be testing the pair $\langle a, b \rangle$.

An intuitively more immediate *rearrangement of the dovetailing process* of (2), which demonstrates the sense in which dovetailing is approaching true parallelism "in the limit", is captured by the matrix below:

$$0; 1$$
$$0,1; 2$$
$$0,1,2; 3$$
$$0,1,2,3; 4$$
$$\vdots$$
$$0,1,2,3,\ldots, i; i+1$$
$$\vdots$$

The number at the far right in each row is the number of steps that we let M run. The other numbers in each row are the inputs we test *for said number of steps*. In the "limit", it is as if we are testing all inputs "simultaneously": input 0 for one step; inputs 0 and 1 for two steps; inputs 0, 1 and 2 for three steps; $\ldots$, inputs $0, 1, \ldots, i$ for $i + 1$ steps; and so on. □

---

[87]Think of a "step" as the passage from one ID to the next.

Mathematically, we repeat the above informal argument, (1) and (2), in 2.5.0.24 below to prove the italicized statement at the beginning of the previous remark. Central in our preceding discussion was the concept of "step". But what *is* a step mathematically? We take as "step" to be the entire computation, coded as y in the Kleene predicate $T(i, x, y)$. That is, for any ϕ_i, its *step-counting* or *complexity function* is

$$\Phi_i = \lambda x.(\mu y)T(i, x, y) \qquad (*)$$

This is reasonable, since the computation y is a strictly increasing function of how many ID-to-ID "real steps" took place in the (terminating) computation.

In fact, Blum (1967) takes as the key, indeed *defining*, properties of the concept of complexity of ϕ_i the following two:

(I) $\phi_i(x) \downarrow$ iff $\Phi_i(x) \downarrow$; that is, the program i halts on input x iff a complexity of computation can be assigned for said input.

(II) $\Phi_i(x) \leq y$ is *recursive*; that is, we can *decide* whether machine i halts within y (i.e., in $\leq y$) "steps".

Blum (1967) takes (I)–(II) as the *axioms* for complexity theory, that is, without specifying Φ explicitly. Many concrete choices of Φ that satisfy the axioms are possible. By the way, for our chosen Φ in $(*)$, (I) is trivially obtained directly from 2.3.0.8. As for (II), $\Phi_i(x) \leq y \equiv (\exists z)_{\leq y} T(i, x, z)$ which is more than recursive: primitive recursive.

On the other hand $y < \Phi_i(x) \equiv \neg \Phi_i(y) \leq y$; also in $\mathcal{PR}_*$.

It is important to observe that we bypass (I), above, when we assess

$$\Phi_i(x) \overset{\leq}{\underset{\geq}{=}} y$$

We do *not* compute $\Phi_i(x)$ (which may diverge!) to figure out the answer.

2.5.0.24 Theorem. *A non-empty set $S \subseteq \mathbb{N}$ is semi-computable iff it is the* range *of some $f \in \mathcal{PR}$.*

Proof. For the part (1), let f be primitive recursive such that $\mathrm{ran}(f) = S$. That is,

$$y \in S \equiv (\exists x)f(x) = y$$

Given that $f(x) = y$ is in $\mathcal{PR}_*$ (2.1.2.25), $y \in S$ is semi-computable by the projection theorem (2.5.0.5).

For the dovetailing part, (2) of 2.5.0.23, assume that the non-empty S is semi-computable. Let i be a semi-index for S, thus,

$$x \in S \equiv (\exists y)T(i, x, y) \qquad (*)$$

for all x. Following directly on the idea in (2), with the concept of "step" made mathematically precise in the preceding remarks, we define the enumerating function by:

$$f(z) = \begin{cases} (z)_0 & \text{if } T(i, (z)_0, (z)_1) \\ a & \text{otherwise} \end{cases}$$

where "a" is some fixed member of S that we keep outputting every time the condition "$T(i, (z)_0, (z)_1)$" fails,[88] ensuring that f is total. Of course f is primitive recursive.

Is it true that $\mathrm{ran}(f) = S$?

Indeed, as $\mathrm{ran}(f)$ contains only numbers of the form $(z)_0$ such that $T(i, (z)_0, (z)_1)$ holds, it is immediate by $(*)$ that $\mathrm{ran}(f) \subseteq S$. Conversely, let $x \in S$ and let b be a value of y that makes $(*)$ true. But then $f([x, b]) = x$, so $x \in \mathrm{ran}(f)$.   □

The above result justifies the following nomenclature:

2.5.0.25 Definition. A set $S \subseteq \mathbb{N}$ is called *computably enumerable* (c.e.) or *recursively enumerable* (r.e.) iff it is either empty, or is the range of a primitive recursive function.   □

There is no loss of generality in presenting the above definition for subsets of $\mathbb{N}$ since via coding [. . .] it can be trivially and naturally extended to sets of n-tuples for $n > 1$. A set $S \subseteq \mathbb{N}^n$ is c.e. iff there is a primitive recursive f such that $\mathrm{ran}(f) = \{[\vec{x}] : S(\vec{x})\}$.

2.5.0.26 Corollary. *A non-empty set $S \subseteq \mathbb{N}$ is semi-recursive iff it is c.e. (r.e.)*

2.5.0.27 Corollary. *A non-empty set $S \subseteq \mathbb{N}^n$ is semi-recursive iff it is c.e. (r.e.)*

Proof. The *if* is straightforward, while the *only if* is a direct adaptation of the proof of 2.5.0.24: Let

$$\vec{x}_n \in S \equiv (\exists y) T^{(n)}(i, \vec{x}_n, y) \qquad (**)$$

for all x. The enumerator f is given by

$$f(z) = \begin{cases} [(z)_0, \dots, (z)_{n-1}] & \text{if } T^{(n)}(i, (z)_0, \dots, (z)_{n-1}, (z)_n) \\ [\vec{a}_n] & \text{otherwise} \end{cases}$$

where "$\vec{a}_n$" is some fixed member of S.   □

2.5.0.28 Corollary. *A set $S \subseteq \mathbb{N}^n$ is semi-recursive iff it the range of an $f \in \mathcal{P}$.*

Proof. The *only if* is proved as above, where we just drop the "$[\vec{a}_n]$ otherwise". For the *if*, suppose that

$$y \in S \equiv (\exists x) f(x) = y$$

---

[88]Condition failed: Either because we did not let the computation $\phi_i(x)$ to go on long enough, or no terminating computation exists.

By 2.5.0.7 and 2.5.0.18, the above yields $S \in \mathcal{P}_*$. $\square$

2.5.0.29 Example. (Another Very Hard Problem) The set $\mathscr{R} = \{x : \phi_x \in \mathcal{R}\}$—which trivially is the same as $\{x : \phi_x$ is total$\}$; cf. 2.1.1.2—is very important in computability. One certainly wants to know whether or not we can "tell" if a program x computes a total function. We can tell in one of two ways: We can fully (algorithmically) *decide* the question $x \in \mathscr{R}$, or we can just *verify* it when true. Which one is it here?

Neither. $\mathscr{R}$ is not semi-recursive, hence nor is it recursive (2.5.0.11). In that sense this is another very hard—and very meaningful—problem of which we cannot even verify the positive instances.

We prove the non semi-recursiveness by proving that $\mathscr{R}$ is not c.e. using diagonalization (cf. Subsection 2.2.2 and 1.3.0.50). So, by way of contradiction, let $f \in \mathcal{PR}$ be such that $\mathscr{R} = \text{ran}(f)$. This means that $\{\phi_{f(x)} : x \in \mathbb{N}\}$ is the set of all total computable functions of one variable. Consider the function $d = \lambda x.\phi_{f(x)}(x) + 1$.

By the preceding remark, and composition with the successor function, $d \in \mathcal{R}$. Thus, for some i,

$$d = \phi_{f(i)} \tag{1}$$

since $\mathscr{R}$ is the set of *all* programs that compute total 1-argument functions, thus a program m for d must be an $f(i)$.

What do we know of $\phi_{f(i)}(i)$? Well,

$$\phi_{f(i)}(i) \overset{\text{By (1)}}{=} d(i) \overset{\text{By Def. of } d}{=} \phi_{f(i)}(i) + 1$$

A contradiction, since all sides of = are defined. So no such f exists, and $\mathscr{R}$ is not c.e. $\square$

The above example concludes a discussion we have started in the ⚕-enclosed passage on p. 147.

> Quite apart from the empirical question of whether or not "Church's Thesis" is correct, we note that *within the formalization of* $\mathcal{P}$, in which the functions ϕ_z of $\mathcal{R}$ have *finite descriptions* $z \in \mathbb{N}$, *it is impossible to have an enumeration of all such descriptions* via a *recursive function* of $\mathcal{R}$.
>
> So, at least, within the $\mathcal{P}$-formalism *we cannot diagonalize out of* $\mathcal{R}$ by an argument like the one given in 2.2.2. This observation will be proved mathematically without invoking any "beliefs".

2.5.0.30 Exercise. (Definition by Positive Cases) Consider a set of *mutually exclusive* relations $R_i(\vec{x})$, $i = 1, \ldots, n$, that is, $R_i(\vec{x}) \wedge R_j(\vec{x})$ is false for each $\vec{x}$ as long as $i \neq j$.

Then we can define a function f *by positive cases* R_i from given functions f_j by the requirement (for all $\vec{x}$) given below:

$$f(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } R_1(\vec{x}) \\ f_2(\vec{x}) & \text{if } R_2(\vec{x}) \\ \dots & \dots \\ f_n(\vec{x}) & \text{if } R_n(\vec{x}) \\ \uparrow & \text{otherwise} \end{cases}$$

Prove that if each f_i is in $\mathcal{P}$ and each of the $R_i(\vec{x})$ is in $\mathcal{P}_*$, then $f \in \mathcal{P}$.

Hint. Use 2.5.0.7 along with closure properties of $\mathcal{P}_*$ relations to examine $y = f(\vec{x})$.   □

A semi-recursive predicate is "positive" having the form $(\exists y)Q(y, \vec{x})$ for some recursive Q (2.5.0.5). It is also known as a Σ_1 predicate.

It is important to note about the last case in the definition:

(1) The *otherwise* condition, is the negation of a *positive* predicate, namely, of the semi-recursive $R_1 \vee \cdots \vee R_n$. A "negative" predicate such as this negation has the form $(\forall y)R(y, \vec{x})$, for some recursive R, since it is the negation of one of the form $(\exists y)Q(y, \vec{x})$, for some recursive Q. Such negative predicates are also called Π_1 predicates.

(2) Note that the "output" in the last case is $\uparrow$. This, intuitively, is as much as is expected in general, given that, for example, the "otherwise" of some positive cases, such as $x \in K$, are not even semi-recursive so that the obvious "program" for the function f will enter into an infinite loop when pondering the condition "otherwise".

This last observation is firmed up mathematically in Exercise 2.12.35 via two examples.

2.6 THE ITERATION THEOREM OF KLEENE

Suppose that i codes a URM program, M, that acts on input variables x and y to compute a function $\lambda xy.f(x, y)$. It is certainly trivial to modify program M to compute $\lambda x.f(x, a)$ instead. In computer programming terms, we replace an instruction such as "read y" by one that says "$y \leftarrow a$".

In URM terms, since the input variables $X11, X111, \dots$ are initialized *before* the computation starts, the way to implement the suggested "decommissioning" of y as an input variable—opting rather to initialize it with the number a explicitly, *first thing* during the computation—is to do the following, assuming x and y of f are mapped to $X11$ and $X111$ of M:

(1) Remove $X111$ from the input variables list, $X11, X111$, and

(2) Modify M into M' by adding the instruction $X111 \leftarrow a$ as the very first instruction.

From the original code, i, a new code (depending on i and a) can be easily calculated. This is the intuition of Kleene's *iteration* or "S-m-n" theorem below.

The mathematical details are as follows.

2.6.0.31 Definition. (Code Concatenation)

$$x * y \stackrel{\text{Def}}{=} x \cdot \Pi_{i < lh(y)} p_{i+lh(x)}^{\exp(i,y)}$$ $\square$

2.6.0.32 Remark. Clearly, $\lambda xy.x * y$ is primitive recursive. The definition's aim is to achieve this—which it clearly does:

$$[a_1, \ldots, a_n] * [b_1, \ldots, b_m] = [a_1, \ldots, a_n, b_1, \ldots, b_m]$$

If $Seq(x)$ or $Seq(y)$ fail, then the result of $x * y$ is irrelevant to us. $\square$

2.6.0.33 Exercise. What is $10 * 5$? $\square$

2.6.0.34 Exercise. What is $1 * z$? $z * 1$? $\square$

2.6.0.35 Definition. (Concatenating URMs) Given two URMs M and N of codes m and n. We denote their *concatenation* by MN and $m \frown n$ in terms of their codes. Note that MN means the superposition of the two URMs, in that order, with the **stop**-instruction removed from M and all the instructions of N adjusted to reflect that the first label of N now is $lh(m)$.

We define $m \frown n$ to be 0 if either of m or n is not a valid URM code. $\square$

2.6.0.36 Lemma. *Let $adj(n,k)$ ("adjust n") be the expression that codes a URM n after k was added to all its instruction numbers (and all if-statements were adjusted to still transfer to the same instructions as before). Let also $adj(n,k) = 0$ if n does not code a URM. Then the function $\lambda nk.adj(n,k)$ is primitive recursive.*

Proof. First let us define a less ambitious function, f, that adjusts one instruction due to adding k to the instruction number:

$$f(z,k) = \begin{cases} 3^k z & \text{if } (\exists L,i,a)_{\leq z}\Big(z = [1,L,i,a] \vee \\ & \qquad\qquad z = [2,L,i] \vee \\ & \qquad\qquad z = [3,L,i] \vee \\ & \qquad\qquad z = [5,L]\Big) \\ 231^k z & \text{if } (\exists L,i,P,Q)_{\leq z} z = [4,L,i,P,Q] \\ 0 & \text{otherwise} \end{cases}$$

Note that $231 = 3 \cdot 7 \cdot 11$. Clearly, $f \in \mathcal{PR}$. Finally,

$$adj(n,k) = \begin{cases} \Pi_{i < lh(n)} p_i^{f((n)_i,k)+1} & \text{if } URM(n) \\ 0 & \text{otherwise} \end{cases}$$

Clearly, adj is primitive recursive. $\square$

2.6.0.37 Lemma. $\lambda mn.m \frown n$ *is primitive recursive.*

Proof.

$$
m \frown n = \begin{cases} \left\lfloor \dfrac{m}{p_{lh(m) \dot- 1}^{\exp(lh(m) \dot- 1, m)}} \right\rfloor * adj(n, lh(m) \dot- 1) & \text{if } URM(m) \wedge URM(n) \\ \\ 0 & \text{otherwise} \end{cases}
$$

The left hand operand of $*$ above represents the removal of the **stop**-instruction of the URM m prior to concatenation. It is immediate from the above and the preceding lemma that $\lambda mn.m \frown n$ is primitive recursive. $\square$

2.6.0.38 Theorem. (Kleene's Iteration or "S-m-n" Theorem) *For each* $m \geq 1$ *and* $n \geq 1$, *there is a primitive recursive function* $\lambda i \vec{y}_n . S_n^m(i, \vec{y}_n)$ *such that, for all* $i, \vec{x}_m, \vec{y}_n$,

$$
\phi_i^{(m+n)}(\vec{x}_m, \vec{y}_n) = \phi_{S_n^m(i,\vec{y}_n)}^{(m)}(\vec{x}_m)
$$

Proof. The construction of S_n^m is guided by the introductory remarks of this section: If i codes a URM M such that

$$
M_{X1}^{X11,\dots,X1^{m+n+1}} = \phi_i^{(m+n)} \tag{1}
$$

then we remove the n variables $X1^{m+2}, \dots, X1^{m+n+1}$ from the *designated input-variable list* and add the instructions below at the top of program M.

$$
X1^{m+2} \leftarrow y_1, X1^{m+3} \leftarrow y_2, \dots, X1^{m+n+1} \leftarrow y_n
$$

To avail ourselves of the tools we have developed in this section, we implement the above plan by concatenating the following URM, N, to the left of M. Note that N is not normalized, but NM is, since M is.

$$
\begin{aligned}
1 : \quad & X1^{m+2} \leftarrow y_1 \\
& \;\;\vdots \\
n : \quad & X1^{m+n+1} \leftarrow y_n \\
n+1 : \;& \textbf{stop}
\end{aligned} \tag{N}
$$

The code for N is a function of $\vec{y}_n$ (recall that m, n are constants) which we will name $init(\vec{y}_n)$. Referring to 2.3.0.2,

$$
init(\vec{y}_n) = p_0^{[1,1,m+2,y_1]+1} p_1^{[1,2,m+3,y_2]+1} \cdots p_{n-1}^{[1,n,m+n+1,y_n]+1} p_n^{[5,n+1]+1} \tag{2}
$$

It is immediate that $\lambda \vec{y}_n . init(\vec{y}_n)$ is primitive recursive. Thus, S_n^m, given below for all $i, \vec{y}_n$, is too by Lemma 2.6.0.37.

$$
S_n^m(i, \vec{y}_n) = init(\vec{y}_n) \frown i \tag{3}
$$

It is important to note that (by 2.6.0.37) if i fails the $URM(i)$ "test", then so does $S_n^m(i, \vec{y}_n)$ (indeed, equals 0; cf. 2.6.0.37) and thus both sides of (1) are (Kleene-) *completely* equal (the empty function is undefined on all inputs). □

2.6.0.39 Remark. (1) It is important to note by inspecting (2) and (3) in the proof above that if $URM(i)$ holds, then S_n^m is strictly increasing with respect to each y_i variable. Of course, if $URM(i)$ fails, then S_n^m returns 0 no matter what the inputs y_i may be.

(2) A note on notation: In S_n^m the *upper* index, m, is a mnemonic tool for how many variables stayed "up" (in the ϕ argument), while the lower index, n, indicates how many variables were moved "down", to be hardwired into the "program" $S_n^m(i, \vec{y}_n)$ as it were.

These considerations led to the nickname of the iteration theorem as the "S-m-n theorem".

(3) *In practice, the S-m-n theorem is applied as follows*: If $\lambda \vec{x}_k y \vec{z}_r . f(\vec{x}_k, y, \vec{z}_r) \in \mathcal{P}$, then there is *a 1-1* $h \in \mathcal{PR}$, such that, for all $\vec{x}_k, y, \vec{z}_r$, we have

$$f(\vec{x}_k, y, \vec{z}_r) \simeq \phi_{h(y)}^{(k+r)}(\vec{x}_k, \vec{z}_r)$$

By the assumption on f and 2.3.0.12, there is an $i \in \mathbb{N}$, such that $f(\vec{x}_k, y, \vec{z}_r) \simeq \phi_i^{k+r+1}(\vec{x}_k, \vec{z}_r, y)$. Note the permutation of variables, where y was moved to the end of the argument list of ϕ_i^{k+r+1}, to align with requirements of the S-m-n theorem. Can we do this? Yes, since we may chose the URM i such that we have mapped the variables $\vec{x}_k, y, \vec{z}_r$ of f to the "formal" variables $X11, \ldots X1^{k+r+2}$ so that y's role is played by $X1^{k+r+2}$.

Pause. Is this the same as saying "use 2.1.2.6(iii) to permute variables so that y is last"?◄

We take $h = \lambda y . S_1^{k+r}(i, y)$. Note that the italicized part, "a 1-1", above is a weakening of the observation (1) above. □

2.7 DIAGONALIZATION REVISITED; UNSOLVABILITY VIA REDUCTIONS

This section further develops the theory of computability and *un*computability by developing tools—in particular, *reducibility*—that are more sophisticated than the ones we encountered so far in this volume, toward discovering undecidable and non c.e. problems. We also demonstrate explicitly that diagonalization is at play in a number of interesting examples.

As we mentioned in the Preface and elsewhere already, the aim of computability is to "formalize" the concept of "algorithm" and then proceed to classify problems as decidable vs. undecidable and verifiable vs. unverifiable.

We continue taking—*by definition*—the term "algorithm" to *mean* URM program, and computable (partial) function to mean a URM-computable function, or equivalently, one that has a $\mathcal{P}$-derivation (cf. 2.3.0.11). Thus *proving* that such and such

a problem $x \in A$ does *not* have an "algorithmic solution", or is not even verifiable, becomes mathematically precise: We need to show that $A \notin \mathcal{R}_*$ or $A \notin \mathcal{P}_*$, respectively.

Church has gone a step further, and observing that all known formalisms of the concept of algorithm were proved to be equivalent (each produced the same computable functions), formulated

> **Church's Thesis.** Any partial function that can be informally demonstrated to be computable by some algorithm, can be mathematically demonstrated to be programmable in any one of the known formalisms (such as Turing machines, Markov algorithms, Post systems, URMs[89]).

Of course, this "thesis" is a belief based on empirical evidence, not a metatheorem. The difficulty (toward theoremhood) lies in the fact that in order to, say, demonstrate mathematically that the concepts of "algorithm" and URM coincide, we must already have a mathematical formulation of *algorithm*!

This is why we said above that we take *by definition* that algorithm means URM. We cannot do better than being arbitrary like this. We already mentioned that while the "Thesis" is widely adopted—indeed, some advanced books such as Rogers (1967) use it to shorten proofs that such and such a function is computable—the adoption is not universal; cf. Kalmár (1957).

This volume will not take the shortcut of relying on Church's Thesis. Whenever we want to prove that *f is computable* we will do so mathematically, invariably using 2.3.0.11 and closure properties of $\mathcal{P}$. Nevertheless, we will often also offer an intuitive argument that establishes the desired computability.

2.7.1   More Diagonalization

We begin the development of the theory by revisiting the proof (2.5.0.16 and 2.5.0.17) of the undecidability of the halting problem.

2.7.1.1 Theorem. (The Undecidability of the Halting Problem; Again) $K \notin \mathcal{R}_*$.

Proof. We will argue by contradiction, so we assume that $K \in \mathcal{R}_*$, that is, the relation $\phi_x(x) \downarrow$ is recursive. We view a one-argument function f as a sequence of values,

$$f(0), f(1), \ldots$$

where (informally), if $f(x) \uparrow$, then we take the symbol "$\uparrow$" as the yielded value.

---

[89] Actually, the URM formalism postdates the formulation of Church's Thesis but is demonstrably equivalent to all the others.

On this understanding we form the infinite matrix below, of which the i-th row represents ϕ_i, for all i.

$$\begin{array}{cccc}
\phi_0(\mathbf{0}) & \phi_0(1) & \phi_0(2) & \ldots \\
\phi_1(0) & \phi_1(\mathbf{1}) & \phi_1(2) & \ldots \\
\vdots & \vdots & \vdots & \\
\phi_i(0) & \phi_i(1) & \phi_i(2) & \ldots \\
\vdots & \vdots & \vdots &
\end{array}$$

We proceed as in 1.3.0.53 to utilize the main diagonal

$$\phi_0(0), \phi_1(1), \ldots, \phi_i(i), \ldots$$

and build a function that *cannot be a row* of the above matrix. We simply change each entry that is $\uparrow$ to $\downarrow$, and vice versa:

$$d(x) = \begin{cases} \downarrow & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases}$$

The above captures the idea, but it is not a well-defined function since we have not said what the output of d is when it is defined. We resolve this "uncertainty" arbitrarily as follows:

$$d(x) = \begin{cases} 42 & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases} \tag{1}$$

Indeed, d does not match any row above, as it differs from each row in the spot where it intersects the diagonal. Why do we care? Well, since $\mathcal{P}$ is closed under definition by cases (Exercise 2.12.28), and since by assumption both $\phi_x(x) \downarrow$ and $\phi_x(x) \uparrow$ are recursive,

Pause. Why "both"?◄

it follows that $d \in \mathcal{P}$, i.e., $d = \phi_i$ for some i—i.e., d *must* be some row in the above matrix. We have a contradiction. □

An intuitive reason as to why the function d as defined in (1) is computable, is presented here by outlining a pseudo algorithm for the computation of $d(x)$: Let M be a URM that decides the predicate $\phi_x(x) \downarrow$. Given input x, run M on x. If it says "no", then print 42 and halt; if it says "yes", then get into a deliberate infinite loop (cf. 2.5.0.12).

Worth repeating. We chose d so that at input x it *differs* from $\phi_x(x)$, and thus it differs from ϕ_x; full stop. We have cancelled x as a possible ϕ-index of d (cf. 1.3.0.51).

Given that we have done this for *all* x, we have cancelled all possible ϕ-indices of d. Thus d is not computable. Since our assumption about $\phi_x(x) \downarrow$ also forced d to *be* computable, we managed to reject said assumption as it forced a contradiction.

A version of unbounded search is the following:

2.7.1.2 Definition. (Alternate Unbounded Search Operator) For any *total* function $\lambda y \vec{x}.g(y, \vec{x})$ the expression $(\widetilde{\mu}y)g(y, \vec{x})$ stands for

$$\begin{cases} \min\{y : g(y, \vec{x}) = 0\} & \text{if the minimum exists} \\ \uparrow & \text{otherwise} \end{cases}$$

$\square$

It is immediate that (μy) (2.1.1.18) and $(\widetilde{\mu}y)$ coincide on total functions since—in $g(y, \vec{x}) = 0 \wedge (\forall z)_{<y}(g(z, \vec{x}) \downarrow)$—the subformula $(\forall z)_{<y}(g(z, \vec{x}) \downarrow)$ is true for such g and therefore its presence or absence in the formula is immaterial.

2.7.1.3 Definition. We say that a class of number-theoretic functions $\mathcal{C}$ is closed under $(\widetilde{\mu}y)$ just in case for every *total* g in the class, $\lambda\vec{x}.(\widetilde{\mu}y)g(y, \vec{x})$—which may fail to be total—is in the class. $\square$

2.7.1.4 Theorem. $\mathcal{P}$ *is closed under* $(\widetilde{\mu}y)$.

Proof. By the preceding Υ-remark, if $g \in \mathcal{R}$, then, for all $\vec{x}$, $(\widetilde{\mu}y)g(y, \vec{x}) \simeq (\mu y)g(y, \vec{x})$, thus $\lambda\vec{x}.(\widetilde{\mu}y)g(y, \vec{x}) \in \mathcal{P}$. $\square$

2.7.1.5 Corollary. *In 2.3.0.11 we may replace* μ *by* $\widetilde{\mu}$.

Proof. We reuse the proof of 2.3.0.11 by simply replacing $\widetilde{\mathcal{P}}$ by $\widetilde{\widetilde{\mathcal{P}}} = \text{Cl}(\mathcal{I}, \{\text{composition, primitive recursion}, \widetilde{\mu}\})$, throughout. In the forward part we use 2.7.1.4. In the part of said proof that begins by "Conversely", we replace μ by $\widetilde{\mu}$. $\square$

Immediately after the proof of 2.3.0.11 we noted

> The preceding corollary provides an alternative formalism—that is, a *syntactic, finite description* other than via URM programs ...

We cannot say the same here. The requirement that $(\widetilde{\mu}y)$ apply on total functions makes it a semantic rather than a syntactic operator: As we have seen, the problem of whether $\phi_i^{(n+1)}$ is in $\mathcal{R}$ or not is undecidable, indeed not even c.e.

Thus, given i we cannot know whether writing $(\widetilde{\mu}y)\phi_i^{(n+1)}(y, \vec{x}_n)$ *makes sense or not*: For we cannot *decide*, as Definition 2.7.1.2 requires, whether $\phi_i^{(n+1)}$ is total.

Pause. Hmm. Why can't we stop worrying about totalness and just *allow*—in defiance of Definition 2.7.1.2—$(\widetilde{\mu}y)$ to apply to all partial functions, including nontotal ones?◄

Well, a first approximation objection to the suggestion of defiance is that while $(\mu y)g(y, \vec{x})$ is correctly computed by the pseudo program below (cf. 2.1.1.17)

$$y \leftarrow 0$$
while
$$\neg g(y, \vec{x}) = 0$$
$$y \leftarrow y + 1$$
end

the *same program* does not compute $(\widetilde{\mu}y)g(y, \vec{x})$.

For the sake of argument, say, for a given $\vec{a}$, we have $g(0, \vec{a}) \uparrow$, but $g(1, \vec{a}) = 0$.

If so, the above pseudo program correctly computes $(\mu y)g(y, \vec{a})$ since the definition requires —for convergence—that $(\forall z < 1)g(z, \vec{a}) \downarrow$.

This is not the case for $(\widetilde{\mu}y)g(y, \vec{a})$, which ought to *return* $\min\{y : g(y, \vec{a}) = 0\} = 1$ (overlooking the nontotal-ness of g) but the program above loops forever, since the call to $g(0, \vec{a})$ does.

But wait! What if there *is* a really clever *alternative* program that *computes correctly* $\min\{y : g(y, \vec{a}) = 0\}$, for any computable g, *total or not*?

How can we establish that such a program does not exist (assuming we believe that it does not)? *By producing a partial recursive, nontotal, $\lambda xy.\psi(x, y)$, for which $\lambda x.(\widetilde{\mu}y)\psi(x, y) \notin \mathcal{P}$!*

2.7.1.6 Theorem. *There is a nontotal $\lambda xy.\psi(x, y) \in \mathcal{P}$ such that $\lambda x.(\widetilde{\mu}y)\psi(x, y) \notin \mathcal{P}$.*

Proof. The proof just firms up the "what if" discussion above that cast some initial doubt on the appropriateness of applying $(\widetilde{\mu}y)$ to nontotal functions. So let us define ψ by

$$\psi(x, y) = \begin{cases} 0 & \text{if } \big(y = 0 \wedge \phi_x(x) \downarrow \big) \vee \ y = 1 \\ \uparrow & \text{otherwise} \end{cases}$$

Given that the predicate $\phi_x(x) \downarrow$ is semi-recursive (2.5.0.15), closure properties of $\mathcal{P}_*$ (2.5.0.18) establish the top condition in the definition of ψ as semi-recursive. By *definition by positive cases* we have that $\psi \in \mathcal{P}$.

Let us evaluate

$$(\widetilde{\mu}y)\psi(x, y) \tag{1}$$

There are just two possible output values: The search returns 0 if $\phi_x(x) \downarrow$, while it returns 1 if $\phi_x(x) \uparrow$. Thus $\lambda x.(\widetilde{\mu}y)\psi(x, y)$ is χ_K and therefore is not in $\mathcal{P}$). □

Incidentally, note that χ_K, being a characteristic function, it is total, even though ψ is not.

2.7.1.7 Proposition. *The problem which requires us to determine for a given URM program i and input x whether a predetermined output y is attained is undecidable.*

We opted to say the above in English, in the first instance. Mathematically we are saying that $\lambda ix.\phi_i(x) = y$ is not in $\mathcal{R}_*$.

Proof. If the stated predicate is in $\mathcal{R}_*$ then so is $\lambda x.\phi_x(x) = y$ by closure properties. We will use a straightforward diagonalization to see that the latter cannot be.

$$\begin{array}{cccc} \phi_{\mathbf{0}}(\mathbf{0}) & \phi_0(1) & \phi_0(2) & \dots \\ \phi_1(0) & \phi_{\mathbf{1}}(\mathbf{1}) & \phi_1(2) & \dots \\ \vdots & \vdots & \vdots & \\ \phi_i(0) & \phi_i(1) & \phi_i(2) & \dots \\ \vdots & \vdots & \vdots & \end{array}$$

Define the new diagonal so that it differs from the one above at every place.

$$d(x) = \begin{cases} y+1 & \text{if } \phi_x(x) = y \\ y & \text{otherwise} \end{cases}$$

Thus, d is *not* a row above. On the other hand, since we *assumed* that $\lambda x.\phi_x(x) = y$ is recursive, we have that $d \in \mathcal{R}$ (it is total) hence $d = \phi_i$ for some i and hence *must* be a row. A contradiction. $\square$

2.7.1.8 Corollary. $\lambda ixy.\phi_i(x) = y$ *is not in* $\mathcal{R}_*$.

Proof. Otherwise we would contradict the preceding proposition [2.1.2.6(ii)]. $\square$

The next result, also based on a variant of diagonalization, has a *computational complexity* flavor: *There are arbitrarily hard-to-compute recursive functions!* Recall our concept of complexity of computable functions, Φ, introduced in the context of the dovetailing technique on p. 169.

2.7.1.9 Theorem. *For any a priori chosen recursive function* $\lambda x.g(x)$, *we can construct an* $f \in \mathcal{R}$ *such that,* for any i, if $f = \phi_i$, *then* $g(x) < \Phi_i(x)$ *for all* $x \geq i$.

Thus, we have *a priori* arbitrarily chosen a *level of computational "difficulty"*, g. We may choose a horrendously "big" g [e.g., $A_x(x)$, where A is the Ackermann function of Section 2.4]. Then we show how to find a function f, which no matter how we program it (via a URM i), such a program will take *more* than $g(x)$ "steps" to terminate on *almost all inputs* x, indeed on all $x \geq i$.

Proof. We want to build an f that for $i \leq x$ cannot be computed within $\leq g(x)$ steps.
   Thus, we need to meet two requirements:
   (1) Ensure that the f we build is recursive.
   (2) Ensure that *all* ϕ-indices i that satisfy

$$i \leq x \text{ and } \Phi_i(x) \leq g(x)$$

are *cancelled*.

   Let us thus set

$$I(x) \stackrel{\text{Def}}{=} \{i : i \leq x \wedge \Phi_i(x) \leq g(x)\}$$

Given that $\Phi_i(x) \leq y$ is recursive (cf. p. 169), so is $\Phi_i(x) \leq g(x)$ since $g \in \mathcal{R}$, and thus we have that $\lambda ix.i \in I(x)$ is recursive. So is the predicate $I(x) \neq \emptyset$, being equivalent to $(\forall i)_{\leq x} \neg \Phi_i(x) \leq g(x)$. We define f, for all x, as follows:

$$f(x) = \begin{cases} 1 + \sum_{i \in I(x)} \phi_i(x) & \text{if } I(x) \neq \emptyset \\ 1 & \text{otherwise} \end{cases} \qquad (3)$$

It is clear that $f \in \mathcal{P}$ from Exercise 2.12.28, but we need to work a bit more to show it is total, before we show that it has property (2) above. Let us define, for all i, x, the function h:
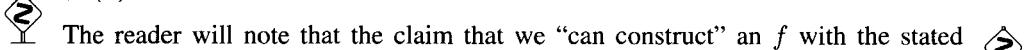
$$h(i, x) \stackrel{\text{Def}}{=} \text{if } i \in I(x) \text{ then } \phi_i(x) \text{ else } 1$$

By 2.1.2.9 and the earlier remark on $i \in I(x)$ we have that $h \in \mathcal{P}$. Since whenever $i \in I(x)$ holds we have $\phi_i(x) \downarrow$ (why?), it follows that $h \in \mathcal{R}$. Thus the totalness of f is established as soon as we rewrite (3) as

$$f(x) = \begin{cases} 1 + \sum_{i \leq x} h(i, x) & \text{if } I(x) \neq \emptyset \\ 1 & \text{otherwise} \end{cases}$$
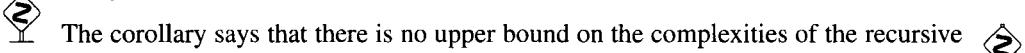
We finally turn to establish property (2) for f. Let then $f = \phi_k$ for some k (as it must since it is recursive) and pick any $x \geq k$. Can it be that $\Phi_k(x) \leq g(x)$?

No, for otherwise $k \in I(x)$ holds and therefore $f(x) = 1 + \ldots + \phi_k(x) + \ldots > \phi_k(x)$. A contradiction. $\square$

The reader will note that the claim that we "can construct" an f with the stated properties is apt.

2.7.1.10 Corollary. *There is no $\lambda x.g(x) \in \mathcal{R}$ such that every recursive ϕ_i is expressed as $\phi_i = \lambda x.d\big((\mu y)_{\leq g(x)} T(i, x, y)\big)$.*

Proof. See Exercise 2.12.41. $\square$

The corollary says that there is no upper bound on the complexities of the recursive functions.

It is noteworthy that there are arbitrarily hard to compute 0-1 valued recursive functions, that is, arbitrarily hard to compute recursive predicates. The following and its proof is due to Blum (1967).

2.7.1.11 Theorem. *For any a priori chosen recursive function $\lambda x.g(x)$, we can construct 0-1 valued $f \in \mathcal{R}$ such that, for any i, if $f = \phi_i$, then $g(x) < \Phi_i(x)$ a.e.*

Proof. With a 0-1 valued function we have to employ a more tricky index cancellation process, following Blum (1967). Adding all the ϕ_i —for the i we want to cancel—and then adding 1 on top of that will not work. We define instead as follows:

$$f(x) = \begin{cases} 1 \dot{-} \phi_k(x) & \text{if } k \text{ is the } smallest \text{ uncancelled } i \text{ in } I(x); \\ & \text{now } cancel \text{ the } k \text{ that was employed above;} \quad (1) \\ 1^{90} & \text{if no uncancelled } i \text{ exists in } I(x) \end{cases}$$

Let us leave for last the rather dull verification that (1) can be made mathematically precise toward showing that $f \in \mathcal{R}$. That f is 0-1 valued is obvious.

For now, we view the description in (1) as a reasonably complete guideline on how to "program" f and embark on proving its claimed complexity.

Let then $f = \phi_r$, for some r, and let us argue by contradiction.

Since $f = \phi_r$, the ϕ-index r is never cancelled.   (2)

---

[90]Could have used output 0; either is fine.

If the claim is false, then there is an infinite sequence of inputs above r,

$$r < x_1 < x_2 < x_3 < \ldots < x_m < \ldots <$$

on which the complexity of ϕ_r is $\leq g(x_i)$, for each x_i: $i = 1, 2, \ldots$.

Now, input x_i, for each i, satisfies $r < x_i$ and, by assumption, also $\Phi_r(x_i) \leq g(x_i)$. Moreover, r is uncancelled—that is, there are available indices to cancel in $I(x_i)$; cf. (1). So we cancel some $j < r$ (why $j < r$?) at this step, and set $f(x_i) = 1 \dotminus \phi_j(x_i)$.

From the above follows that we will have an *infinite sequence* of indices, j_i, that we will cancel, one for each x_i:

$$
\begin{array}{ccccccc}
j_1 & < & j_2 & < & j_3 & < \ldots < r \\
\uparrow & & \uparrow & & \uparrow & \\
\text{due to } x_1 & & \text{due to } x_2 & & \text{due to } x_3 &
\end{array}
$$

The inequalities are clear: Since r cannot be cancelled, it must be $j_1 < r$, and indeed j_1 is the smallest available. For x_2, index j_1 is already cancelled, so the next larger uncancelled index, j_2, is cancelled. Always, it must be that the indices we cancel are below r, as the latter is never cancelled.

This leads us to the absurdity that we have an infinite ascending sequence of integers between j_1 and r. We conclude that $\Phi_r(x) > g(x)$ a.e. as claimed.

But why is f recursive? We build f together with $\lambda x.c(x)$, the latter a function that stores, via prime power coding, the cancelled indices $i \leq x$, *after $f(x)$ has been defined*. We start with c, but first we recall the notation $x \in z$ from 2.4.4.1. The function c is given by a primitive recursion.

$$
\begin{cases}
c(0) & = 1 \\
& \qquad\qquad\qquad\qquad\qquad\qquad \overset{\text{uncancelled}}{\underset{\downarrow}{}} \\
c(x+1) & = c(x) * \Big[\text{if } (\exists y)_{\leq x}\big(y \in I(x) \wedge \neg y \;\in\; c(x)\big) \\
& \qquad \text{then } (\mu y)_{\leq x}\big(y \in I(x) \wedge \neg y \in c(x)\big) \text{ else } 1\Big]
\end{cases}
$$

$\lambda y x.y \in I(x)$ being in $\mathcal{R}_*$, we conclude that $c \in \mathcal{R}$. We return to f:

$$
f(x) = \begin{cases}
1 \dotminus \phi_{(\mu y)_{\leq x}\big(y \in I(x) \wedge \neg y \in c(x)\big)}(x) & \text{if } (\exists y)_{\leq x}\big(y \in I(x) \wedge \neg y \in c(x)\big) \\
1 & \text{if } \neg(\exists y)_{\leq x}\big(y \in I(x) \wedge \neg y \in c(x)\big)
\end{cases}
$$

Since $\lambda xy.\phi_x(y) \in \mathcal{P}$ and by Exercise 2.12.28, $f \in \mathcal{P}$. It is also defined on any x since the index of ϕ is in $I(x)$. $\square$

The above theorem establishes the existence of arbitrarily hard to compute *recursive predicates*. Does this mean that there are recursive predicates that are *not* in $\mathcal{PR}_*$? Yes (cf. Exercise 5.3.32).

2.7.2 Reducibility via the S-m-n Theorem

We now turn to the development of the technique of reductions, using the S-m-n theorem.

2.7.2.1 Definition. (Strong Reducibility) We say that the set A (subset of $\mathbb{N}$) is *strongly reducible* to set B, in symbols $A \leq_m B$, iff there is a recursive function f such that for all x, we have $x \in A$ iff $f(x) \in B$. We say that f *effects the reducibility*. $\square$

2.7.2.2 Remark. Several remarks are in order:

(1) The m in the reducibility symbol reflects the fact that f is not required to be 1-1. So, strong reducibility, by default, is a *many-one reducibility* or *m-reducibility*.

   We also have *1-1 reducibility* or *1-reducibility*. This is when f is 1-1.

(2) The condition $x \in A$ iff $f(x) \in B$ says that if we know how to decide $z \in B$ and also know how to compute $f(x)$ for all x, then we know how to decide $x \in A$. Thus, the symbol $\leq_m$ is apt: Intuitively A is "more solvable" than B since we can decide it if we can decide B. Conversely, B is more unsolvable than A.

   We express this technically in the proposition below.

(3) By definition, $A \leq_m B$ iff $A = \{x : f(x) \in B\}$. That is, iff $A = f_{\leftarrow}(B)$. $\square$

2.7.2.3 Proposition. *Suppose that $A \leq_m B$. Then*

(1) *A is recursive if B is. Contrapositively, $B \notin \mathcal{R}_*$ if $A \notin \mathcal{R}_*$.*

(2) *A is semi-computable if B is. Contrapositively, B is not c.e. if A is not c.e.*

Proof.

(1) If $z \in B$ is recursive, then so is $f(x) \in B$ by the assumption on f and by 2.1.2.24.

(2) If $z \in B$ is semi-recursive, then so is $f(x) \in B$ by the assumption on f and by 2.5.0.20. $\square$

2.7.2.4 Definition. (Complete Index Sets) Given a subset $\mathcal{C} \subseteq \mathcal{P}$, we call $\{x : \phi_x \in \mathcal{C}\}$ a *complete index set* (defined by $\mathcal{C}$). $\square$

2.7.2.5 Remark. That is, a complete index set $A = \{x : \phi_x \in \mathcal{C}\}$ is the set of *all* (codes of) URM programs that compute the functions of some given subset $\mathcal{C}$ of $\mathcal{P}$. Indeed say $f \in \mathcal{C}$. As this is computable, take *any* program i for f, that is, $f = \phi_i$. Now $\phi_i \in \mathcal{C}$ yields $i \in A$ by the definition of A. $\square$

   This subsection deals with the undecidability of membership in several complete index sets. Indeed, "several" is an understatement. We will conclude with the rather

surprising Theorem of Rice, according to which the *only* decidable such problems involve the *two* trivial cases: $\mathcal{C} = \emptyset$ or $\mathcal{C} = \mathcal{P}$.

2.7.2.6 Remark. (The General Technique) The technique in general outline goes like this: We want to show that some A given as $\{x : \phi_x \in \mathcal{C}\}$, for some $\mathcal{C} \subseteq \mathcal{P}$, is not recursive.

Equipped with 2.7.2.3 we attempt to show either $K \leq_m A$ or $\overline{K} \leq_m A$—whichever is easier. The latter, of course, yields more information (a stronger result): that A is not c.e.

To this end, we need to demonstrate that there is an $h \in \mathcal{R}$ that effects one of $K \leq_m A$ or $\overline{K} \leq_m A$.

To execute this plan, we utilize the S-m-n theorem so that we come up with a primitive recursive h such that

Case of $K \leq_m A$:

$$\phi_{h(x)} = \begin{cases} \text{some specific } f \in \mathcal{C} & \text{if } \phi_x(x) \downarrow \\ \text{some specific } g \notin \mathcal{C} & \text{if } \phi_x(x) \uparrow \end{cases}$$

Thus, $h(x) \in A$ iff the top case holds, iff $x \in K$—that is, $\phi_x(x) \downarrow$. For short, $K \leq_m A$ via h.

Case of $\overline{K} \leq_m A$:

$$\phi_{h(x)} = \begin{cases} \text{some specific } f \in \mathcal{C} & \text{if } \phi_x(x) \uparrow \\ \text{some specific } g \notin \mathcal{C} & \text{if } \phi_x(x) \downarrow \end{cases}$$

Thus, $h(x) \in A$ iff the top case holds, iff $x \in \overline{K}$—that is, $\phi_x(x) \uparrow$. For short, $\overline{K} \leq_m A$ via h. □

The following theorem is important both in content and in regards to the technique employed for its proof.

2.7.2.7 Theorem. *The following sets are not recursive.*

(1) $A = \{x : \phi_x \text{ is a constant function}\}$

(2) $B = \{x : \phi_x \text{ is total}\} = \{x : \phi_x \in \mathcal{R}\}$

(3) $C = \{\langle x, y \rangle : y \in \mathrm{ran}(\phi_x)\}$

(4) $D = \{\langle x, y, z \rangle : z = \phi_x(y)\}$

(5) $E = \{x : \mathrm{dom}(\phi_x) = \emptyset\}$

(6) $F = \{x : \mathrm{dom}(\phi_x) \text{ is finite}\}$

(7) $G = \{x : \mathrm{dom}(\phi_x) \text{ is infinite}\}$

(8) $H = \{x : \mathrm{ran}(\phi_x) = \emptyset\}$

(9) $I = \{x : \mathrm{ran}(\phi_x) \text{ is finite}\}$

(10) $J = \{x : \text{ran}(\phi_x) \text{ is infinite}\}$

Proof.

(1) $A = \{x : \phi_x \text{ is a constant function}\}$.

We will be more expansive in just this *first* case. Following 2.7.2.6, we want to find an $h \in \mathcal{R}$ such that

$$\phi_{h(x)} = \begin{cases} \text{some specific constant function} & \text{if } \phi_x(x) \downarrow \\ \text{some specific non constant function} & \text{if } \phi_x(x) \uparrow \end{cases} \quad (\dagger)$$

Well, the simplest solution is probably this: Define, for all x and y

$$\psi(x, y) \simeq \begin{cases} 0 & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{if } \phi_x(x) \uparrow \end{cases} \quad (*)$$

We see first at the intuitive level that ψ is computable: Given x, y. We ignore y. Next, we fetch the URM M of code x and call it on input x. If it ever halts, then we print "0" and halt everything. If M never halts, then our process will never return from the call, which is the correct behavior for $\psi(x, y)$—bottom case.

Intuitively, we cannot expect to yield some output in the bottom case, since, at least in the process described above, the call to M for input x will never halt to give us an opportunity to print anything.

Pause. Do you have a *mathematical* reason as to why defining ψ so that, say, it yields 42 in the bottom case renders ψ non computable (i.e., not in $\mathcal{P}$)?◄

Mathematically, $\psi \in \mathcal{P}$ via definition by positive cases (2.5.0.30). Of course, "$\phi_x(x) \uparrow$" is not a positive case, being non c.e. However, reference back to 2.5.0.30 shows that the last (bottom) case is the "otherwise" case.

By the S-m-n theorem, there is an $h \in \mathcal{PR}$ such that, for all x and y,

$$\phi_{h(x)}(y) \simeq \begin{cases} 0 & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{if } \phi_x(x) \uparrow \end{cases} \quad (**)$$

This can be rewritten as (note the change from $\simeq$ to $=$)

$$\phi_{h(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \downarrow \\ \emptyset & \text{if } \phi_x(x) \uparrow \end{cases} \quad (***)$$

where $\emptyset$ is the empty function—clearly not a constant function! We have achieved the setup ($\dagger$) and we conclude by directly invoking 2.7.2.6.

(2) $B = \{x : \phi_x \text{ is total}\} = \{x : \phi_x \in \mathcal{R}\}$.

Note that $(***)$ can be recast as

$$\phi_{h(x)} = \begin{cases} \text{a specific total } f & \text{if } \phi_x(x) \downarrow \\ \text{a specific nontotal } g & \text{if } \phi_x(x) \uparrow \end{cases}$$

Thus, $K \leq_m B$ and therefore $B \notin \mathcal{R}_*$ as in 2.7.2.6.

This result says less than what we already proved in 2.5.0.29, however, it is important to see this alternative technique even if (seemingly) achieves less. Seemingly. We will refine the technique in the next theorem, to redidcover the non semi-recursiveness of B.

(3) $C = \{\langle x, y \rangle : y \in \mathrm{ran}(\phi_x)\}$.

We have seen this example, and the next, already in Subsection 2.7.1. Let us use the present technique. If C is recursive, then so is $C_0 = \{x : 0 \in \mathrm{ran}(\phi_{h(x)})\}$ (by 2.1.2.6)—where h is the same as above.

But $0 \in \mathrm{ran}(\phi_{h(x)})$ can happen iff we are in the top case of $(***)$, which is in the case $x \in K$. Thus $K \leq_m C_0$ via h.

(4) $D = \{\langle x, y, z \rangle : z = \phi_x(y)\}$.

If D is recursive, then so is $D_0 = \{x : 0 = \phi_{h(x)}(0)\}$. The latter is equivalent to $x \in K$ as above and we conclude exactly as in the case of C above: $K \leq_m D_0$.

(5) $E = \{x : \mathrm{dom}(\phi_x) = \emptyset\}$.

We can still mine diverse unsolvability results from the very same setup $(***)$ above. We rewrite this as

$$\phi_{h(x)} = \begin{cases} \text{a } g \text{ with a non-empty domain} & \text{if } \phi_x(x) \downarrow \\ \text{an } f \text{ with an empty domain} & \text{if } \phi_x(x) \uparrow \end{cases}$$

Thus, as in 2.7.2.6, $h(x) \in E$ iff we are in the bottom case; iff $x \in \overline{K}$. That is, $\overline{K} \leq_m E$ via h. We have proved more than what we were asked to: E is not even semi-recursive, let alone decidable.

(6) $F = \{x : \mathrm{dom}(\phi_x) \text{ is finite}\}$.

We rewrite $(***)$ as

$$\phi_{h(x)} = \begin{cases} \text{a } g \text{ with an infinite domain} & \text{if } \phi_x(x) \downarrow \\ \text{an } f \text{ with a finite domain} & \text{if } \phi_x(x) \uparrow \end{cases}$$

Thus, $h(x) \in F$ iff we are in the bottom case; iff $x \in \overline{K}$. That is $\overline{K} \leq_m F$ via h. Once again we have proved more than we were asked to: F is not semi-recursive.

(7) $G = \{x : \mathrm{dom}(\phi_x) \text{ is infinite}\}$.

Yet again, we rely on $(***)$. Indeed, see the argument for F above. We have that $h(x) \in G$ iff we are in the *top* case; iff $x \in K$. That is, $K \leq_m G$ via h and thus G is not recursive.

(8) $H = \{x : \mathrm{ran}(\phi_x) = \emptyset\}$.

One last time we mine $(***)$, rewriting it as

$$\phi_{h(x)} = \begin{cases} \text{a } g \text{ with a non-empty range} & \text{if } \phi_x(x) \downarrow \\ \text{an } f \text{ with an empty range} & \text{if } \phi_x(x) \uparrow \end{cases}$$

Thus, $h(x) \in H$ iff we are in the bottom case; iff $x \in \overline{K}$. Thus $\overline{K} \leq_m H$ and H is not semi-recursive.

(9) $I = \{x : \mathrm{ran}(\phi_x) \text{ is finite}\}$.

This case needs a fresh start, since neither $\lambda y.0$ nor $\emptyset$ have an infinite *range*, as needed for the "dichotomy" *infinite* vs. *finite* (range), toward applying the technique of 2.7.2.6. So, we define a new function, for all x and y, by

$$\chi(x,y) \simeq \begin{cases} y & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

χ is computable. Intuitively, given x and y we decode x to get the URM M that it codes. We then call M on input x. If it ever halts, we print y and halt all; otherwise we keep going.

Mathematically, χ is defined by positive cases, since $\phi_x(x) \downarrow$ is c.e. Thus it is in $\mathcal{P}$. The S-m-n theorem allows the existence of a $k \in \mathcal{R}$, such that, for all x and y,

$$\phi_{k(x)}(y) \simeq \begin{cases} y & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Put more conveniently, with no reference to inputs,

$$\phi_{k(x)} = \begin{cases} \lambda y.y & \text{if } \phi_x(x) \downarrow \\ \emptyset & \text{otherwise} \end{cases} \tag{$\ddagger$}$$

Note that $k(x) \in I$ iff we are in the bottom case of $(\ddagger)$; iff $x \in \overline{K}$. Thus $\overline{K} \leq_m I$ via k, rendering I non c.e., which says more than what we set out to prove.

(10) $J = \{x : \mathrm{ran}(\phi_x) \text{ is infinite}\}$.

We reuse $(\ddagger)$. Here $k(x) \in J$ iff we are in the *top* case of $(\ddagger)$; iff $x \in K$. Thus $K \leq_m J$ via k, rendering $x \in J$ undecidable. $\qquad\square$

Worth stating. Since the h and k utilized above are S-m-n functions, they are 1-1 (cf. 2.6.0.39). Therefore *all* reducibilities that we have effected above are 1-reducibilities, $\leq_1$.

2.7.2.8 Theorem. *None of the sets in 2.7.2.7 are semi-recursive, except C and D.*

Proof. D is semi-recursive by 2.5.0.7. As for C, we see that $y \in \mathrm{ran}(\phi_x) \equiv (\exists z)\phi_x(z) = y$. Its semi-recursiveness follows from 2.5.0.7 via 2.5.0.18.

We now turn to those sets listed in 2.7.2.7, which we have not already proved to be non c.e. in the proof of said theorem.

(1) $A = \{x : \phi_x \text{ is a constant function}\}$.

The obvious approach, that is, badly imitating 2.7.2.6 and modifying $(*)$ to read

$$\psi(x, y) \simeq \begin{cases} 0 & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases}$$

will not work. The *intuitive* reason is that if we try to compute this new ψ in the obvious way, given x and y we will ignore y, and will decode x to obtain the machine it denotes, M. We will run M on input x and will output and stop everything precisely if M is in an infinite loop, which is precisely if $\phi_x(x) \uparrow$.

Otherwise ($\phi_x(x) \downarrow$) we will ensure that the overall computation never halts by a technique we have already employed (in 2.5.0.12).

The catch is that this "obvious" way is doomed, for our program—indeed, no program—can test, or even just verify that $\phi_x(x) \uparrow$.

> The *definitive* reason that this ψ is not computable is this: If it were, then so would be $\lambda x.\psi(x, x)$. But the domain of the latter is $\overline{K}$. Impossible, because this set is not the domain of any partial recursive function.

Pause. Why "definitive"? Isn't the intuitive reason (of the uncomputability of ψ) enough?◄

No. The intuition only *warns* and *guides*; it does not *prove*. After all, the suggested "program that did not work" was just *one* suggested, and "obvious", program to compute ψ.

Why can it not be the case that a future programmer might come up with a really clever and non obvious URM that computes ψ?

Precisely because we got the definitive answer *mathematically*: There can be *no* such a URM, now or ever; it does not exist.

OK, here is how to do it right: We want to build a partial recursive ψ such that

$$\psi(x, y) \simeq \begin{cases} 0 & \text{if } \phi_x(x) \uparrow \\ \text{not a constant result} & \text{if } \phi_x(x) \downarrow \end{cases}$$

In view of the above remarks, we cannot use the condition "$\phi_x(x) \uparrow$" outright as the top condition, so we will *approximate* it with "$\phi_x(x)$ does not converge in $\leq y$ steps".

Note that for a "large" (number of steps) y, the casual (and impatient) observer will consider a computation for $\phi_x(x)$, which is still going, as divergent.

So we finally define

$$\psi(x, y) \simeq \begin{cases} 0 & \text{if } \phi_x(x) \text{ does not converge in } \leq y \text{ steps} \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases} \qquad (i)$$

This ψ *is* computable! Let us see why, intuitively at first. We program as follows: Given inputs x and y. We call the URM M, coded by x, on input x. If M has not stopped after y steps of its computation, then we print 0 and stop everything. In the contrary case—that is, the call to M with input x stopped within $\leq y$ steps—we deliberately enter an infinite loop as in 2.5.0.12.

Mathematically, (i) can be rewritten as

$$\psi(x, y) \simeq \begin{cases} Z(y) & \text{if } \neg\Phi_x(x) \leq y \\ \emptyset(y) & \text{if } \Phi_x(x) \leq y \end{cases} \qquad (ii)$$

where $Z = \lambda y.0$. Since the conditions are recursive (cf. p. 169), we have, by Exercise 2.12.28, that $\psi \in \mathcal{P}$.[91]

By the S-m-n theorem, we have a primitive recursive σ such that, for all x and y,

$$\phi_{\sigma(x)}(y) \simeq \begin{cases} Z(y) & \text{if } \neg\Phi_x(x) \leq y \\ \emptyset(y) & \text{if } \Phi_x(x) \leq y \end{cases} \qquad (iii)$$

Let us now consider the two cases below:

Case 1: $\phi_x(x) \uparrow$. Then $\neg\Phi_x(x) \leq y$ is true; for this x and *all* y, the top case applies. That is:

$$\phi_{\sigma(x)} = \lambda y.0 \qquad (iv)$$

Case 2: $\phi_x(x) \downarrow$. Let y_0 be smallest such that $\Phi_x(x) \leq y_0$. That is

$$\text{For } y = 0, 1, \ldots, y_0 - 1, \text{ we have } \neg\Phi_x(x) \leq y$$

In this case

$$\phi_{\sigma(x)} = \overbrace{\langle 0, 0, \ldots, 0\rangle}^{y_0 \text{ zeros}} \qquad (v)$$

where in (v) we have denoted the finite function

$$f(y) = \text{if } x = 0 \lor x = 1 \lor \cdots \lor x = y_0 - 1 \text{ then } 0 \text{ else } \emptyset(y)$$

as the finite sequence of its outputs. Of course, $f \in \mathcal{P}$.

We summarize what cases 1 and 2 say in (iv) and (v):

$$\phi_{\sigma(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \uparrow \\ \underbrace{\langle 0, 0, \ldots, 0\rangle}_{y_0 \text{ zeros}} & \text{if } \phi_x(x) \downarrow \end{cases} \qquad (\dagger)$$

Given that the function in the bottom case is *not* a constant function, we immediately have $\sigma(x) \in A$ iff $x \in \overline{K}$, or $\overline{K} \leq_m A$ as needed.

---

[91]One is normally less pedantic and rather than explicit function calls $Z(y)$ and $\emptyset(y)$ writes 0 and $\uparrow$ respectively.

(2) $B = \{x : \phi_x \text{ is total}\} = \{x : \phi_x \in \mathcal{R}\}$.

We may reuse (†) immediately above, since the top case is total while the bottom case is nontotal. Thus, $\sigma(x) \in B$ iff $x \in \overline{K}$, or $\overline{K} \leq_m B$ as needed.

(3) $G = \{x : \text{dom}(\phi_x) \text{ is infinite}\}$.

We may reuse (†) since the top case has infinite domain while the bottom case has finite domain. Thus, $\sigma(x) \in G$ iff $x \in \overline{K}$, or $\overline{K} \leq_m G$ as needed.

(4) $J = \{x : \text{ran}(\phi_x) \text{ is infinite}\}$.

We cannot reuse (†) here as both the top and bottom cases have finite ranges. We work entirely analogously to (ii) above, and define, for all x, y,

$$\chi(x, y) \simeq \begin{cases} y & \text{if } \neg\Phi_x(x) \leq y \\ \emptyset(y) & \text{if } \Phi_x(x) \leq y \end{cases}$$

As χ is defined from partial recursive functions by recursive cases, it is in $\mathcal{P}$. By S-m-n we have a primitive recursive τ such that, for all x, y,

$$\phi_{\tau(x)}(y) \simeq \begin{cases} y & \text{if } \neg\Phi_x(x) \leq y \\ \emptyset(y) & \text{if } \Phi_x(x) \leq y \end{cases}$$

A similar analysis as above shows readily that

If $\phi_x(x) \uparrow$, then $\phi_{\tau(x)} = \lambda y.y$, while if $\phi_x(x) \downarrow$, then

$$\phi_{\tau(x)} = \langle 0, 1, \ldots, y_0 - 1 \rangle$$

a finite function displayed as a sequence of outputs, where y_0 is smallest y such that $\Phi_x(x) \leq y$.

Thus,

$$\phi_{\tau(x)} = \begin{cases} \lambda y.y & \text{if } \phi_x(x) \uparrow \\ \langle 0, 1, \ldots, y_0 - 1 \rangle & \text{if } \phi_x(x) \downarrow \end{cases}$$

and therefore $\tau(x) \in J$ iff $x \in \overline{K}$, or $\overline{K} \leq_m J$ as needed.   □

The techniques used so far are unified in the results that we develop below.

2.7.2.9 Theorem. (The Rice Lemma) *Given a complete index set $A = \{x : \phi_x \in \mathcal{C}\}$—where $\mathcal{C} \subseteq \mathcal{P}$. If some $f \in \mathcal{C}$ has an extension $g \in \mathcal{P} - \mathcal{C}$, then $\overline{K} \leq_m A$.*

Proof. Let $\phi_m \in \mathcal{C}$ and $\phi_n \notin \mathcal{C}$, where $\phi_m \subseteq \phi_n$.[92] The plan is to prove that a primitive recursive h exists such that

$$\phi_{h(x)} = \begin{cases} \phi_m & \text{if } \phi_x(x) \uparrow \\ \phi_n & \text{if } \phi_x(x) \downarrow \end{cases} \tag{1}$$

[92]Cf. p. 45.

As we have already observed, to avail ourselves of the "definition by positive cases" technique, the top case must be the "otherwise". But if so, we cannot allow, in general, an "output" other than "↑" (cf. the remarks in 2.5.0.30 and the followup in Exercise 2.12.35).

Thus, once again, we will approximate the condition $\phi_x(x) \uparrow$.

$$\chi(x, y) \simeq \begin{cases} \phi_m(y) & \text{if } \phi_x(x) \text{ does } not \text{ converge before } \phi_m(y) \text{ does} \\ \phi_n(y) & \text{if } \phi_x(x) \text{ converges before } \phi_m(y) \text{ does} \end{cases}$$

Is χ computable? Intuitively, it is. Here is how: Let H be a URM for verifying $\phi_x(x) \downarrow$, M a URM that computes ϕ_m, and N a URM for ϕ_n. Let x and y be given.

We run H on input x, and M on input y in parallel. If M halts but H is still running, then we print $\phi_m(y)$ and stop everything.

Worth noting. If each of H and M loops for ever, then the top condition is valid; we *correctly* output $\phi_m(y)$ in this case (we "output" ↑—that is, nothing).

If, on the other hand, H halts before M does, then we abort M and call N on input y in order to (if convergence is achieved) output $\phi_n(y)$.

The mathematical reason for the computability of χ is based on the above informal description.

The content of the above ⚡-comment is the "otherwise"; thus we achieve a definition by positive cases:

$$\chi'(x, y) \simeq \begin{cases} \phi_m(y) & \text{if } (\exists z)\big(\Phi_m(y) = z \wedge \neg \Phi_x(x) < z\big) \\ \phi_n(y) & \text{if } (\exists z)\big(\Phi_x(x) = z \wedge \neg \Phi_m(y) \leq z\big) \\ \uparrow & \text{otherwise} \end{cases}$$

Since the above is a definition by positive cases—recall that $\Phi_i(x) = w$, $\Phi_i(x) < w$, and $\Phi_i(x) \leq w$ are (primitive) recursive—$\chi' \in \mathcal{P}$.

Pause. But is $\chi = \chi'$?◄

Yes. The top condition for χ' says "$\phi_x(x)$ does *not* converge before $\phi_m(y)$ does"—this is the case of $\phi_m(y) \downarrow$, where $\phi_x(x)$ may or may not converge. The case of $\phi_m(y) \uparrow$ and $\phi_x(x) \uparrow$ is covered by the "otherwise" as we already have remarked, noticing that both the top and middle cases now fail. The middle condition says "$\phi_x(x)$ converges *before* $\phi_m(y)$ does".

By the S-m-n theorem there is an $h \in \mathcal{PR}$ such that, for all x and y,

$$\phi_{h(x)}(y) \simeq \begin{cases} \phi_m(y) & \text{if } (\exists z)\big(\Phi_m(y) = z \wedge \neg \Phi_x(x) < z\big) \\ \phi_n(y) & \text{if } (\exists z)\big(\Phi_x(x) = z \wedge \neg \Phi_m(y) \leq z\big) \\ \uparrow & \text{otherwise} \end{cases}$$

We can now verify that we have (1) above.

- Let $\phi_x(x) \uparrow$. Then $\Phi_x(x) \leq z$ is false for all z, hence the middle case cannot apply.

(a) If we have $\phi_m(y) \downarrow$, then $(\exists z)\Phi_m(y) = z$, thus the top condition is true and $\phi_{h(x)}(y) = \phi_m(y)$.

(b) If we have $\phi_m(y) \uparrow$, then $\Phi_m(y) \leq z$ is false for all z, thus only the "otherwise" applies. We have, once more $\phi_{h(x)}(y) \simeq \phi_m(y)$ (note the "$\simeq$").

For short, $\phi_{h(x)} = \phi_m$ in this case.

- Let $\phi_x(x) \downarrow$.   Let z be smallest such that $\Phi_x(x) = z$. Now fix a y.

  (i) If $\Phi_m(y) \leq z$, then the top case holds, thus $\phi_{h(x)}(y) \simeq \phi_m(y)$. But $\phi_m(y) \downarrow$ (why?), thus, by $\phi_m \subseteq \phi_n$, we have $\phi_m(y) = \phi_n(y)$ (note the "=").

  Therefore $\phi_{h(x)}(y) = \phi_n(y)$.[93]

  (ii) If $\neg\Phi_m(y) \leq z$, then the middle case holds, and again $\phi_{h(x)}(y) \simeq \phi_n(y)$ ("$\simeq$" this time is essential, as it may be that $\phi_n(y) \uparrow$.)

For short, $\phi_{h(x)} = \phi_n$ in this case.

This establishes (1), and hence the equivalences $h(x) \in A$ iff $\phi_{h(x)} \in \mathcal{C}$ iff $\phi_{h(x)} = \phi_m$ iff $x \in \overline{K}$. That is, $\overline{K} \leq_m A$ via h. □

2.7.2.10 Corollary. *Given a complete index set $A = \{x : \phi_x \in \mathcal{C}\}$—where $\mathcal{C} \subseteq \mathcal{P}$. If some $f \in \mathcal{C}$ has an extension $g \in \mathcal{P} - \mathcal{C}$, then A is not c.e.*

2.7.2.11 Corollary. (The Theorem of Rice) *A compete index set $A = \{x : \phi_x \in \mathcal{C}\}$ is recursive iff it is* trivial, *meaning that either $A = \emptyset$ or $A = \mathbb{N}$.*

Proof. The *if* part is immediate since, in fact, $\emptyset$ and $\mathbb{N}$ are primitive recursive.

As for the *only if*, say A is recursive. Then A and $\mathbb{N} - A$ (or $\overline{A}$), that is,

$$\{x : \phi_x \in \mathcal{P} - \mathcal{C}\}$$

are both c.e.

We consider two cases. First, let $\emptyset^{94} \in \mathcal{C}$. Since A is semi-recursive, 2.7.2.10 yields that every computable extension of $\emptyset$ is in $\mathcal{C}$. Thus $\mathcal{C} = \mathcal{P}$ and hence $A = \mathbb{N}$.

Second, let $\emptyset \in \mathcal{P} - \mathcal{C}$. As above, since $\mathbb{N} - A$ is c.e., 2.7.2.10 yields that every computable extension of $\emptyset$ is in $\mathcal{P} - \mathcal{C}$. That is, $\mathcal{P} - \mathcal{C} = \mathcal{P}$ and hence $\mathbb{N} - A = \mathbb{N}$. Therefore, $A = \emptyset$. □

2.7.2.12 Example. We look back to 2.7.2.7. We see at once by application of Rice's theorem that each of the sets A, B, and $E-J$ are not recursive.

Each of them is a nontrivial complete index set. For example, the set of constants $\mathcal{C}$ is not equal to either $\emptyset$ or $\mathcal{P}$, for, on one hand, constant functions exist (!), such as

---

[93] Again note the "=" used for emphasis. The more general symbol $\simeq$ would also be correct.
[94] The empty function in this context

$\lambda x.380$, or $\lambda x.0$, and, on the other hand, not every computable function is a constant; for example, $\lambda xy.x + y$, $\lambda x.x$, etc. Thus, $\emptyset \neq A \neq \mathbb{N}$.

Similarly one shows all of E–J to be nontrivial.

The sets C and D are not complete index sets so the theorem of Rice does not help. One can employ either the technique of 2.7.2.7 or direct diagonalization (cf. 2.7.1.7). □

2.7.2.13 Example. Wait a minute! We have not verified that, e.g., set F of 2.7.2.7 is a complete index set.

As a principle we *must*, in each case prior to the application of Rice's theorem. Often it is easy to do so.

Apropos $F = \{x : \mathrm{dom}(\phi_x) \text{ is finite}\}$, let us set $C = \{f \in \mathcal{P} : \mathrm{dom}(f) \text{ is finite}\}$. Thus $F = \{x : \phi_x \in C\}$. And this is a complete index set in the format defined in 2.7.2.4. □

2.7.2.14 Example. We have seen already that every computable function has infinitely many ϕ-indices by arguing the case via URM programs. Here is a "high level" approach: Let $f \in \mathcal{P}$. Then $\emptyset \neq \{x : \phi_x = f\} \neq \mathbb{N}$. By Rice's theorem, $\{x : \phi_x = f\}$ is not recursive, hence must be infinite (every finite set is primitive recursive). □

2.7.2.15 Example. But how about K? Is $K = \{x : \phi_x(x) \downarrow\}$ a complete index set? That is, is there a $C \subseteq \mathcal{P}$ such that $K = \{x : \phi_x \in C\}$? We will answer this negatively in Section 2.9. □

The Rice Lemma 2.7.2.9 can help in easily establishing non semi-recursiveness. Let us revisit 2.7.2.8.

2.7.2.16 Example. Look at $E = \{x : \mathrm{dom}(\phi_x) = \emptyset\}$. For convenience let us set

$$C \overset{\mathrm{Def}}{=} \{f \in \mathcal{P} : \mathrm{dom}(f) = \emptyset\}$$

Note that $\mathrm{dom}(\emptyset) = \emptyset$. However, $\lambda x.0$ extends $\emptyset$ but is not in C—its domain is $\mathbb{N}$. By 2.7.2.10, E is not c.e.

A similar argument holds for F, since $\emptyset$ has a finite domain hence is in F's implied "C". Any constant function extends $\emptyset$ but is not in this "C". Thus, F is not c.e.

Corollary 2.7.2.10 can similarly show that H and I are not c.e. although we should use $\lambda x.x$ or any other computable infinite-range function—instead of $\lambda x.0$—as an extension of $\emptyset$ that lies outside the "C" of these two complete index sets.

However the corollary does not help the proof of non semi-recursiveness of A, B, G, or J. A new *general* technique is needed. □

The following theorem is the contribution of several people (Rice, Myhill, Shapiro, McNaughton). First a definition.

2.7.2.17 Definition. (Finite Functions) A number theoretic function f is finite iff $\mathrm{dom}(f)$ is a finite set. $\qquad\square$

2.7.2.18 Theorem. *Given $A = \{x : \phi_x \in \mathcal{C}\}$, where $\mathcal{C} \subseteq \mathcal{P}$. Suppose that some $f \in \mathcal{C}$ has no finite subfunction ξ—i.e., $\xi \subseteq f$—that is a member of $\mathcal{C}$. Then $\overline{K} \leq_m A$.*

Proof. The idea is to find, using the S-m-n theorem, an $h \in \mathcal{PR}$ such that

$$\phi_{h(x)} = \begin{cases} \xi & \text{if } \phi_x(x) \downarrow \\ f & \text{if } \phi_x(x) \uparrow \end{cases} \tag{1}$$

If this succeeds, then $h(x) \in A$ iff $\phi_{h(x)} \in \mathcal{C}$ iff $\phi_{h(x)} = f$ iff $x \in \overline{K}$. Thus, $\overline{K} \leq_m A$.

Now to justify (1) we straightforwardly generalize the technique from 2.7.2.8, case (4). Thus we define, for all x, y,

$$\chi(x, y) \simeq \begin{cases} f(y) & \text{if } \neg\Phi_x(x) \leq y \\ \emptyset(y) & \text{if } \Phi_x(x) \leq y \end{cases}$$

This is a definition by recursive cases, thus $\chi \in \mathcal{P}$. By the S-m-n theorem we have an $h \in \mathcal{PR}$ such that

$$\phi_{h(x)}(y) \simeq \begin{cases} f(y) & \text{if } \neg\Phi_x(x) \leq y \\ \emptyset(y) & \text{if } \Phi_x(x) \leq y \end{cases} \tag{2}$$

Let us now consider the two cases:

(a) $\phi_x(x) \uparrow$. Then the top condition of (2) is true for all y, thus $\phi_{h(x)} = f$ in this case.

(b) $\phi_x(x) \downarrow$. Let y_0 be the smallest y-value such that the bottom condition in (2) holds. Assume first that $y_0 \geq 1$. Thus, for $y = 0, 1, \ldots, y_0 - 1$, we have that $\neg\Phi_x(x) \leq y$ holds, and therefore

$$\text{for } y = 0, 1, \ldots, y_0 - 1, \text{ it is } \phi_{h(x)}(y) \simeq f(y), \text{ but } \phi_{h(x)}(y) \uparrow, \text{ if } y \geq y_0 \tag{3}$$

Let us call ξ the finite subfunction of f in (3) above: $\xi = f \upharpoonright \{0, 1, \ldots, y_0 - 1\}$ (cf. definitions and notation on p. 45). Thus, $\phi_{h(x)} = \xi \,(\subseteq f)$ in this case, for $y_0 > 0$.

If $y_0 = 0$, then the bottom condition holds for all y, thus $\phi_{h(x)} = \emptyset$. But $\emptyset \subseteq f$.

We have verified (1). $\qquad\square$

2.7.2.19 Corollary. *Let the complete index set $A = \{x : \phi_x \in \mathcal{C}\}$ be c.e. Then $f \in \mathcal{C}$ iff some finite subfunction of f is in $\mathcal{C}$.*

Proof. The *only if* is by 2.7.2.18, for otherwise $\overline{K} \leq_m A$, contradicting the assumption. The *if* is by 2.7.2.9, for if $\mathcal{C} \ni \xi \subseteq f$, then $f \in \mathcal{C}$. $\qquad\square$

2.7.2.20 Example. We return to the concluding comment

> However the corollary does not help the proof of non semi-recursiveness of
> A, B, G, or J. A new *general* technique is needed.

from 2.7.2.16. The respective C-sets in each of the index sets A and B contain total
functions only. Thus no f in such a C can have a *finite* (nontotal!) subfunction also
in C. By 2.7.2.19, neither A nor B can be c.e.

Similarly, the respective C-sets in each of the index sets G and J contain functions
with *infinite domains* only. Thus no f in such a C can have a *finite* subfunction also
in C. Again, by 2.7.2.19, neither G nor J can be c.e. $\square$

2.7.2.21 Remark. A variant of a complete index set has this form

$$A = \{x : W_x \in C_*\} \tag{1}$$

where $C_* \subseteq \mathcal{P}_*$. That indeed this is only a variant of the notation $\{x : \phi_x \in \mathcal{D}\}$ is
immediate from (cf. 2.5.0.3)

$$W_x \overset{\text{Def}}{=} \mathrm{dom}(\phi_x)$$

Thus, setting $C = \{f \in \mathcal{P} : \mathrm{dom}(f) \in C_*\}$, we may rewrite (1) as

$$A = \{x : \phi_x \in C\} \tag{2}$$

$\square$

This new notation is very useful in computability theory. We explore here rewordings of the Rice-related theorems that utilize this new notation.

2.7.2.22 Theorem. *Given a complete index set* $A = \{x : W_x \in C_*\}$—*where*
$C_* \subseteq \mathcal{P}_*$. *If some* $S \in C_*$ *satisfies* $S \subseteq T$, *where* $T \in \mathcal{P}_* - C_*$, *then* $\overline{K} \leq_m A$.

Proof. Following the notational translations of 2.7.2.21, we write $C = \{f \in \mathcal{P} : \mathrm{dom}(f) \in C_*\}$. So,

$$A = \{x : \phi_x \in C\}$$

Let S and T be as given. Thus, for some $f \in C$, $S = \mathrm{dom}(f)$. The function
$g = \lambda x.1 \dot{-} (1 \dot{-} f(x))$ satisfies $S = \mathrm{dom}(g)$ as well. Let $T = \mathrm{dom}(h)$ with $h \in \mathcal{P}$
and $\mathrm{ran}(h) = \{0\}$ [if necessary, we use $\lambda x.1 \dot{-} (1 \dot{-} h(x))$ instead].

Notice that $g \in C$, $g \subseteq h$, and $h \notin C$.[95] By 2.7.2.9, $\overline{K} \leq_m A$. $\square$

2.7.2.23 Theorem. (Rice's Theorem—W-Version) *A compete index set* $A = \{x : W_x \in C_*\}$ *is recursive iff it is* trivial, *meaning that either* $A = \emptyset$ *or* $A = \mathbb{N}$.

Proof. This can be proved by invoking 2.7.2.11 after we translate the W_i-notation
into ϕ_i-notation, using 2.7.2.21. It can be proved just as easily, directly from 2.7.2.22.

---

[95]If $h \in C$, then $\mathrm{dom}(h) \in C_*$ by definition; hence $T \in C_*$.

A trivial index set being primitive recursive, we turn to the *only if*. So let both A and $\overline{A}$ be c.e.

Case where $\emptyset \in \mathcal{C}_*$. Then $S \in \mathcal{C}_*$ for every c.e. subset of $\mathbb{N}$ by $\emptyset \subseteq S$. It follows that $\mathcal{C}_* = \mathcal{P}_*$, i.e., $A = \mathbb{N}$.

Case where $\emptyset \notin \mathcal{C}_*$. We work with the c.e. set $\overline{A} = \{x : W_x \in \mathcal{P}_* - \mathcal{C}_*\}$. As above, $S \in \mathcal{P}_* - \mathcal{C}_*$ for every c.e. subset of $\mathbb{N}$. Thus, $\mathcal{P}_* - \mathcal{C}_* = \mathcal{P}_*$, i.e., $A = \emptyset$.   □

2.7.2.24 Theorem. *Given $A = \{x : W_x \in \mathcal{C}_*\}$, where $\mathcal{C}_* \subseteq \mathcal{P}_*$. Suppose that some $S \in \mathcal{C}_*$ has no finite subset D that is a member of $\mathcal{C}_*$. Then $\overline{K} \leq_m A$.*

Proof. We derive this as a corollary of 2.7.2.18, using the notational translations from 2.7.2.21. We write $\mathcal{C} = \{f \in \mathcal{P} : \mathrm{dom}(f) \in \mathcal{C}_*\}$. So,

$$A = \{x : \phi_x \in \mathcal{C}\}$$

Let S be as given. Thus, for some $f \in \mathcal{C}_*$, $S = \mathrm{dom}(f)$. As before, we may assume that $\mathrm{ran}(f) = \{0\}$. If we had a finite $\xi \subseteq f$ as a member of $\mathcal{C}$, then it would be that

(a) $\mathrm{dom}(\xi)$ is a finite set $D \in \mathcal{C}_*$—by the translation—and

(b) $D \subseteq S$.

(a) and (b) contradict the hypothesis. Thus, no such ξ exists, and thus $\overline{K} \leq_m A$ by 2.7.2.18.   □

2.7.2.25 Example. $\{x : W_x = \emptyset\}$ is not c.e. by 2.7.2.22. Indeed, $\mathcal{C}_* = \{\emptyset\}$. But $\emptyset \subseteq \mathbb{N}$, yet $\mathbb{N} \notin \{\emptyset\}$. $\{x : W_x \text{ is infinite}\}$ is not c.e. by 2.7.2.22. Indeed, the $\mathcal{C}_*$-set here is the set of all infinite *members* of $\mathcal{P}_*$. This $\mathcal{C}_*$ fails to contain any *finite* subset of any set that it contains. We are led to $\overline{K} \leq_m \{x : W_x \text{ is infinite}\}$ by 2.7.2.24.   □

2.7.3 More Dovetailing

A c.e. set can be given either via a URM that is a verifier—the set is the domain of the verifier—or a URM that is an enumerator: the set is the range of the URM-computed function.

Suppose we are given one of these two types of finite description of a set. Can we *construct* the other? Yes!

Say that a semi-computable set S has been given by a verifier M of code m, that is, $S = W_m$. How do we construct a URM program N, of code n to compute ϕ_n, such that

$$\mathrm{ran}(\phi_n) = W_m? \tag{1}$$

The idea is that, *for each* x for which M halts—that is, $x \in W_m$—our new program N must output x. Now, we cannot have N enumerate all $x = 0, 1, 2, 3, \ldots$ and simulate M on each such input in turn, since, on some such inputs, M—and therefore N—will be stuck forever; an unfortunate event as there may well be later x-values that need outputting. So, we dovetail instead! (Cf. 2.5.0.23.) Thus, N will simulate M on all inputs $x = 0, 1, 2, 3, \ldots$ as follows: It will simulate one step of M on input 0; then two steps on inputs 0 *and* 1; then $\ldots$ then $k+1$ steps on *all* inputs

$0, 1, \ldots, k$. It will continue in this pattern, for all $k \geq 0$. All inputs which during the simulation cause M to halt will be output by N. Note that we have "*constructed*" N (in outline), using knowledge of M ("subroutine"). Clearly, the set of outputs generated by N is precisely the set of inputs on which M halts.

Mathematically, the fact that the program n is *constructed* from m is captured by the existence of a primitive recursive h such that $n = h(m)$. We state and prove

2.7.3.1 Theorem. *There is a primitive recursive h such that for all x, $\mathrm{ran}(\phi_{h(x)}) = W_x$.*

Proof. This is a trivial adaptation of the proof of 2.5.0.24 following the idea in 2.5.0.28: Define, for all x and z,

$$\psi(x, z) \simeq \begin{cases} (z)_0 & \text{if } T(x, (z)_0, (z)_1) \\ \uparrow & \text{otherwise} \end{cases} \tag{1}$$

ψ is trivially partial recursive, hence (by S-m-n) there is a primitive recursive h such that

$$\phi_{h(x)}(z) \simeq \begin{cases} (z)_0 & \text{if } T(x, (z)_0, (z)_1) \\ \uparrow & \text{otherwise} \end{cases}$$

By 2.5.0.28, $\mathrm{ran}(\phi_{h(x)}) = W_x$.   □

Pause. Can we ensure that whenever $W_x \neq \emptyset$ we obtain a total $\phi_{h(x)}$?◄

Yes, if we can replace $\uparrow$ in (1) of the above proof by a *computed* "a", analogous to the one employed in 2.5.0.24, where we said

> ... where "a" is some fixed member of S that we keep outputting every time the condition "$T(i, (z)_0, (z)_1)$" fails ...

The "some fixed" a can easily be replaced by a computed value (from the verifier code x of W_x) via the *selection theorem*.

2.7.3.2 Theorem. (Selection Theorem) *There is a partial recursive g of one variable such that, for all x,*

(i) $W_x \neq \emptyset$ *iff* $g(x) \downarrow$

(ii) *If* $W_x \neq \emptyset$, *then* $g(x) \in W_x$.

Proof. A familiar dovetailing technique is at play: We compute the first pair $[y, w]$ such that $T(x, y, w)$ holds. That is, setting $z = [y, w]$, we let

$$g(x) \simeq \Big((\mu z)T(x, (z)_0, (z)_1)\Big)_0 \tag{1}$$

For (i), let $W_x \neq \emptyset$. Thus $(\exists y, w)T(x, y, w)$ is true and the search in (1) succeeds. Conversely, if the search succeeds, then $(z)_0 \in W_x$. For (ii), the successful search yields a $(z)_0 \in W_x$. But $g(x) = (z)_0$.   □

The selection theorem is a computable version of the *axiom of choice*. The axiom of choice postulates that if $\{S_a : a \in I\}$ is an indexed collection of sets—indexed by I—then there is an f such that $S_a \neq \emptyset$ implies $f(a) \in S_a$.

Our "g" does just that for the collection $\{W_x : x \in \mathbb{N}\}$; *computably*!

2.7.3.3 Corollary. *There is a primitive recursive h such that for all x,* $\mathrm{ran}(\phi_{h(x)}) = W_x$. *Moreover, if $W_x \neq \emptyset$, then $\phi_{h(x)} \in \mathcal{R}$.*

Proof. We use the proof of 2.5.0.24, adding the i as an argument, and using $g(i)$—the selection function at i—rather than the intangible "a". That is,

$$\psi(i, z) \simeq \begin{cases} (z)_0 & \text{if } T(i, (z)_0, (z)_1) \\ g(i) & \text{otherwise} \end{cases}$$

ψ is trivially partial recursive, hence (by S-m-n) there is a primitive recursive h such that

$$\phi_{h(i)}(z) \simeq \begin{cases} (z)_0 & \text{if } T(i, (z)_0, (z)_1) \\ g(i) & \text{otherwise} \end{cases}$$

$\square$

2.7.3.4 Example. Rogers (1967) offers the following simple proof that obscures the dovetailing at work: Let

$$\tau(x, z) \simeq \begin{cases} z & \text{if } \phi_x(z) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Via definition by positive cases, τ is partial recursive. Let (by S-m-n) σ—primitive recursive—be such that $\phi_{\sigma(x)}(z) \simeq \tau(x, z)$, for all x, z. Note that $z \in \mathrm{ran}(\phi_{\sigma(x)})$ iff $\phi_x(z) \downarrow$, that is, iff $z \in W_x$.

Of course, the definition-by-positive-cases theorem (2.5.0.30) already includes a dovetailing argument buried in its proof. $\square$

Next, say that a semi-computable set S has been given by an enumeration, by some URM M of code m. That is, $S = \mathrm{ran}(\phi_m)$. How do we construct a URM program N of code n, for ϕ_n, that is a verifier for S, in other words,

$$\mathrm{ran}(\phi_m) = W_n? \tag{2}$$

If ϕ_m is total, then all that N need do is this: Given x, N tests "$x \in S$?" by

> for $y = 0, 1, 2, \ldots$ do
> if M on input y outputs x, then **halt**

For nontotal ϕ_m this will fail, since, say, it could be that $\phi_m(1) = x$ while $\phi_m(0) \uparrow$. The call to M on input 0 would go forever, never reaching the computation $\phi_m(1)$. Once again, one must dovetail:

> for $T = 1, 2, \ldots$ do
> if M on any input $y < T$ outputs x within T steps, then **halt**

Mathematically,

2.7.3.5 Theorem. *There is a primitive recursive k such that for all x, $\mathrm{ran}(\phi_x) = W_{k(x)}$.*

Proof. Define, for all x and y,

$$\chi(x,y) \simeq \begin{cases} 0 & \text{if } (\exists z, w)\big(T(x, z, w) \wedge y = d(w)\big) \\ \uparrow & \text{otherwise} \end{cases}$$

The top predicate is c.e. thus χ is defined by positive cases. This makes χ partial recursive. By the S-m-n theorem, there is a $k \in \mathcal{PR}$ such that $\phi_{k(x)}(y) \simeq \chi(x, y)$, for all x, y. The definition of χ immediately yields that $\phi_{k(x)}(y) \downarrow$ iff $y \in \mathrm{ran}(\phi_x)$. $\quad\square$

2.7.3.6 Corollary. *There are σ and τ in $\mathcal{PR}$ such that, for all x,*

(i) $W_x = \mathrm{ran}(\phi_{\sigma(x)})$ *and, moreover, $\phi_{\sigma(x)}$ is 1-1. If it is nontotal, then $\mathrm{dom}(\phi_{\sigma(x)}) = \{i \in \mathbb{N} : i < n\}$, for some n.*

(ii) $\mathrm{ran}(\phi_x) = \mathrm{ran}(\phi_{\tau(x)})$ *and, moreover, $\phi_{\tau(x)}$ is 1-1. If it is nontotal, then $\mathrm{dom}(\phi_{\tau(x)}) = \{i \in \mathbb{N} : i < n\}$, for some n.*

Proof.

(i) We adapt the proof of 2.7.3.3, being careful not to output values that have already been output. The technique for generating outputs is dovetailing; the technique to avoid repetitions of outputs is to keep a list of "outputs so far", and not output a generated value if it is in the list.

So this time we define a modified ψ via a course-of-values recursion following the careful approach of 2.1.2.50 and 2.1.2.51, albeit in the present application we will not start with, nor write down, the (simple) primitive recursion for the history function
$$H(i, x) = [\psi(i, 0), \dots, \psi(i, x)]$$
In this connection we recall the predicate $x \in y$ used in the proof of 2.4.4.1.

$$\psi(i, 0) \simeq g(i)$$
$$\psi(i, x + 1) \simeq \Big((\mu z)\big(T(i, (z)_0, (z)_1) \wedge \neg(z)_0 \in H(i, x)\big)\Big)_0$$

The S-m-n theorem yields a σ such that $\phi_{\sigma(x)}(y) \simeq \psi(x, y)$, for all x, y. The claim for $\mathrm{dom}(\phi_{\sigma(x)})$ follows from 2.1.2.51. That $W_x = \mathrm{ran}(\phi_{\sigma(x)})$ is immediate.

(ii) By 2.7.3.5 we get an h such that $\mathrm{ran}(\phi_x) = W_{h(x)}$. Applying part (i) we get a σ such that $W_{h(x)} = \mathrm{ran}(\phi_{\sigma(h(x))})$. We set $\tau(x) = \sigma(h(x))$, for all x. $\quad\square$

We continue with a brief exploration of *effectively*—a synonym of *computably*—obtaining the results of computable set-theoretic operations on c.e. sets.

2.7.3.7 Example. There is a primitive recursive h such that $W_{h(x,y)} = W_x \cap W_y$. Herein lies the effectiveness of this operation: If we know the verifiers x and y of two c.e. sets, then we can construct (via h) a verifier $h(x, y)$ for their intersection. Note that $W_x \cap W_y = \mathrm{dom}(\phi_x + \phi_y)$. Fix an i such that, for all x, y and z, we have $\phi_i^{(3)}(x, y, z) \simeq \phi_x(z) + \phi_y(z)$. By S-m-n, $\phi_{S_2^1(i,x,y)}(z) \simeq \phi_i^{(3)}(x, y, z)$. Take $h = \lambda xy.S_2^1(i, x, y)$. $\qquad\square$

2.7.3.8 Example. (2.7.3.6 Revisited) How about proving 2.7.3.6, case (i), by selecting an a in W_i; selecting an a' in $W_i - \{a\}$; selecting an a'' in $(W_i - \{a\}) - \{a'\}$; etc.?

OK, let us organize this idea. First off, we can find a $k \in \mathcal{PR}$ such that $W_{k(x,i)} = W_i - \{x\}$ for all i, x. We do this by noting that $W_i - \{x\} = W_i \cap (\mathbb{N} - \{x\}) = W_i \cap W_{r(x)}$, the last equality by Exercise 2.12.58. Using 2.7.3.7 we have

$$W_{h(i,r(x))} = W_i - \{x\}$$

Our k is $\lambda ix.h(i, r(x))$.

We can now take care of the iteration "etc." above using the selection function g:

We let $a = g(i)$; then remove a to obtain $W_{k(a,i)}$;

We next let $a' = g\Big(k\big(a, i\big)\Big)$; then remove a' to obtain $W_{k(a',k(a,i))}$;

We next let $a'' = g\Big(k\big(a', k(a, i)\big)\Big)$; then remove a'' to obtain $W_{k(a'',k(a',k(a,i)))}$;

etc. We can define an f by recursion,

$$f(0, i) = i$$
$$f(x + 1, i) \simeq k\big(g(f(x, i)), f(x, i)\big)$$

f enumerates the W-sets—verifiers of, that is—

$$W_i = W_{f(0,i)} \supset W_{f(1,i)} \supset W_{f(2,i)} \supset W_{f(3,i)} \supset \cdots$$

while the sequence
$$g(f(0, i)), g(f(1, i)), g(f(2, i)), \ldots$$

1-1-enumerates (the *members* of) W_i, because $g(f(x, i))$—if defined—is not in $W_{f(y,i)}$, for $y > x$, and thus will not be chosen again.

Indeed, by the second equation for f, $f(x + 1, i)$ is what g selects in the original W_i *after* the removal of $g(f(x, i))$ in this very step—the $g(f(x - 1, i)), g(f(x - 2, i)), \ldots, g(f(0, i))$ having been removed in the preceding steps (by an obvious induction).

It is also clear that either $\lambda x.g(f(x, i))$ is total, or an initial segment of $\mathbb{N}$, since the enumeration will not get stuck until W_i is depleted. $\qquad\square$

2.7.3.9 Example. Let A be a c.e. set and f a partial recursive function. What can we conclude about the *inverse image* of A under f, that is, $f_{\leftarrow}(A)$? (Cf. p. 43.)

Well, $x \in f_\leftarrow(A) \equiv (\exists y)(f(x) = y \wedge y \in A)$. Thus, by closure properties of $\mathcal{P}_*$ and 2.5.0.7, the inverse image is c.e.

We can do better than this: From programs for f and A we can construct an enumerator for the inverse image. This is done by yet another dovetailing argument!

Intuitively, we dovetail the "enumeration" f: Enumerate all triples $\langle x, y, z \rangle$ in some systematic way. For example, as $\langle (w)_0, (w)_1, (w)_2 \rangle$, for $w = 0, 1, 2, \ldots$

For each $\langle x, y, z \rangle$ generated, we do z steps of the verification of $f(x) = y \wedge y \in A$ using the given program for f and the verifier for "$y \in A$?"

If we get a "yes" then we enumerate x into $f_\leftarrow(A)$.

Mathematically this is even simpler, although the dovetailing is hidden. Note that $z \in (\phi_x)_\leftarrow(W_y) \equiv \phi_x(z) \in W_y$. But $\phi_x(z) \in W_y \equiv \phi_y(\phi_x(z)) \downarrow$. By S-m-n applied to $\psi = \lambda xyz.\phi_y(\phi_x(z))$ we have an $h \in \mathcal{PR}$ such that $\psi(x, y, z) \simeq \phi_{h(x,y)}(z)$, for all x, y, z. That is, $\mathrm{dom}(\phi_{h(x,y)}) = (\phi_x)_\leftarrow(W_y)$, or $W_{h(x,y)} = (\phi_x)_\leftarrow(W_y)$. $\qquad\square$

2.7.3.10 Example. Let us address the same question as in 2.7.3.9, but for the forward image $f_\rightarrow(A) = \{f(x) : x \in A\}$ for c.e. A and partial recursive f. First off, choose a verifier ϕ_y for A. Then we can dovetail as follows: Enumerate systematically, as in the preceding example, all triples $\langle x, w, z \rangle$. For each triple generated as a number $u = [x, w, z]$, check whether in z steps we have $f(x) \downarrow$ while we have $\phi_y(x) \downarrow$ in w steps; if so, print $f(x)$.

Mathematically, let $A = \mathrm{dom}(\phi_y)$ and $f = \phi_x$. Thus, we can enumerate $f_\rightarrow(A)$ by

$$\psi(x, y, u) \simeq \begin{cases} d((u)_2) & \text{if } T(y, (u)_0, (u)_1) \wedge T(x, (u)_0, (u)_2) \\ \uparrow & \text{otherwise} \end{cases}$$

By S-m-n, there is a σ in $\mathcal{PR}$ such that for all x, y, u, we have $\phi_{\sigma(x,y)}(u) \simeq \psi(x, y, u)$. Clearly, $\mathrm{ran}(\phi_{\sigma(x,y)}) = (\phi_x)_\rightarrow(W_y)$. $\qquad\square$

We have so many times remarked that the "dovetailing is hidden" or "obscured". In a way this is a deliberate outcome of the tools of this subsection—2.7.3.1, 2.7.3.5, and 2.7.3.2 along with their variants and corollaries—which "hardwire" dovetailing arguments into themselves, making it often unnecessary to use such an argument explicitly.

Here is another example that verifies the above point.

2.7.3.11 Example. Let us revisit the "hard direction" of 2.5.0.7, that is, *if $y = f(\vec{x}_n)$ is semi-recursive, then f is partial computable*. We utilize the *selection theorem* as it was generalized in Exercise 2.12.61. By assumption, for some i,

$$y = f(\vec{x}_n) \equiv \phi_i^{(n+1)}(y, \vec{x}_n) \downarrow$$

Then $f = \lambda \vec{x}_n.Sel^{(n+1)}(i, \vec{x}_n)$. $\qquad\square$

2.7.4   Recursive Enumerations

The definition of the term c.e. (r.e.) was via a recursive enumeration f for non-empty such sets. It turns out that we can say a bit more if the c.e. set is recursive.

2.7.4.1 Definition. A total function $f : \mathbb{N} \to \mathbb{N}$ is *increasing* iff, for all x and y, we have that $x < y$ implies $f(x) \leq f(y)$. It is *strictly increasing* iff, for all x and y, we have that $x < y$ implies $f(x) < f(y)$. □

2.7.4.2 Theorem. *A non-empty recursive set A has an increasing recursive enumerating function, and conversely: If the c.e. set B has an increasing recursive enumerating function, then it is recursive.*

Proof. Let A be as stated. If it is *finite*, with distinct members in the following increasing order,

$$a_0, a_1, \ldots, a_k$$

then

$$f(x) = \begin{cases} a_0 & \text{if } x = 0 \\ a_1 & \text{if } x = 1 \\ \ldots & \ldots \\ a_{k-1} & \text{if } x = k - 1 \\ a_k & \text{if } x \geq k \end{cases}$$

is primitive recursive, increasing, and $\mathrm{ran}(f) = A$.

Assume now that A is *infinite*. Define a function h by

$$h(0) = a$$
$$h(x + 1) = (\mu y)\big(y \in A \wedge y > h(x)\big)$$

where a is the smallest member of A. Thus, $h \in \mathcal{P}$, and since the search succeeds for all x, we have $h \in \mathcal{R}$. It is clear that h is strictly increasing and that $\mathrm{ran}(h) \subseteq A$. Is it possible that $m \in A - \mathrm{ran}(h)$? If so, let m be smallest such. Now, $m \neq h(0) = a$. Let then $h(y) < m < h(y + 1)$ for the appropriate y [which is such that $y + 1$ is smallest with $h(y + 1) > m$, a minimum that exists since h is strictly increasing and thus its outputs increase without bound]. But then, $h(y + 1)$ is wrongly chosen, since it is not the smallest $> h(y)$ in A! Therefore we can have no such m, and $A = \mathrm{ran}(h)$.

For the converse, let $B = \mathrm{ran}(f)$, where f is a recursive increasing function. If B is finite, then it is recursive—indeed primitive recursive—since in predicate form it is a disjunction of a finite number of elementary formulae of the type $x = a$, one for each $a \in B$.

Let then B be infinite. We need to show that the predicate $x \in B$ is recursive. Well, B is the increasing "array"

$$f(0), f(1), \ldots, f(i), \ldots$$

How do we search such an array for x with a view of finding the first i, *if any*, such that $x = f(i)$? Our programming experience suggests the pseudo program

$$i \leftarrow 0$$
$$\text{while} \quad x > f(i)$$
$$i \leftarrow i + 1$$
$$\text{end while}$$
$$\text{if} \quad x = f(i) \text{ then print } i$$
$$\text{else print "not found"}$$

This "proves"—modulo rewriting the program as a URM—that B is recursive. We can easily turn the above to a mathematical proof.

$$x \in B \equiv f\big((\mu i)x \le f(i)\big) = x$$

Since B is infinite, the partial recursive function $\lambda x.(\mu i)x \le f(i)$ is recursive. Thus, so is $\lambda x. f\big((\mu i)x \le f(i)\big)$. □

We have at once:

2.7.4.3 Corollary. *An infinite recursive set A has a strictly increasing recursive enumerating function, and conversely: If the c.e. set B has a strictly increasing recursive enumerating function, then it is recursive and infinite.*

2.7.4.4 Corollary. *An infinite c.e. set A has an infinite recursive subset.*

Proof. Let $A = \text{ran}(f)$ for recursive f. We define a strictly increasing (recursive) sub-sequence

$$g(0), g(1), \dots$$

of

$$f(0), f(1), \dots \tag{1}$$

and then invoke 2.7.4.3. Note that because A is infinite the sequence (1) is unbounded hence, for any x, there is a y such that $g(x) < f(y)$.

This leads to this primitive recursion for g:

$$g(0) = f(0)$$
$$g(x + 1) = f\big((\mu y)f(y) > g(x)\big)$$

Since the search above always succeeds, the partial recursive g is total, hence recursive. But it is also, by construction, strictly increasing. □

Note that the above proof was constructive in that from the knowledge of f we "programmed" g. This can be made more precise mathematically as follows.

2.7.4.5 Corollary. *There is a primitive recursive r such that if ϕ_x is recursive and $\text{ran}(\phi_x)$ is infinite, then $\phi_{r(x)}$ is recursive, $\text{ran}(\phi_{r(x)})$ is recursive and infinite, and, moreover, $\text{ran}(\phi_{r(x)}) \subsetneq \text{ran}(\phi_x)$.*

Proof. A straightforward adaptation of the proof above. Define instead

$$g(0, x) \simeq \phi_x(0)$$
$$g(y + 1, x) \simeq \phi_x\big((\mu z)\phi_x(z) > g(y, x)\big)$$

As above, the recursiveness and unboundedness of ϕ_x means that g is recursive and strictly increasing with respect to the y variable. S-m-n gives us an r such that, for all x, y, we have $\phi_{r(x)}(y) \simeq g(y, x)$. $\qquad\square$

2.7.4.6 Corollary. *There is a primitive recursive h such that if W_x is infinite, then $W_{h(x)} \subseteq W_x$, and $W_{h(x)}$ is infinite and recursive.*

Proof. A direct consequence of the proof of 2.7.4.5: Employ first 2.7.3.3 to obtain a $\sigma \in \mathcal{PR}$ such that $\mathrm{ran}(\phi_{\sigma(x)}) = W_x$ and where $W_x \neq \emptyset$ implies $\phi_{\sigma(x)} \in \mathcal{R}$. Now apply Corollary 2.7.4.5 to obtain $\phi_{r(\sigma(x))}$. Finally, apply 2.7.3.5 to obtain a primitive recursive h such that $W_{h(x)} = \mathrm{ran}(\phi_{r(\sigma(x))})$. $\qquad\square$

2.7.4.7 Remark. The "programs" r and h in Corollaries 2.7.4.5 and 2.7.4.6 do not simply exist; we demonstrated their existence by *constructing* them. We say that the proofs were *constructive*. Constructive proofs that "a program exists" are not always possible. For example, we know from the proof of 2.7.4.2 that for any finite, non-empty set, there is an algorithm that can enumerate it in strict ascending order and then it will "taper off" repeating (the enumeration of) the maximum entry forever.

It turns out that *given* a finite set we *cannot* construct said algorithm.[96] We will demonstrate the particulars of this claim in the following discussion. $\qquad\square$

Pause. But how is a finite set "*given*"? ◄

Well, a recursive set S can be given by a *decider* for S—that is, an x, such that $\phi_x = \chi_S$—or by a *verifier* for S—that is, a z, such that $W_z = S$. Note that giving a set as an *enumerator* is computably equivalent to giving it as a verifier, since one can go back and forth between these two representations computably (cf. 2.7.3.1 and 2.7.3.5).

2.7.4.8 Example. We have already seen that we can convert a decider for a set S to a verifier for the same set (2.5.0.12). The process described was clearly constructive, albeit at an informal level.

Mathematically, let

$$\psi(x, y) \simeq \begin{cases} 0 & \text{if } \phi_x(y) = 0 \\ \uparrow & \text{otherwise} \end{cases} \tag{1}$$

---

[96]This statement is dependent on how the finite set is "given", as the reader will see shortly (cf. 2.7.4.11–2.7.4.14).

As (1) is an instance of definition by positive cases, ψ is computable, so, by S-m-n, let $h \in \mathcal{PR}$ be such that $\psi(x, y) \simeq \phi_{h(x)}(y)$ for all x, y. Thus

$$\phi_{h(x)}(y) \downarrow \equiv \phi_x(y) = 0 \tag{2}$$

If now $\phi_x = \chi_S$ for some S (rendering S recursive), then, by (2), $W_{h(x)} = S$.   □

For recursive sets, can we go, computably, from a verifier to a decider representation? No, for one way to go about it would be to seek a $\psi \in \mathcal{P}$ such that it satisfies

- $\psi(x) \downarrow$ iff W_x is recursive

  and

- in case W_x is recursive, $\phi_{\psi(x)} = \chi_{W_x}$.

The first bullet requires $\mathrm{dom}(\psi) = \{x : W_x \in \mathcal{R}_*\}$, which cannot be, since the left hand side is c.e. but the right hand side is not (cf. Exercise 2.12.48). No such ψ exists.

2.7.4.9 Remark. Hmm. What about weakening ψ so that we allow it to converge as it pleases outside $\{x : W_x \in \mathcal{R}_*\}$, yielding irrelevant answers? That is, can we have a "new" ψ of which we only ask:

$$\text{if } W_x \text{ is recursive, then } \psi(x) \downarrow \text{ and } \phi_{\psi(x)} = \chi_{W_x} \tag{1}$$

This is also impossible [Rogers (1967)]. To see this, we go back to Theorem 2.7.2.7 and reuse $(* * *)$ (reproduced below)

$$\phi_{h(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \downarrow \\ \emptyset & \text{if } \phi_x(x) \uparrow \end{cases}$$

from the proof of its item (1), rewriting it in this guise:

$$W_{h(x)} = \begin{cases} \mathbb{N} & \text{if } x \in K \\ \emptyset & \text{if } x \in \overline{K} \end{cases} \tag{2}$$

Of course, h is primitive recursive. Suppose now that we have this new, weaker ψ that we are looking for. Then,

(a) $\psi(h(x)) \downarrow$, for all x, since either way, "top" or "bottom", $W_{h(x)}$ is recursive.

(b) Since $\mathrm{ran}(\chi_{\mathbb{N}}) = \{0\}$, we have that $1 \in \mathrm{ran}(\phi_{\psi(h(x))})$ iff we have the bottom case in (2).

Thus,

$$1 \in \mathrm{ran}(\phi_{\psi(h(x))}) \equiv x \in \overline{K}$$

This is untenable, since one side of $\equiv$ is c.e. but the other is not. Indeed, $1 \in \text{ran}(\phi_{\psi(h(x))}) \equiv (\exists y)(\exists z)\big(T(\phi_{\psi(h(x))}, y, z) \wedge d(z) = 1\big)$ □

Of course, the foregoing comments also apply to finite sets since these are (primitive) recursive. However, finiteness provides yet another manner to finitely code such a set, not coding it via a decider or verifier, but rather coding the *set of its elements* itself by a single number.

This can be done in any one of the many ways that we have at our disposal for coding finite *sequences*, for example, prime-power coding (allowing 0 to be the code for the empty set). As this entails a decision on selecting one of the $n!$[97] orders of a set of n elements, the following coding is more elegant.

2.7.4.10 Definition. (Canonical Indices) The *canonical index* of a finite set $S = \{a_0, \dots, a_n\}$—where the a_i are distinct but not otherwise sorted in any "preferred" order—is the number $u = 2^{a_0} + 2^{a_1} + \cdots + 2^{a_n}$.

The canonical index of $\emptyset$ is 0. □

u—expressed in binary notation—has a digit ("bit") 1 in precisely the positions a_i, for $i = 0, \dots, n$ (where position-0 is the rightmost or "least significant").

2.7.4.11 Example. It is rather trivial to see that if we have a canonical index u of a finite set S, then we can *construct* a program for its characteristic function, and therefore —by 2.7.4.8—can also construct a program for a verifier of S.

Indeed, if $u = 0$, then the characteristic function is $\lambda x.1$. Alternatively, once *we have recovered* from u (by just looking where its binary notation has 1s) *the members*

$$\{b_0, \dots, b_m\}$$

of S, the characteristic function—and indeed the enumerator of p. 202—can be *constructed* at once:

$$\chi_S(x) = \begin{cases} 0 & \text{if } x = b_0 \\ 0 & \text{if } x = b_1 \\ \vdots & \\ 0 & \text{if } x = b_m \\ 1 & \text{otherwise} \end{cases}$$

□

The converse will not work:

2.7.4.12 Example. We cannot (in general) have a computable function ψ such that if W_x is finite, then $\psi(x) \downarrow$ and $\psi(x)$ equals the canonical index of W_x, while, if W_x is not finite, then $\psi(x) \uparrow$.

---

[97]$0! = 1$ and $(n+1)! = n! \times (n+1)$.

Indeed, if we had such a ψ, then $\mathrm{dom}(\psi) = \{x : W_x \text{ is finite}\}$.

This is impossible, as the left hand side is c.e. but the right hand side is not (cf. Exercise 2.12.50).   □

2.7.4.13 Example. If a finite set S is given by its characteristic function, ϕ_x, then we have more information about S than we would have if it were given as a W_z. Does this extra information allow us to have a computable function ξ such that if $\{y : \phi_x(y) = 0\}$ is finite, then $\xi(x) \downarrow$ and $\xi(x)$ equals the canonical index of $\{y : \phi_x(y) = 0\}$; while, if $\{y : \phi_x(y) = 0\}$ is not finite, then $\xi(x) \uparrow$?

The answer is still negative, for consider the following definition by recursive cases

$$f(x, y) = \begin{cases} 1 & \text{if } \neg\Phi_x(x) \leq y \\ 0 & \text{otherwise} \end{cases}$$

$f \in \mathcal{R}$ and thus, by S-m-n, we have a $\sigma \in \mathcal{PR}$, such that

$$f(x, y) = \phi_{\sigma(x)}(y), \text{ for all } x, y$$

Note that if $\phi(x) \uparrow$, then the top condition is always true, thus, using *sequence-notation*

$$\phi_{\sigma(x)} = \underbrace{111\ldots}_{\text{1s forever}},$$

that is,

$$x \in \overline{K} \text{ implies that } \phi_{\sigma(x)} = \chi_\emptyset \tag{1}$$

If, on the other hand, $\phi_x(x) \downarrow$, let $y = y_x$ be smallest such that $\Phi_x(x) \leq y$. Thus,

$$\phi_{\sigma(x)} = \underbrace{111\ldots 1}_{y_x - 1 \text{ 1s}}\underbrace{000\ldots}_{\text{0s forever}}, \text{ in this case}$$

Thus, $\phi_{\sigma(x)}$ is the characteristic function of an infinite set in the $x \in K$ case, and hence $\xi(\sigma(x)) \uparrow$. Therefore, assuming we have our ξ as stated, we also have

$$\xi(\sigma(x)) \downarrow \equiv x \in \overline{K}$$

This will not do, since the left hand side is a c.e. relation, while the right hand side is not c.e.   □

2.7.4.14 Example. Perhaps, if we weaken the requirement on ξ, we may have a computable passage from the program of the characteristic function of a finite set to its canonical index.

Thus, let us *relax* the requirement that if $\{y : \phi_x(y) = 0\}$ is not finite, then we are "informed" of this by "$\xi(x) \uparrow$". Since we cannot computably know whether $\{y : \phi_x(y) = 0\}$ is not finite, we lose nothing if we allow ξ to converge for such x (cf. the analogous situation in 2.7.4.9).

So, *can* we have the validity of the weaker statement below?

"A computable ξ exists such that if $\{y : \phi_x(y) = 0\}$ is finite, then $\xi(x) \downarrow$ and $\xi(x)$ equals the canonical index of $\{y : \phi_x(y) = 0\}$."

Thus, we are willing to live with a much less informative version of "ξ", where the statement "$\{y : \phi_x(y) = 0\}$ is not finite" may well coexist with $\xi(x) \downarrow$. Such a ξ still would always correctly build the canonical index of a set S if it received as input a program x that computes the characteristic function of S, but would produce nonsensical outputs for some x that satisfy "$\{y : \phi_x(y) = 0\}$ is not finite".

It turns out that we cannot have this version of ξ either! Indeed, define

$$g(x, y) = \begin{cases} 0 & \text{if } y = \min\{z : \Phi_x(x) \le z\} \\ 1 & \text{otherwise} \end{cases}$$

Note that

$$y = \min\{z : \Phi_x(x) \le z\} \equiv \Phi_x(x) \le y \wedge (\forall z)_{<y} \neg \Phi_x(x) \le z$$

hence both defining conditions (for g) are recursive. Thus g is recursive. Let σ in $\mathcal{PR}$ be such that, for all x and y, we have $\phi_{\sigma(x)}(y) = g(x, y)$. Thus,

$$\text{if } \phi_x(x) \uparrow, \text{ then } \phi_{\sigma(x)} = \lambda x.1\text{—that is, } \chi_\emptyset$$

and

$$\text{if } \phi_x(x) \downarrow, \text{ say, at the earliest in } y_x \text{ steps, then } \phi_{\sigma(x)} = \chi_{\{y_x\}}$$

Since the canonical index of $\emptyset$ is 0, we have

$$0 = \xi(\sigma(x)) \equiv x \in \overline{K}$$

contradicting the non semi-recursiveness of $\overline{K}$. So, no such ξ exists. $\square$

We now return to the question that motivated all this discussion. We noted that for any finite set *there exists* an algorithm, that of p. 202, that enumerates the set in strict ascending order and once it reaches *the maximum element* it keeps outputting said element forever.

We claimed that this algorithm's existence cannot be proved constructively, that is, there is no computable process that constructs said algorithm for the arbitrarily *given* finite set S.

Indeed, say, S is given by a program (ϕ-index) for its *characteristic function*—for a decider. If we can construct its *enumerator* from this information, then we will obtain knowledge of all the members of S by running the enumerator. From this we can construct the canonical index of S, contradicting the preceding example.

Next, say, S is given as a W_x. If we can construct from this information its enumerator, *in the form given on p. 202*, then we could also construct it from a program for its characteristic function, since the latter can be algorithmically obtained from x (2.7.4.8). This cannot be. Trivially, if S is "given" by its canonical index, then the enumerator, a verifier, and a decider for S can be constructed.

Pause. Hmm. In 2.7.3.6 we saw that there is a constructive way, σ, that takes us from W_x to a 1-1 $\phi_{\sigma(x)}$ such that $W_x = \text{ran}(\phi_{\sigma(x)})$ and, if $\phi_{\sigma(x)}$ is nontotal, then its domain is $\{i : i < n\}$, for some n. So, why not use this $\phi_{\sigma(x)}$ to *determine* the elements

$$\phi_{\sigma(x)}(0), \ldots, \phi_{\sigma(x)}(n-1)$$

of the finite set W_x—if indeed it is finite?◄

2.8 PRODUCTIVE AND CREATIVE SETS

Some non r.e. sets S are, in a way of speaking, "effectively non r.e." in that we have an algorithmic way to refute their r.e.-ness in the context of any claim of the form "$W_x = S$". That is, we can *construct* a counterexample $m \in S - W_x$. Such sets were called *productive* by Dekker.

2.8.0.15 Definition. A set S is *productive* with a *productive function* $f \in \mathcal{R}$ if, for all x, whenever we have $W_x \subseteq S$ then $f(x) \in S - W_x$. □

Clearly, every productive set S is not c.e. for if we think otherwise, then $S = W_m$ for some m. But then $f(m) \in S - W_m$—where f is a given productive function for S—and this yields a contradiction.

2.8.0.16 Example. $\overline{K}$ is productive with productive function $\lambda x.x$. Indeed, let $W_x \subseteq \overline{K}$. Note that this entails that $x \in W_x$ is false, for it says $\phi_x(x) \downarrow$, that is, $x \in K$, which is incompatible with the assumption.

Thus, $x \notin W_x$. But this says $x \in \overline{K}$ so, together, $x \in \overline{K} - W_x$. □

We can discover more productive sets via m-reducibility.

2.8.0.17 Theorem. *If $A \leq_m B$ and A is productive, then so is B.*

Proof. Let $A \leq_m B$ be via g and let f be a productive function for A. We will construct a productive function for B.

So let $W_x \subseteq B$. Thus $g_{\leftarrow}(W_x) \subseteq g_{\leftarrow}(B)$. By 2.7.3.9, there is an $h \in \mathcal{PR}$ such that $W_{h(x)} = g_{\leftarrow}(W_x)$ for all x. Thus, we have $W_{h(x)} \subseteq A = g_{\leftarrow}(B)$. Since f is a productive function for A, we have $f(h(x)) \in g_{\leftarrow}(B) - W_{h(x)}$, that is,

$$f(h(x)) \in g_{\leftarrow}(B) - g_{\leftarrow}(W_x) \tag{1}$$

Since $g_{\leftarrow}(X - Y) = g_{\leftarrow}(X) - g_{\leftarrow}(Y)$ (cf. 1.8.19), (1) yields $g(f(h(x))) \in B - W_x$. Thus, $\lambda x.g(f(h(x)))$ is a productive function for B. □

2.8.0.18 Example. Thus, since we know that $\overline{K} \leq_1 \{x : \phi_x \in \mathcal{R}\}$, $\overline{K} \leq_1 \{x : W_x \text{ is finite}\}$, and $\overline{K} \leq_1 \{x : W_x \text{ is infinite}\}$, all of $\{x : \phi_x \in \mathcal{R}\}$, $\{x : W_x \text{ is finite}\}$, and $\{x : W_x \text{ is infinite}\}$ are productive. □

A concept closely related to that of productive sets is that of creative sets, due to Post.

2.8.0.19 Definition. A c.e. set with a productive complement is called *creative*.   □

A creative set cannot be recursive, since its complement is not c.e.

It turns out that the set of theorems of theories such as Peano arithmetic is creative[98] and thus has no deciders. This prompted Post to choose the name "creative" for such sets since it takes more than a mechanical process to decide theoremhood for such theories.

2.8.0.20 Example. K is creative, since it is c.e. and $\overline{K}$ is productive.   □

2.8.0.21 Corollary. *If A is creative and $A \leq_m B$, then $\overline{B}$ is productive.*

Proof. Let f effect the reducibility. Then $\overline{A} \leq \overline{B}$ (negating both sides of $x \in A \equiv f(x) \in B$). Now apply 2.8.0.17.   □

2.8.0.22 Theorem. *For any productive set A and any $W_x \subseteq A$ there is an infinite W_y such that $W_y \subseteq A$ and $W_x \cap W_y = \emptyset$.*

Proof. At the intuitive level this is obvious, for given a productive f for A we can build W_y by explicit enumeration: Set $a_0 = f(x)$. Assuming that distinct $a_0, \ldots, a_n$ have been enumerated, let z be a semi-index for $W_x \cup \{a_0, \ldots, a_n\}$—a c.e. set by closure properties. Accepting for now that z can be computed from x and $a_0, \ldots, a_n$, we set $a_{n+1} = f(z)$.

Now for the mathematical formalities: We will organize inductively the above loosely described process, building two related sequences

$$W_{z_0}, W_{z_1}, W_{z_2}, \ldots, W_{z_n}, \ldots$$

and

$$a_0, a_1, a_2, \ldots, a_n, \ldots$$

such that

- $a_i \in A - W_{z_i}$, using $a_i = f(z_i)$

  and

- $W_{z_{i+1}} = W_{z_i} \cup \{a_i\}$,

  each for all i, where $z_0 = x$.

---

[98] Where this set is converted to a set of numbers, as we show how in Section 2.11.

From Exercises 2.12.59 and 2.12.60 we know that we have primitive recursive q and k such that, for all i,

$$W_{z_{i+1}} = W_{k\left(z_i, q\left(f(z_i)\right)\right)}$$

Thus the recursive function that builds the sequence z_i (as a function also of x) is

$$\begin{cases} g(0, x) & = x \\ g(i+1, x) & = k\Big(g(i, x), q\big(f(g(i, x))\big)\Big) \end{cases}$$

Assuming $W_x \subseteq A$, the set "W_y" that we set out to build is

$$f\big(g(0, x)\big), f\big(g(1, x)\big), f\big(g(2, x)\big), \ldots, f\big(g(n, x)\big), \ldots$$

that is, $\mathrm{ran}\big(\lambda n.f\big(g(n, x)\big)\big)$. Of course, this range is infinite as the enumeration is 1-1. By the second bullet, $W_x \subseteq W_{z_i}$, for all i, thus, by the first bullet, the built set is indeed disjoint from W_x. □

The above proof is *constructive*. Given x and the productive function f (the latter by a ϕ-index) we can construct the set W_y of the theorem, i.e., can *compute* the y using 2.7.3.5.

Also note that if $W_x \not\subseteq A$, then, even though g still enumerates a c.e. set $\mathrm{ran}\big(\lambda n.f\big(g(n, x)\big)\big)$, this set does not have the described properties. For example, $f(x)$ may not even be in A.

2.8.0.23 Corollary. *A productive set has an infinite c.e. subset. Moreover, this subset can be constructed from a phi-index of the given productive function.*

Proof. Take $W_x = \emptyset$. □

2.8.0.24 Corollary. *A productive set has an infinite recursive subset. Moreover, this subset can be constructed from a phi-index of the given productive function.*

Proof. By 2.7.4.4. □

2.8.0.25 Example. Let A be c.e. and $A \cap B = C$, where C is productive. Then B is productive.

To see this let us construct a productive function for B from a productive function f for C and a semi-index e of A.

So, let $W_x \subseteq B$. Thus

$$A \cap W_x \subseteq C \tag{1}$$

By 2.7.3.7, we have an h in $\mathcal{PR}$ such that $A \cap W_x = W_{h(x)}$, for all x. By (1), $W_{h(x)} \subseteq C$, thus $f(h(x)) \in C - W_{h(x)}$ and therefore

$$f(h(x)) \in A \cap B - A \cap W_x \tag{2}$$

In particular, $f(h(x)) \in A$, thus, from (2), $f(h(x)) \in B - W_x$. We found a productive function for B. □

2.9   THE RECURSION THEOREM

The *recursion theorem*, by that name, is due to Kleene but owes its existence (and proof) to the work of Gödel (1931) on the incompleteness phenomenon. We look at a few versions of this very powerful tool in what follows, and then come back to this comment regarding the connection with Gödel's work.

2.9.0.26 Theorem. (Kleene's Recursion Theorem) *If $\lambda z \vec{x}.f(z, \vec{x}_n) \in \mathcal{P}$, then for some e, we have $\phi_e^{(n)}(\vec{x}_n) \simeq f(e, \vec{x}_n)$, for all $\vec{x}_n$.*

Proof. Let $\phi_a^{(n+1)} = \lambda z \vec{x}_n.f(S_1^n(z, z), \vec{x}_n)$. Then

$$f(S_1^n(a, a), \vec{x}_n) \simeq \phi_a^{(n+1)}(a, \vec{x}_n)$$

$$\simeq \phi_{S_1^n(a,a)}^{(n)}(\vec{x}_n) \qquad \text{by 2.6.0.38}$$

Take $e = S_1^n(a, a)$. $\square$

2.9.0.27 Corollary. (Recursion Theorem—Rogers's version) *If $\lambda x.g(x)$ is recursive. then there is an e such that $\phi_{g(e)} = \phi_e$.*

This *does not* say that $g(e) = e$. Rather, that the two programs $g(e)$ and e compute the same function.

Proof. Let $f(x, y) \simeq \phi_{g(x)}(y)$, for all x, y. Since $f \in \mathcal{P}$, 2.9.0.26 applies to yield an e such that $f(e, y) \simeq \phi_e(y)$, for all y. $\square$

2.9.0.28 Remark. It is instructive to see Rogers's direct proof for the corollary: Define ψ by

$$\psi(x, y) \simeq \phi_{\phi_x(x)}(y) \qquad (1)$$

By the normal form theorem, $\psi \in \mathcal{P}$. Let h be obtained by S-m-n such that $\psi(x, y) \simeq \phi_{h(x)}(y)$, for all x, y. Let a be such that $(gh) = \phi_a$. By (1) we now get (for all y)

$$\phi_{h(a)}(y) \simeq \psi(a, y) \simeq \phi_{\phi_a(a)}(y) \simeq \phi_{g(h(a))}(y)$$

$e = h(a)$ works.

     Note that, in (1), $\phi_x(x)$ is not a ϕ-index unless $\phi_x(x) \downarrow$, but this is irrelevant to the proof, in which we applied the S-m-n theorem to the computable function ψ given—in detail—by

$$\lambda x y.d\Big((\mu z)T\Big(d((\mu w)T(x, x, w)), y, z\Big)\Big)$$

     $\square$

2.9.0.29 Corollary. (Recursion Theorem with Parameters)
     If $\lambda z \vec{y}_m \vec{x}_n.f(z, \vec{y}_m, \vec{x}_n) \in \mathcal{P}$, then there is a 1-1 primitive recursive function h such that $f(h(\vec{y}_m), \vec{y}_m, \vec{x}_n) \simeq \phi_{h(\vec{y}_m)}^{(n)}(\vec{x}_n)$, for all $\vec{y}_m, \vec{x}_n$.

Proof. Let $F \overset{\text{Def}}{=} \lambda z \vec{y}_m \vec{x}_n . f(S^n_{m+1}(z, z, \vec{y}_m), \vec{y}_m, \vec{x}_n)$ and $a \in \mathbb{N}$ be such that $F = \lambda z \vec{y}_m \vec{x}_n . \phi^{(n+m+1)}_a (z, \vec{y}_m, \vec{x}_n)$. Thus,

$$f(S^n_{m+1}(a, a, \vec{y}_m), \vec{y}_m, \vec{x}_n) \simeq \phi^{n+m+1}_a (a, \vec{y}_m, \vec{x}_n) \overset{\text{by S-m-n}}{\simeq} \phi_{S^n_{m+1}(a,a,\vec{y}_m)}(\vec{x}_n)$$

Just take $h = \lambda \vec{y}_m . S^n_{m+1}(a, a, \vec{y}_m)$ and note that the S-m-n functions are 1-1. $\square$

2.9.0.30 Corollary. (Recursion Theorem with Parameters—Rogers Style)
For any $g \in \mathcal{R}$ of $m + 1$ arguments, there is a 1-1 primitive recursive function h such that $\phi^{(n)}_{g(h(\vec{y}_m), \vec{y}_m)} = \phi^{(n)}_{h(\vec{y}_m)}$, for all $\vec{y}_m$.

Proof. $f = \lambda z \vec{y}_m \vec{x}_n . \phi^{(n)}_{g(z, \vec{y}_m)}(\vec{x}_n)$ is in $\mathcal{P}$. Now apply 2.9.0.29. $\square$

The reader has noted no doubt that the proofs of all versions of the recursion theorem are constructive. For example, in the proof of 2.9.0.30, if g is given as $\phi^{(m+1)}_w$, then we use a new f: $f = \lambda z w \vec{y}_m \vec{x}_n . \phi^{(n)}_{\phi^{(m+1)}_w (z, \vec{y}_m)}(\vec{x}_n)$ and, correspondingly, the new h will be an S-m-n function dependent on w as well, yielding for all $w, \vec{y}_m, \vec{x}_n$,

$$\phi^{(n)}_{\phi^{(m+1)}_w (h(w, \vec{y}_m), \vec{y}_m)} = \phi^{(n)}_{h(w, \vec{y}_m)}$$

2.9.0.31 Remark. (Indebtedness to Gödel (1931)) The proof that Gödel gave in loc. cit. to his *first incompleteness theorem* was based on a modification to the *liar's paradox* (the latter due to the Cretan philosopher Epimenides). Epimenides is credited with the sentence "All Cretans are liars". But then, him being a Cretan, the statement must be false, so there is at least one Cretan who is not a liar. This is rather unsettling since by virtue of him simply making this utterance he forced the existence of truthful Cretans! A more unsettling version is the statement "I am lying", for, if I am, then I am not, for my statement is false; if I am not, then I am, for my statement is true. Gödel built an analogous sentence $\mathscr{S}$ within Peano arithmetic, one that states its own unprovability: "I am not a theorem" (of Peano arithmetic).

Intuitively—and based on the fact that Peano arithmetic cannot prove false statements—such a sentence is neither provable, nor is its negation: Indeed, if Peano arithmetic can prove $\mathscr{S}$, then it has just proved a false statement. So $\mathscr{S}$ cannot be a theorem. But then it is true, as it says just that! This makes $\neg \mathscr{S}$ false, therefore it cannot be proved either.

But let us get back on track. Gödel built within Peano arithmetic a so-called *provability predicate*, $\Theta(x)$, analogous to the Kleene predicate. $\Theta(x)$ says "the formula *coded* by the number x is a theorem".

Pause. By the way, in honor of Gödel, we call such formula-codes "Gödel numbers".◀

Gödel also devised in his paper a primitive recursive *substitution function* of two variables, $s(a, b)$, which computes the (new) Gödel number of a formula obtained

from $\mathscr{F}(x)$—the latter of Gödel number a—if we replace x by b. *This is an S-m-n function, but in the context of formulae.*

Next, he proved his *diagonalization lemma*, that for any formula $\mathscr{F}(x)$, there is an e such that e is the Gödel number of the sentence $\mathscr{F}(e)$.

He did this as follows: Let us consider the new formula $\mathscr{F}(s(x,x))$ of Gödel number, say, a. But then, the Gödel number of $\mathscr{F}(s(a,a))$ is $s(a,a)$. Take $e = s(a,a)$. Now compare with the statement and proof of 2.9.0.26!

To conclude this discussion, apply the diagonalization lemma to $\neg\Theta(x)$ to find an m such that m is the Gödel number of $\neg\Theta(m)$. So the latter says, "m is not a theorem", that is, "I am not a theorem". □

2.9.1 Applications of the Recursion Theorem

2.9.1.1 Theorem. (Rice's Theorem—Revisited) *A complete index set is recursive iff it is trivial.*

Proof. [The idea of this proof is attributed in Rogers (1967) to G.C. Wolpin.]

if-part. Immediate, since $\chi_\emptyset = \lambda x.1$ and $\chi_\mathbb{N} = \lambda x.0$.

only if-part. By contradiction, suppose that $A = \{x : \phi_x \in \mathcal{C}\}$ is nontrivial, yet $A \in \mathcal{R}_*$. So, let $a \in A$ and $b \notin A$. Define f by

$$f(x) = \begin{cases} b & \text{if } x \in A \\ a & \text{if } x \notin A \end{cases}$$

Clearly,
$$x \in A \text{ iff } f(x) \notin A, \text{ for all } x \tag{1}$$

By the recursion theorem (e.g., 2.9.0.27), there is an e such that $\phi_{f(e)} = \phi_e$.

Thus, $e \in A$ iff $f(e) \in A$, contradicting (1). □

2.9.1.2 Corollary. *If $A = \{x : \phi_x \in \mathcal{C}\}$, then $A \not\leq_m \overline{A}$ and thus $A \not\equiv_m \overline{A}$.*

Proof. Suppose
$$x \in A \text{ iff } f(x) \in \overline{A} \tag{$*$}$$

where f is recursive. If e is as above, then $e \in A$ iff $\phi_e \in \mathcal{C}$ iff $\phi_{f(e)} \in \mathcal{C}$ iff $f(e) \in A$, contradicting $(*)$. □

2.9.1.3 Example. K is not a complete index set, that is, *there is no $\mathcal{C} \subseteq \mathcal{P}$ such that*
$$K = \{x : \phi_x \in \mathcal{C}\} \tag{1}$$

Suppose such a $\mathcal{C}$ exists. Define

$$\psi(x,y) \simeq \begin{cases} 0 & \text{if } x = y \\ \uparrow & \text{otherwise} \end{cases}$$

By definition by positive cases, ψ is computable, so let by 2.9.0.26, e be such that $\psi(e, y) \simeq \phi_e(y)$, for all y. It follows that

$$\phi_e(e) \downarrow \tag{2}$$

but

$$\phi_e(y) \uparrow, \text{ if } y \neq e \tag{3}$$

By (2), $e \in K$ and, by (1),

$$\phi_e \in \mathcal{C} \tag{4}$$

Let now $m \neq e$ be another among the infinitely many indices of ϕ_e, that is, $\phi_e = \phi_m$.

By (4), $\phi_m \in \mathcal{C}$. By (1) and the original definition of K, this entails $\phi_m(m) \downarrow$ which also says $\phi_e(m) \downarrow$, contradicting (3). We cannot have (1)!   □

2.9.1.4 Example. A machine that ignores the input and just outputs itself! Define ψ as $\psi = \lambda xy.x$. By 2.9.0.26 there is an e such that $\phi_e(y) = \psi(e, y) = e$, for all y.   □

The next application is about self-referential (recursive) definitions of functions F that may look arbitrarily complicated as the one below

$$F(\vec{x}_n) \simeq f\Big(\dots F\big(\dots F(\dots)\dots\big)\dots F\big(\dots F(\dots F(\dots)\dots)\dots\big)\dots\Big) \tag{1}$$

where *nesting* of occurrences of F is allowed (unlike the case of primitive recursion).

For example, the Ackermann function $\lambda nx.A_n(x)$ involves some nesting of A inside A in its definition (2.4.1.2). This function's definition is an instance of (1) as we can immediately see if we rephrase it a bit: $A_n(x)$ is given by

$$A_n(x) = \begin{cases} x + 2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ A_{n \dot{-} 1}\big(A_n(x \dot{-} 1)\big) & \text{otherwise} \end{cases}$$

We are interested in just those cases that the right hand side of (1), the "$f(\dots)$" consists only of "computable operations", meaning that the right hand side can be built by a $\mathcal{P}$-derivation *modified* to utilize F as an *initial function*. Another way to describe the shape of the right hand side of (1) is in terms of closures (cf. 1.6.0.12).

2.9.1.5 Definition. We say that a function is *partial recursive in F* iff it is in the closure of $\mathcal{I} \cup \{F\}$ under composition, primitive recursion, and (μy), where $\mathcal{I}$ is the already adopted set of initial functions of $\mathcal{P}$.   □

2.9.1.6 Remark. It follows from 2.9.1.5 that if $F \in \mathcal{P}$, then a function that is partial recursive in F is just partial recursive—see Exercise 2.12.70.

In particular, if we replace F *throughout the right hand side* of (1) by a partial recursive function $\phi_e^{(n)}$ of the same arity as F, then we end up with a partial recursive function. $\square$

In (1) F acts as a *function variable* to "solve" for. A *solution* h for F is a *specific function* that makes (1) true for all $\vec{x}_n$ if we replace all occurrences of F by h.

We next show that if the right hand side of (1) is partial recursive in F, then (1) *always has a partial recursive solution for F*. That is,

$$(\exists e)\Big(\text{if } F \text{ in (1) is replaced everywhere by } \phi_e^{(n)}, \text{ (1) becomes true for all } \vec{x}_n\Big) \quad (2)$$

Indeed, the function $\lambda z\vec{x}_n.G(z, \vec{x}_n)$ given below by substituting all F in the right hand side of (1) by $\lambda z\vec{x}_n.\phi_z^{(n)}(\vec{x}_n)$ is partial recursive by 2.9.1.6.

$$G(z, \vec{x}_n) \tag{3}$$
$$\simeq f\Big(\ldots\phi_z^{(n)}\Big(\ldots\phi_z^{(n)}(\ldots)\ldots\Big)\ldots\phi_z^{(n)}\Big(\ldots\phi_z^{(n)}\big(\ldots\phi_z^{(n)}(\ldots)\ldots\big)\ldots\Big)\ldots\Big)$$

By the recursion theorem there is an e such that

$$G(e, \vec{x}_n) \simeq \phi_e^{(n)}(\vec{x}_n), \text{ for all } \vec{x}_n$$

Thus, (3) yields

$$\phi_e^{(n)}(\vec{x}_n)$$
$$\simeq f\Big(\ldots\phi_e^{(n)}\Big(\ldots\phi_e^{(n)}(\ldots)\ldots\Big)\ldots\phi_e^{(n)}\Big(\ldots\phi_e^{(n)}\big(\ldots\phi_e^{(n)}(\ldots)\ldots\big)\ldots\Big)\ldots\Big)$$

That is, setting the function variable F equal to $\phi_e^{(n)}$ we have solved (1), and with a $\mathcal{P}$-solution at that!

The above technique says nothing about uniqueness of solution for F, or totalness. Such issues must be explored separately by methods other than the recursion theorem.

2.9.1.7 Example. Here is a second solution to the question "is $\lambda nx.A_n(x) \in \mathcal{R}$?". $A_n(x)$ is given by

$$A_n(x) = \begin{cases} x+2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ A_{n\dot-1}\big(A_n(x \dot- 1)\big) & \text{otherwise} \end{cases}$$

We rewrite the above using F as a function variable and setting $F(n, x) = A_n(x)$.
Thus, F is "given" by

$$F(n, x) = \begin{cases} x+2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ F\big(n \dot- 1, F(n, x \dot- 1)\big) & \text{otherwise} \end{cases} \tag{4}$$

(4) has the form (1) of the preceding general discussion, and all assumptions are met. Thus, for some e, $F = \phi_e^{(2)}$ works. But is this $\phi_e^{(2)}$ total? Is it the same as $A_n(x)$? Yes, provided (4) has a *unique* total solution! That (4) indeed does have a unique total solution is an easy (double) induction exercise that shows $F(n, x) = F'(n, x)$ for all n, x if

$$F'(n, x) = \begin{cases} x + 2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ F'\big(n \,\dot{-}\, 1, F'(n, x \,\dot{-}\, 1)\big) & \text{otherwise} \end{cases}$$

The existence (of a solution), is of course, provided by the recursion theorem. See Exercise 2.12.71. □

2.9.1.8 Example. Here is a recursion that goes "backwards", that is, defines a function at input n in terms of its value at input $n + 1$. Suppose that g is recursive and define

$$h(n, \vec{x}_k) \simeq \begin{cases} n & \text{if } g(n, \vec{x}_k) = 0 \\ h(n + 1, \vec{x}_k) & \text{otherwise} \end{cases} \tag{1}$$

Replacing the h in the right hand side by $\phi_z^{(k+1)}$, we have a partial recursive G:

$$G(z, n, \vec{x}_k) \simeq \begin{cases} n & \text{if } g(n, \vec{x}_k) = 0 \\ \phi_z^{(k+1)}(n + 1, \vec{x}_k) & \text{otherwise} \end{cases} \tag{2}$$

By the recursion theorem, there is an e such that $\phi_e^{(k+1)}$ solves (1):

$$\phi_e^{(k+1)}(n, \vec{x}_k) \simeq \begin{cases} n & \text{if } g(n, \vec{x}_k) = 0 \\ \phi_e^{(k+1)}(n + 1, \vec{x}_k) & \text{otherwise} \end{cases} \tag{3}$$

What is the import of this example? That we rediscover the computability of functions defined by $\widetilde{\mu}$. Indeed, we see that $\phi_e^{(k+1)}(0, \vec{x}_k) \simeq (\widetilde{\mu}y)g(y, \vec{x}_k)$, for all $\vec{x}_k$ (cf. 2.7.1.2 and Exercise 2.12.72). □

2.10 COMPLETENESS

The concept of reducibility has been instrumental toward certifying in the preceding papes that several problems were unsolvable or non c.e. culminating to the proof of Rice's lemmata and theorem. At the heart of the use of the technique was the observation that when $A \leq_m B$ or $A \leq_1 B$, then B is "more unsolvable" than A. Does this ordering, $\leq_m$ (resp. $\leq_1$), have a "maximal" element among c.e. sets? Indeed, it does have several. Such sets are called m-complete (resp. 1-complete).

2.10.0.9 Definition. (m- **and** 1-**completeness**) A set A is called m-complete (resp. 1-complete) iff the two conditions below hold

(1) A is c.e.

(2) If S is any c.e. set, then $S \leq_m A$ (resp. $S \leq_1 A$). $\square$

2.10.0.10 Example. $K_1 = \{[x, y] : \phi_x(y) \downarrow\}$ is 1-complete.
Indeed, first K_1 is semi-recursive since

$$z \in K_1 \equiv (\exists y)(\exists y)\big(z = [x, y] \wedge \phi_x(y) \downarrow\big)$$

Second, let S be c.e., that is, $S = W_e$ for some e. Then $x \in S \equiv [e, x] \in K_1$. Thus, $S \leq_1 K_1$, since $\lambda x . [e, x]$ is 1-1. $\square$

2.10.0.11 Example. K 1-complete. Indeed, first we know that K is semi-recursive. Second, let S be semi-recursive. Define

$$\psi(x, y) \simeq \begin{cases} 0 & \text{if } x \in S \\ \uparrow & \text{otherwise} \end{cases}$$

ψ is defined by positive cases, so it is computable. By S-m-n, there is an 1-1 $h \in \mathcal{PR}$ such that, for all x and y, we have $\psi(x, y) \simeq \phi_{h(x)}(y)$. Hence

$$\phi_{h(x)}(y) \simeq \begin{cases} 0 & \text{if } x \in S \\ \uparrow & \text{otherwise} \end{cases}$$

and therefore

$$\phi_{h(x)}(h(x)) \downarrow \equiv x \in S \qquad (1)$$

(1) says "$S \leq_1 K$". $\square$

2.10.0.12 Proposition. *A c.e. set A is m-complete iff $S \leq_m A$ for some m-complete S.*

Proof. For the *if*, let B be c.e. Then $B \leq_m S$. Hence (Exercise 2.12.64), $B \leq_m A$. For the *only if*, the assumption on A and the semi-recursiveness of S will do. $\square$

2.10.0.13 Corollary. *If A is m-complete, then it is creative.*

Proof. By assumption, $K \leq_m A$. By 2.8.0.21, $\overline{A}$ is productive. But A is c.e. $\square$

It is clear that all 1-complete sets have the "same difficulty" and the same is true of all m-complete sets, for if A and B are 1-complete, then we have both $A \leq_1 B$ and $B \leq_1 A$—applying the second condition from 2.10.0.9 first for B and then for A. This "equal difficulty (or complexity)" concept has a symbol: We write $A \equiv_1 B$ for $A \leq_1 B \wedge B \leq_1 A$ and $A \equiv_m B$ for $A \leq_m B \wedge B \leq_m A$.

2.10.0.14 Example. Each of $\equiv_1$ and $\equiv_m$ are equivalence relations. Cf. Exercise 2.12.65. $\square$

2.10.0.15 Definition. The *equivalence classes* (cf. 1.2.0.29) of $\equiv_1$ and $\equiv_m$ are called 1-degrees and m-degrees, respectively. If $A \equiv_1 B$ we then say that "A and B have, or belong to, the same 1-degree". If $A \equiv_m B$ we then say that "A and B have, or belong to, the same m-degree". □

2.10.0.16 Example. Thus, K and K_0 belong to the same 1-*degree* and to the same m-*degree*. By Theorem 1.2.0.30, $K \in [K_0]_{\equiv_1}$ and also $K \in [K_0]_{\equiv_m}$. □

2.10.0.17 Example. Since trivially $A \leq_1 B$ implies $A \leq_m B$ we also have that $A \equiv_1 B$ implies $A \equiv_m B$. Therefore, for any A, we have $[A]_{\equiv_1} \subseteq [A]_{\equiv_m}$ (cf. 1.2.0.30). □

Along with a concept of "having the same complexity" one needs a concept of "having strictly less (more) complexity".

2.10.0.18 Definition. We write $A <_m B$ (resp. $A <_1 B$) for $A \leq_m B \wedge \neg B \leq_m A$ (resp. $A \leq_1 B \wedge \neg B \leq_1 A$). □

2.10.0.19 Example. $A <_m B$ is equivalent also to $A \leq_m B \wedge \neg A \equiv_m B$. A similar observation applies to $<_1$. □

A usual shorthand for denoting the negation of a relation R is to write $\not\!R$. Thus we may write $B \not\leq_m A$ for $\neg B \leq_m A$ and $B \not\equiv_m A$ for $\neg B \equiv_m A$.

2.10.0.20 Example. Is there any set A such that $A <_1 K$? Well, yes! Every recursive set A satisfies this inequality, since $K \leq_1 A$ cannot be; it would render A non recursive. □

How about c.e. sets? Are all non recursive c.e. sets 1-complete? Post has answered this negatively.

2.10.0.21 Definition. (Simple sets) A set S is called *simple* if it fits the three following conditions:

(1) S is c.e.

(2) $\overline{S}$ is infinite.

(3) S intersects[99] every infinite W_x. □

Thus a simple set is c.e. but not recursive (cf. Exercise 2.12.66). Post proved, constructively, that simple sets exist.

2.10.0.22 Theorem. (Post (1944)) *Simple sets exist.*

---

[99]That is, for every such W_x, we have $S \cap W_x \neq \emptyset$.

Proof. We construct one! In building (enumerating) a simple set S by stages, we will achieve requirement (1) of Definition 2.10.0.21 by enumerating computably the set that we are constructing. We achieve requirement (2) by making S have large "gaps" in its enumeration, thus ensuring a "large" complement.

Requirement (3) is met by systematically putting into S at least one member from *each* W_x—and therefore from each *infinite* W_x.[100] We meet (3) by modifying the *selection function* (2.7.3.2): For each x, we select a member m of W_x and place it in the under construction S as long as $m > 2x$. This guarantees that our S intersects every infinite W_x, as it *will* have a member such as m.

This process being computable, makes S c.e. On the other hand, in every interval of integers, $0, \ldots, 2m$ a k will be in S only if $k > 2i$ for some $i \leq m$. The largest "i" that would contribute a "k" is $m - 1$. Thus there are at most m members of S in the interval, and hence at least $m + 1$ members in $\overline{S}$. m being arbitrary, $\overline{S}$ is infinite.

Mathematically, the enumerator f is given, for all x, by

$$f(x) \simeq \Big((\mu z)\big(T(x, (z)_0, (z)_1) \wedge (z)_0 > 2x\big) \Big)_0$$

Clearly, $f \in \mathcal{P}$, and thus $\mathrm{ran}(f)$ is c.e.   □

2.10.0.23 Corollary. *There is a c.e. set that is not m- nor 1-complete.*

Proof. The simple set S fits the bill, for were it m-complete it would then be also creative by 2.10.0.13. But a creative set has a productive complement which contains an infinite W_x.   □

From the above: A simple set cannot be creative.

2.10.0.24 Example. Myhill proved that *creative sets are 1-complete*. This entails that the concepts of 1-complete and m-complete coincide, since, by 2.10.0.13, if A is m-complete then it is creative, and therefore 1-complete. But, trivially, 1-completeness implies m-completeness.

We show here a less involved and weaker result that *creative sets are m-complete*.

So, let A be creative and let f be a productive function for $\overline{A}$. Now A is c.e. so we only need to show that $K \leq_m A$ (cf. 2.10.0.12).

Define, for all x, y, z,

$$\psi(x, y, z) \simeq \begin{cases} 0 & \text{if } z = f(x) \wedge y \in K \\ \uparrow & \text{otherwise} \end{cases}$$

ψ is defined by positive cases hence is computable. By S-m-n we have a 1-1 primitive recursive g of two variables such that $\psi(x, y, z) \simeq \phi_{g(x,y)}(z)$, for all x, y, z. Thus, by 2.9.0.30, there is a 1-1 (primitive recursive) function h such that

$$\phi_{g(h(y),y)} = \phi_{h(y)} \tag{1}$$

---

[100]We cannot algorithmically focus on infinite W_x only, since $\{x : W_x \text{ is infinite}\}$ is not c.e. (cf. Exercise 2.12.51).

for all y. Hence,

$$\phi_{h(y)}(z) \simeq \phi_{g(h(y),y)}(z) \simeq \psi(h(y), y, z) \simeq \begin{cases} 0 & \text{if } z = f(h(y)) \wedge y \in K \\ \uparrow & \text{otherwise} \end{cases} \qquad (2)$$

Taking domains, we have $W_{g(h(y),x)} = W_{h(y)}$ by (1) and

$$W_{h(y)} = \begin{cases} \{f(h(y))\} & \text{if } y \in K \\ \emptyset & \text{otherwise} \end{cases} \qquad (3)$$

by (2). We claim that $K \leq_m A$ via (fh). To see this, let first $y \in K$. Thus, by (3),

$$W_{h(y)} = \{f(h(y))\} \qquad (4)$$

We argue that $f(h(y)) \in A$. For if not, then $f(h(y)) \in \overline{A}$, that is, $W_{h(x)} \subseteq \overline{A}$, by (4). Since f is a productive function for $\overline{A}$ we also obtain $f(h(x)) \in \overline{A} - W_{h(x)}$ which is untenable by (4). This settles the claim.

Conversely, let $y \notin K$. We want $f(h(y)) \notin A$, that is, $f(h(y)) \in \overline{A}$. This is so, since, by (3), in this case we have $W_{h(y)} = \emptyset \subseteq \overline{A}$, thus, by productiveness, $f(h(y)) \in \overline{A} - W_{h(y)} = \overline{A}$.

All told, $y \in K$ iff $f(h(y)) \in A$.

Pause. There is a short distance from this result to the full result of Myhill's that proves A to be 1-complete. The latter readily would follow if f, the productive function, were 1-1—in which case so would be (fh). It turns out that we can prove that we can always choose a 1-1 productive function, a proof we will not get into here. See Rogers (1967) or Tourlakis (1984).◀   □

2.11   UNPROVABILITY FROM UNSOLVABILITY

This section draws from background developed in Section 1.1 and in particular in Subsection 1.1.1. Nevertheless, we will need to indulge in some repetition here, aiming to make this section self-sufficient on one hand, and, on the other, making the underlying logic that we use and discuss here more *formal* than we managed to get away with so far,[101] since in the present section logic will not be just a tool, but primarily will be an object for study—precision is called for!

We will prove here a *semantic version* of Gödel's first incompleteness theorem that relies on computability techniques. In this form the theorem states that any "reasonable" axiomatic system that attempts to have as theorems *precisely* all the "true" (first-order) formulae of formal arithmetic[102] will fail to be *complete* in this sense: There will be infinitely many *true* formulae that are *not* theorems. Imitating

---

[101]*In this volume we apply logic informally,* we announced on p. 2.
[102]What makes arithmetic *formal* is its foundation on axioms and precise rules of logic, where the *form* of formulae, rules of inference and proofs matters.

Cantor's separation of infinities of sets, between "small" (enumerable) and "large" (non-enumerable) we will show that the set of true formulae of arithmetic is "computably large" (non c.e., indeed, *productive*) while the set of provable formulae is "computably small" (c.e.). *Thus the two cannot coincide.*

The qualifier *reasonable* could well be replaced by *practical*: One must be able to tell, algorithmically, whether or not a formula is an axiom—how else can one check a proof, let alone write one? "True" means true in the *standard interpretation* of the abstract symbols of the formal arithmetic, terminology that we will carefully introduce in what follows.

Now, in order to "do" axiomatic arithmetic we need, first of all, a first-order (logical) language, which we use to write down formulae and proofs. Before we get to the language—that is, the set of "important" strings, namely, terms and formulae, of our axiomatic arithmetic—we need an *alphabet*. This alphabet has two parts: One that is standard in all logical languages (*logical symbols*) that we employ to do mathematics, namely

(1) Two symbols, v and $\#$, used to *generate* all the (infinitely many) *formal names* of variables of natural number *type*. These names are

$$\underbrace{v \# \ldots \# v}_{n \; \#\text{'s}}, \text{ for all } n \geq 1 \qquad (A)$$

(2) The logical part also contains the following symbols (see also Subsection 1.1.1)

$$\neg, \vee, =, \forall, (,)$$

The other part of the alphabet is *specific to doing arithmetic*. It contains the *special symbols*

$$0, S, +, \times, @, <$$

These are the *nonlogical* symbols for arithmetic, which we—in principle—can interpret any way we please, *but* we will interpret them in the *standard* (expected) way, namely,

Abstract (language) symbol	Concrete interpretation
0	0 (zero)
S	$\lambda x.x + 1$
$+$	$\lambda xy.x + y$
$\times$	$\lambda xy.x \times y$
@	$\lambda xy.x^y$
$<$	$\lambda xy.x < y$

We next turn to the definition of the *objects*, technically known as *terms*, which the language can express—beyond the obvious objects that are the variables and the constant 0.

2.11.0.25 Definition. A term is a variable or the symbol 0, or—assuming that t and s designate terms—any of $St, (t + s), (t \times s), (t@s)$. If a term is variable-free it is called *closed*. $\qquad \square$

One forms formulae now in the standard way (see also Subsection 1.1.1) starting with the *atomic* formulae, and then building more complicated ones using the connectives and the quantifier.

2.11.0.26 Definition. An atomic formula has one of the forms $t = s$ or $t < s$.   □

2.11.0.27 Definition. A formula, or well-formed-formula (wff) is one of:

(a) atomic

or, assuming that $\mathscr{A}$ and $\mathscr{B}$ stand for formulae, and x stands for any variable,

(b) $(\neg\mathscr{A})$

(c) $(\mathscr{A} \vee \mathscr{B})$

(d) $((\forall x)\mathscr{A})$

A formula with no free variables is *closed*; it is also called a *sentence*.   □

Unlike the case of Subsection 1.1.1, we adopt here only three logical operators since, as we know, all the others, namely, $\wedge, \rightarrow, \equiv, \exists$, can be introduced by definitions, such as "$\mathscr{A} \rightarrow \mathscr{B}$ means $\neg\mathscr{A} \vee \mathscr{B}$", etc. The priorities of p. 8 hold for all these derived operators as well.

In all that follows, t, s will stand for any terms; $\mathscr{A}, \mathscr{B}, \mathscr{C}, \mathscr{D}$ will stand for any formulae; and x, y, z, in this section and in Subsection 2.11.1, will *stand for any* formal variable—such as $v\#\#v$, etc.

They are all *metavariables (syntactic variables)*.

Each of these may be embellished by subscripts and/or primes, thus we can effectively generate infinitely many metavariables of each sort.

As outlined in Subsection 1.1.1, one omits redundant brackets by adopting priorities for the various connectives and operations. The priority sequence from highest (least scope) to lowest is taken to be

$$S, @, \times, +$$

for operations. For the logical operations $\forall, \neg$ and $\vee$ priorities are as in Subsection 1.1.1, p. 8.

Pivotal in this (and the next sub-) section is the expression "$\phi_x(x) \uparrow$" (cf. 2.3.0.9) of computability. We will want to know that it is *definable* by a formula in the language of arithmetic so that its central relevance to the incompleteness problem *of arithmetic* can be established. *What "definable" means hinges on what we mean when we say that "a closed formula of arithmetic is true".*

Thus we immediately visit the semantics of the language of arithmetic. The traditional way to do so is to define the so-called Tarski semantics of any first-order language, and then apply it to the special case of the language of arithmetic—as was done in Tourlakis (2003a, 2008). Instead we will take a shortcut here, following Smullyan (1992).

2.11.0.28 Definition. (Interpretation of Closed Terms in $\mathbb{N}$**)** The *meaning* or *interpretation* of any closed term t, in symbols $t^{\mathbb{N}}$, is some member k of $\mathbb{N}$, defined as follows (cf. the table on p. 222). We will write $t^{\mathbb{N}} = k$.

(1) $0^{\mathbb{N}} = 0$

(2) $(St)^{\mathbb{N}} = t^{\mathbb{N}} + 1$

(3) $(t + s)^{\mathbb{N}} = t^{\mathbb{N}} + s^{\mathbb{N}}$

(4) $(t \times s)^{\mathbb{N}} = t^{\mathbb{N}} \cdot s^{\mathbb{N}}$

(5) $(t@s)^{\mathbb{N}} = (t^{\mathbb{N}})^{s^{\mathbb{N}}}$

The symbol $=$ above, and the symbols $+$ and $\cdot$ on its right, are the informal equals, plus and times over $\mathbb{N}$, respectively. As usual, we will write ab rather than $a \cdot b$ when a and b are natural numbers. □

2.11.0.29 Definition. (Numerals) For any natural number n the (meta)symbol $\widetilde{n}$ is a short name for the closed term

$$\underbrace{SS \cdots S}_{n \text{ times}} 0 \qquad (N)$$

and it is called a *numeral* (for n). □

Clearly, $\widetilde{0}$ denotes the formal 0, since, in this case, we have an empty S-prefix in (N) above. Intuitively, $\widetilde{n}$ is a formal representation of n in the language of arithmetic. Conversely,

2.11.0.30 Example. (Interpretation of Numerals) For every n, we have $\widetilde{n}^{\mathbb{N}} = n$. We can see this by induction on n: For $n = 0$, we have $\widetilde{0}^{\mathbb{N}} = 0^{\mathbb{N}} = 0$—the first equality by the previous remark, the second by (1) in 2.11.0.28. We next fix n and take the I.H. that $\widetilde{n}^{\mathbb{N}} = n$.

For the I.S. we compute as follows:

$$\widetilde{n+1}^{\mathbb{N}} =^{2.11.0.29} (S\widetilde{n})^{\mathbb{N}}$$
$$=^{2.11.0.28} \widetilde{n}^{\mathbb{N}} + 1$$
$$=^{\text{I.H.}} n + 1 \qquad \square$$

Lastly, we define when a closed formula in the language of arithmetic is true. This is done by induction on the complexity of a formula, that is, the number of occurrences (counting repetitions) of the symbols $\neg, \vee, \forall$ in the formula.

Recall from Subsection 1.1.1, p. 7, that $\mathscr{A}(\mathbf{x})$ means that $\mathbf{x}$ is the only free variable of formula $\mathscr{A}$.

2.11.0.31 Definition. (Truth of Formulae) For closed t and s, $t < s$ and $t = s$ are true precisely when $t^{\mathbb{N}} < s^{\mathbb{N}}$ and $t^{\mathbb{N}} = s^{\mathbb{N}}$ are true over $\mathbb{N}$.

For closed formulae $\mathscr{A}$ and $\mathscr{B}$, $\neg\mathscr{A}$ is true iff $\mathscr{A}$ is not; $\mathscr{A} \vee \mathscr{B}$ is true iff at least one of $\mathscr{A}$ or $\mathscr{B}$ are. $(\forall \mathbf{x})\mathscr{A}(\mathbf{x})$ is true iff, for all $n \in \mathbb{N}$, $\mathscr{A}(\tilde{n})$ is true. □

We embark now on stating and proving Gödel's First Incompleteness Theorem. As this speaks of *any recursive axiomatization* of arithmetic, that is, *any* first-order theory that attempts to prove all the true sentences of arithmetic, starting from a recognizable set of special axioms, we will outline what such an axiomatic system would look like, what its components, (1)–(4) must be.

(1) A *recursive set* of strings: The well-formed formulae (wff).

(2) A *recursive subset* of wff that characterize the "behavior" of (the symbols of) arithmetic: These are the *special* (or *nonlogical*) axioms for arithmetic. We will leave them unspecified, beyond the requirement of them forming a recursive set, so that the requirement that the incompleteness theorem "speaks of *any recursive axiomatization of arithmetic*" is met.[103]

(3) The logical axioms (see Subsection 1.1.1)

(4) The *modus ponens* rule of inference (see Subsection 1.1.1)

2.11.0.32 Remark.

(I) *Intuitively*, (1) above says that there is an algorithm that, for every string over the following alphabet

$$\neg, \vee, =, \forall, (,), v, \#, 0, S, +, \times, <, @, ; \qquad (B)$$

will decide the string's membership in wff—i.e., whether or not the string parses correctly as a formula. The reader with some programming under his belt, who has not skipped over Subection 1.1.1, will not have any trouble believing that he can actually write a program that does just that. We will not expand at such level of detail here.[104]

(II) The symbol ";" in (B) will be justified shortly.

(III) But what do we mean, *mathematically*, when we say that a set of *strings* is recursive? After all, until now all our recursive sets were sets of numbers (or of

---

[103] A particularly famous choice of axioms is due to Peano—the so-called *Peano Arithmetic*, for short "PA". It has axioms that give the behavior of every nonlogical symbol, plus the induction *axiom schema*:

$$\mathscr{P}(0) \wedge (\forall \mathbf{x})(\mathscr{P}(\mathbf{x}) \to \mathscr{P}(S\mathbf{x})) \to (\forall \mathbf{x})\mathscr{P}(\mathbf{x})$$

This "schema" (or form) gives one axiom for each choice of wff $\mathscr{P}$.

[104] A rigorous mathematical proof of the recursiveness of the set of all formulae over the alphabet (B) is not difficult, but is tedious. The reader who would like to see how this is done may refer to Tourlakis (2008).

tuples of numbers). Well, nothing has changed! Imagine that the symbols in our alphabet (B) above, *in precisely the order given*, are just (strange) symbols for the numbers 1 through 15. Then *any* string over the alphabet *denotes a number* base 16[105] (if you are comfortable with hexadecimal numbers like BBC—i.e., 3004 in decimal—then you have to also be with what I have just said). For example, $(\forall v \# v) v \# v = 0$ denotes (in decimal notation) 5804772525881.

Thus, to speak of a set of strings over (B) is the same as talking about a subset of $\mathbb{N}$.

Notation. For any string T over the alphabet (B), $\ulcorner T \urcorner = x$ means that x is the decimal value of the string T. Thus we can recast the concluding example above in the notation $\ulcorner (\forall v \# v) v \# v = 0 \urcorner = 5804772525881$.

(IV) We have said right at the beginning that the set of axioms must be reasonable, i.e., recognizable. That is precisely what item (2) above asks for, when *stipulating* recursiveness. Since there are in principle infinitely many different ways to choose axioms for arithmetic, the correct approach is to "stipulate" recursiveness—as in "let us assume that the axioms are chosen to be recursive"—rather than *expect* it to be an inevitable fact.

Indeed, the set of all true formulae of arithmetic is a fine set of axioms for arithmetic, since from that axiom set precisely all true statements of arithmetic are derivable (the tools of Subsection 1.1.1 lead immediately to this obvious fact).

Except for one thing: This set of axioms is *not* recognizable! See 2.11.0.36.

But don't we want to be able to recognize *all* the axioms, including the logical ones? Yes, but this latter, fixed, set of axioms that does not change from one mathematical theory to another, can be *proved* to form a recursive (recognizable) set. Intuitively, a formula can be recognized to be a logical axiom just because of its form (cf. 1.1.1). Again the details are elementary but very tedious and we will omit them.

We know that the union of two recursive sets is recursive, thus the full set of axioms is recursive once we designed the special axiom set to satisfy (2).

(V) It is also worth observing that the single rule of inference, modus ponens, is recognizable by its form. That is, there is a recursive relation of three arguments, $MP(x, y, z)$, which is true precisely when, for some formulae $\mathscr{X}$ and $\mathscr{Y}$, we have either $x = \ulcorner \mathscr{X} \urcorner$, $y = \ulcorner \mathscr{X} \to \mathscr{Y} \urcorner$, and $z = \ulcorner \mathscr{Y} \urcorner$; or $x = \ulcorner \mathscr{X} \to \mathscr{Y} \urcorner$, $y = \ulcorner \mathscr{X} \urcorner$, and $z = \ulcorner \mathscr{Y} \urcorner$, where (see Section 1.1.1) $\mathscr{X} \to \mathscr{Y}$ abbreviates $\neg \mathscr{X} \vee \mathscr{Y}$.   □

---

[105]Why not number the symbols in (B) by 0 through 14 and work base 15? Because we will have trouble with strings such as $\neg 0 < 0$. The "digit" in the most significant position is (of value) 0 and we lose information as we pass to a numerical value. I.e., both $\neg 0 < 0$ and $0 < 0$ denote the same number.

2.11.0.33 Lemma. **(Informal)** *The set of all theorems of an axiomatic arithmetic as this has been described in* (1)–(4) *above is c.e.*

Proof. **(Informal)** We will use the symbol ";" of the alphabet (B) as "glue" to concatenate all the formulae in any given proof that we may write. I.e., we explicitly write ";" *once* between every pair of consecutive formulae

$$\mathscr{F}_1, \mathscr{F}_2, \ldots, \mathscr{F}_n$$

of a proof. We thus convert the *sequence* to a *single string* (over (B))

$$\mathscr{F}_1; \mathscr{F}_2; \ldots; \mathscr{F}_n \tag{i}$$

We can think of any expression (string) like (i) above as a base-16 notation of a number.

Now let us have the relation $P(x, y)$ stand for the statement: *For some proof* $\mathscr{F}_1, \mathscr{F}_2, \ldots, \mathscr{F}_n$, *it is the case that* $y = \ulcorner \mathscr{F}_1 \# \mathscr{F}_2 \# \ldots \# \mathscr{F}_n \urcorner$ *and* $x = \ulcorner \mathscr{F}_n \urcorner$.

It can be proved that $P(x, y)$ is recursive: Intuitively, one first needs to express the (given in decimal) numbers x and y in base 16, that is, convert them into strings over (B).

One can now test (algorithmically; being able to tell an axiom from a non axiom and to perform modus ponens algorithmically helps here!) whether or not the string obtained from y has the form (i) *and*, if so, whether it is the "glued form" of a proof whose last formula is the string extracted from x. If the test succeeds, then $P(x, y)$ is true, else it is false.

Let next $\Theta(x)$ stand for "the representation of x in base 16 is a string that is a theorem of arithmetic". Then $\Theta(x) \equiv (\exists y) P(x, y)$ and we are done by the projection theorem. $\square$

2.11.0.34 Corollary. *The set of all closed theorems (theorems that are sentences) of an axiomatic arithmetic as this has been described in* (1)–(4) *above is c.e.*

Proof. Computably build two lists simultaneously: List1 is that of the preceding lemma. For each *sentence* enumerated in List1, copy it in List2. $\square$

Following Smullyan (1992) we will call a recursive axiomatization of arithmetic *correct* iff *all* the nonlogical axioms are true in the standard interpretation given in this subsection. For example, the Peano axiomatization is correct.

Since the only rule of inference is modus ponens and we have $\mathscr{A}, \mathscr{A} \to \mathscr{B} \models_{taut} \mathscr{B}$, it follows that all the theorems of a correct axiomatization are true in the standard interpretation. But are *all* truths theorems?

Let us call **Complete Arithmetic**, for short CA, the set

$$\{\ulcorner S \urcorner : S \text{ is a true } sentence \text{ of arithmetic}\}$$

We now have:

2.11.0.35 Theorem. (Gödel's First Incompleteness Theorem) Every *correct and recursive axiomatic system for arithmetic that satisfies* (1)–(4) *above is* incomplete *in the sense that its set of closed theorems cannot equal the set CA.*

Proof. In view of Corollary 2.11.0.34, it suffices to prove that CA is not c.e. To this end, consider the two sets of sentences below:

$$\widetilde{K} = \{\ulcorner \phi_{\widetilde{a}}(\widetilde{a}) \uparrow \urcorner : \phi_{\widetilde{a}}(\widetilde{a}) \uparrow \text{ is true as interpreted over } \mathbb{N}\}$$

and

$$Q = \{\ulcorner \phi_{\widetilde{a}}(\widetilde{a}) \uparrow \urcorner : a \in \mathbb{N}\} \tag{1}$$

Before we proceed, let me note that "$\phi_{\widetilde{a}}(\widetilde{a}) \uparrow$" is an abbreviation of a formula of arithmetic—in other words, it can be written down, in principle, using no more than the symbols from the alphabet (B), and this formula will say (i.e, will be true iff) "$\phi_a(a) \uparrow$ holds" (see next subsection). Thus,

$$\widetilde{K} = \{\ulcorner \phi_{\widetilde{a}}(\widetilde{a}) \uparrow \urcorner : a \in \overline{K}\} \tag{2}$$

Now, given any $a \in \mathbb{N}$, we can *construct* the formula $\phi_{\widetilde{a}}(\widetilde{a}) \uparrow$, which, using a different abbreviation, is $\neg(\exists y)T(\widetilde{a}, \widetilde{a}, y)$. That is, the function f that on input a outputs $\ulcorner \phi_{\widetilde{a}}(\widetilde{a}) \uparrow \urcorner$ is, intuitively, computable.[106] Thus

$$\overline{K} \leq \widetilde{K} \tag{3}$$

by (2), since (2) says $a \in \overline{K}$ iff $f(a) \in \widetilde{K}$. On the other hand,

$$Q = \operatorname{ran}(f) \tag{4}$$

by (1). Thus, $\widetilde{K}$ is *not* c.e., while Q *is* c.e. But $CA \cap Q = \widetilde{K}$, thus CA cannot be c.e. by closure properties.   □

   Note the emphasized "every" in the theorem. It draws attention to the fact that we have not *fixed* any *particular* "reasonable" theory: whatever we have said holds for *all correct, recursive theories* that axiomatize arithmetic.

The main result in the above proof was that CA is not c.e., a fact derivable *irrespective* of correctness. Thus, that the set of theorems of the axiomatic theory does not equal CA holds even *without* the correctness assumption. On the other hand, it would be unreasonable to expect an "incorrect" axiomatization to have *all* its closed theorems in CA anyway.

2.11.0.36 Corollary. *CA is productive.*

Proof. By (3) above and 2.8.0.17 we have that $\widetilde{K}$ is productive. Then, by 2.8.0.25 and $CA \cap Q = \widetilde{K}$, we have that CA is productive.   □

---

[106]If one decides to verify this carefully, he will find that f is in fact even primitive recursive.

The corollary and 2.8.0.22 establish that not only we *miss* infinitely many true arithmetic sentences in *any* recursive axiomatization of arithmetic, but, moreover, we can *algorithmically list* an infinite subset of these missed true sentences.

Pause. Why "subset"?◄

Indeed, the recursive axiomatization has a c.e. set of *closed* theorems W_x, by 2.11.0.34. By correctness, we have that $W_x \subseteq$ CA. By 2.8.0.22 we can *build* an infinite c.e. set, W_y—*in CA*—that avoids *all* of W_x.

2.11.1 Supplement: $\phi_x(x) \uparrow$ is Expressible in the Language of Arithmetic

The title of this subsection means that there is a formula of arithmetic, let us call it $\mathscr{A}(\mathbf{x})$, such that, for all n, $\phi_n(n) \uparrow$ is true iff $\mathscr{A}(\widetilde{n})$ is true—the latter in the sense of Definition 2.11.0.31.

2.11.1.1 Definition. A relation $R(x)$ over the natural numbers—that is, a relation in the metatheory of formal arithmetic—is *expressed* (also called *defined*) by a formula of arithmetic, $\mathscr{B}(\mathbf{x})$, iff for all $n \in \mathbb{N}$, we have

$$R(n) \text{ is true iff } \mathscr{B}(\widetilde{n}) \text{ is true}$$

Suppressing reference to $\mathscr{B}(\mathbf{x})$ we can also say that $R(x)$ is *expressible* (or *definable*, in the language of arithmetic).

The definition can be extended in the obvious way in the case of relations of many variables. □

Let us next define the set of *arithmetical* relations on the set of natural numbers.[107]

2.11.1.2 Definition. The set of *arithmetical relations* is the smallest set of relations over the set of natural numbers that satisfies:

It contains the "initial" relations (of three variables) $z = x + y$, $z = x \cdot y$, and $z = x^y$, where the exponentiation is the "ex" of Example 2.1.2.14 but we have here reverted to the standard notation.

Moreover,

(1) If $Q(\vec{x})$ and $P(\vec{y})$ are in the set, then so are $\neg Q(\vec{x})$ and $Q(\vec{x}) \vee P(\vec{y})$.

(2) If $R(y, \vec{x})$ is in the set, then so is $(\forall y)R(y, \vec{x})$.

(3) If $Q(\vec{x})$ is in the set, then so are all its *explicit transformations*.

---

[107]The arithmetical relations have a lot of tolerance for variations in their definition: Sometimes as much as all of $\mathcal{R}_*$ is taken as the "initial" arithmetical relations. Sometimes as little as $z = x + y$ and $z = x \cdot y$. For technical convenience we have added the graph of exponentiation rather than choosing the most minimalist approach.

Explicit transformations [Smullyan (1961) and Bennett (1962)] are exactly the following: substitution of any constant into a variable, expansion of the variables-list by "don't care" variables (arguments), permutation of variables, identification of variables—that is, Grzegorczyk operations (ii)–(iv) (cf. 2.1.2.6), albeit applied to relations. □

Clearly the set of arithmetical relations is closed under the remaining Boolean connectives and $(\exists y)$.

2.11.1.3 Lemma. *Every arithmetical relation is expressible in the language of arithmetic, over the alphabet (B) of p. 225.*

Proof. We proceed by induction along the cases of Definition 2.11.1.2. The basis contains three cases, $z = x + y$ and $z = x \cdot y$ and $z = x^y$.

We argue that the metamathematical relation $z = x + y$ is expressed by the *formal* $\mathbf{z} = \mathbf{x} + \mathbf{y}$ of arithmetic.

This requires us (cf. 2.11.1.1) to establish for all m, n, k:

$$m = n + k \text{ holds iff } \widetilde{m} = \widetilde{n} + \widetilde{k} \text{ is true} \qquad (*)$$

Indeed,

$$\widetilde{m} = \widetilde{n} + \widetilde{k} \text{ is true } \text{ iff } (2.11.0.31) \ \widetilde{m}^{\mathbb{N}} = (\widetilde{n} + \widetilde{k})^{\mathbb{N}} \text{ is true}$$
$$\text{iff } (2.11.0.28) \ \widetilde{m}^{\mathbb{N}} = \widetilde{n}^{\mathbb{N}} + \widetilde{k}^{\mathbb{N}} \text{ is true}$$
$$\text{iff } (2.11.0.30) \ m = n + k \text{ is true}$$

The verifications for the relations $z = xy$ and $z = x^y$ are omitted being entirely analogous. We leave it to the reader to verify that if $R(\vec{x})$ and $Q(\vec{y})$ are expressed by the formulae $\mathscr{A}(\vec{\mathbf{x}})$ and $\mathscr{B}(\vec{\mathbf{y}})$ respectively, then $\neg R(\vec{x})$ and $R(\vec{x}) \vee Q(\vec{y})$ are expressed by $\neg \mathscr{A}(\vec{\mathbf{x}})$ and $\mathscr{A}(\vec{\mathbf{x}}) \vee \mathscr{B}(\vec{\mathbf{y}})$, respectively.

Next, we show that $(\forall y) R(y, \vec{x}_r)$ is defined by $(\forall \mathbf{y}) \mathscr{A}(\mathbf{y}, \vec{\mathbf{x}}_r)$, if $R(y, \vec{x}_r)$ is defined by $\mathscr{A}(\mathbf{y}, \vec{\mathbf{x}}_r)$. We are given, for any $c, b_1, \ldots, b_r$ in $\mathbb{N}$, that

$$R(c, b_1, \ldots, b_r) \text{ is true iff } \mathscr{A}(\widetilde{c}, \widetilde{b}_1, \ldots, \widetilde{b}_r) \text{ is true} \qquad (**)$$

Now, we fix $b_1, \ldots, b_r$ in $\mathbb{N}$. $(\forall y) R(y, b_1, \ldots, b_r)$ holds iff for all $c \in \mathbb{N}$ we have that $R(c, b_1, \ldots, b_r)$ holds. By $(**)$ this is equivalent to saying "for all $c \in \mathbb{N}$ we have $\mathscr{A}(\widetilde{c}, \widetilde{b}_1, \ldots, \widetilde{b}_r)$ is true". By 2.11.0.31 the latter says precisely $(\forall \mathbf{x}) \mathscr{A}(\mathbf{x}, \widetilde{b}_1, \ldots, \widetilde{b}_r)$ is true.

We next look into explicit transformations. Let then $Q(y, \vec{x}_r)$ be defined by the formula $\mathscr{A}(\mathbf{y}, \vec{\mathbf{x}}_r)$. Then, for any fixed $i \in \mathbb{N}$, $Q(i, \vec{x}_r)$ is clearly defined by $\mathscr{A}(\widetilde{i}, \vec{\mathbf{x}}_r)$, since for all $a, b_1, \ldots, b_r$ we have $Q(a, b_1, \ldots, b_r)$ is true iff $\mathscr{A}(\widetilde{a}, \widetilde{b}_1, \ldots, \widetilde{b}_r)$ is true; in particular, $Q(i, b_1, \ldots, b_r)$ is true iff $\mathscr{A}(\widetilde{i}, \widetilde{b}_1, \ldots, \widetilde{b}_r)$ is true.

The case of identifying or permuting variables being trivial, we conclude by looking at the case of adding one "don't care" variable (a case that is extensible by a

trivial induction to any fixed number). So let $\mathscr{A}(\vec{\mathbf{x}}_r)$ define $Q(\vec{x}_r)$ and let z be a *new* informal variable.

I will argue that $\mathscr{A}(\vec{\mathbf{x}}_r) \wedge \mathbf{z} = \mathbf{z}$ defines the relation $R = \lambda z \vec{x}_r.Q(\vec{x}_r)$:

We have, on one hand, for all $b_1, \ldots, b_r$:

$$Q(b_1, \ldots, b_r) \text{ is true iff } \mathscr{A}(\widetilde{b}_1, \ldots, \widetilde{b}_r) \text{ is true} \qquad (\ast\ast\ast)$$

On the other hand, for all $c, b_1, \ldots, b_r$, $Q(b_1, \ldots, b_r) \equiv R(c, b_1, \ldots, b_r)$ and, since $\widetilde{c} = \widetilde{c}$ is true (2.11.0.31),

$$\mathscr{A}(\widetilde{b}_1, \ldots, \widetilde{b}_r) \equiv \mathscr{A}(\widetilde{b}_1, \ldots, \widetilde{b}_r) \wedge (\widetilde{c} = \widetilde{c})$$

Along with $(\ast\ast\ast)$ we get

$$R(c, b_1, \ldots, b_r) \text{ is true iff } \mathscr{A}(\widetilde{b}_1, \ldots, \widetilde{b}_r) \wedge (\widetilde{c} = \widetilde{c}) \text{ is true} \qquad \Box$$

Thus, to show that $\phi_x(x) \uparrow$ is expressible in the language of arithmetic it suffices, because of the preceding lemma, to prove that it is arithmetical. In turn, since $\phi_x(x) \uparrow \equiv \neg(\exists y)T(x, x, y)$, it suffices to prove that the Kleene predicate is arithmetical.

It will so follow if we can prove that every function $f \in \mathcal{PR}$ has an arithmetical graph, for then if χ_T is the characteristic function of T, we will have that $\chi_T(x, y, z) = w$—and therefore $\chi_T(x, y, z) = 0$ by explicit transformation—is arithmetical.[108]

Items (7)–(10) below are due to Grzegorczyk (1953).

2.11.1.4 Lemma. *The following relations are arithmetical.*

(1) $x = 0$ *(and hence $x \neq 0$)*

(2) $x \leq y$ *(and hence $x < y$)*

(3) $z = x \dotminus y$

(4) $x \mid y$

(5) $Pr(x)$

(6) $Seq(z)$

(7) $Next(x, y)$ *(meaning $x < y$ are consecutive primes)*

(8) $pow(z, x, y)$ *(meaning $x > 1$ and x^y is the highest power of x dividing z)*

(9) $\Omega(z)$ *(meaning z has the form $p_0 p_1^2 p_2^3 \cdots p_n^{n+1}$ for some n)*

(10) $y = p_n$

---

[108]Gödel proved all this without the need to have exponentiation as a primitive operation in arithmetic. However, adopting this operation makes things considerably easier and, as mentioned earlier (footnote 107 on p. 229), it does not change the set of arithmetical relations.

(11) $z = \exp(x, y)$ *(cf. 2.1.2.40)*

We need not worry about *bounding* our quantifications, for it is not our purpose to show these relations in $\mathcal{PR}_*$. Indeed we know from earlier work that they are in this set. This time we simply want to show that they are arithmetical.

Proof.

(1) $x = 0$ (and hence $x \neq 0$): $x = 0$ is an explicit transform of $x = y + z$; $x \neq 0$ is obtained by negation.

(2) $x \leq y$ (and hence $x < y$): This is equivalent to $(\exists z)(x + z = y)$.

(3) $z = x \dotminus y$: This is equivalent to $z = 0 \wedge x < y \vee x = z + y$.

(4) $x \mid y$: This is equivalent to $(\exists z)y = xz$ (I am using "implied multiplication" throughout: "xy" rather than "$x \times y$" or "$x \cdot y$").

(5) $Pr(x)$: This is equivalent to $x > 1 \wedge (\forall y)(y \mid x \rightarrow y = 1 \vee y = x)$.

(6) $Seq(z)$: This is equivalent to $z > 1 \wedge (\forall x)(\forall y)(Pr(x) \wedge Pr(y) \wedge x < y \wedge y \mid z \rightarrow x \mid z)$.

(7) $Next(x, y)$: This is equivalent to $Pr(x) \wedge Pr(y) \wedge x < y \wedge \neg(\exists z)(Pr(z) \wedge x < z \wedge z < y)$.

(8) $pow(z, x, y)$: This is equivalent to $x > 1 \wedge x^y \mid z \wedge \neg x^{y+1} \mid z$. [109]

(9) $\Omega(z)$: This is equivalent to $Seq(z) \wedge \neg 4 \mid z \wedge (\forall x)(\forall y)\big(Next(x, y) \wedge y \mid z \rightarrow (\exists w)(pow(z, x, w) \wedge pow(z, y, w + 1))\big)$.

(10) $y = p_n$: This is equivalent to $(\exists z)(\Omega(z) \wedge pow(z, y, n + 1))$.

(11) $z = \exp(x, y)$: This is equivalent to $(\exists w)(pow(y, w, z) \wedge w = p_x)$. $\square$

We can now prove the following theorem that concludes the business of this subsection.

2.11.1.5 Theorem. *For every $f \in \mathcal{PR}$, its graph $y = f(\vec{x}_n)$ is arithmetical.*

Proof. We do induction over $\mathcal{PR}$ (cf. 2.1.2.4):

(1) *Basis.*   There are three graphs to work with here: $y = x + 1$, $y = 0$ and $y = x$ (or, fancily, $y = x_i$; or more fancily, $y = U_i^n(\vec{x}_n)$). They all are explicit transforms of $y = x + z$.

---

[109] Again recall that x^y is that of 2.1.2.14, and $x^{y+1} \mid z \equiv (\exists u)(u = y + 1 \wedge x^u \mid z)$. On the other hand, $x^u \mid z \equiv (\exists w)(w = x^u \wedge w \mid z)$.

(2) *Composition.*   Say, the property is true for the graphs of $f, g_1, \ldots, g_n$. This is the I.H. How about $y = f(g_1(\vec{x}_m), g_2(\vec{x}_m), \ldots, g_n(\vec{x}_m))$? Well, this graph is equivalent to

$$(\exists u_1)\cdots(\exists u_n)\big(y = f(\vec{u}_n) \wedge u_1 = g_1(\vec{x}_m) \wedge \cdots \wedge u_n = g_n(\vec{x}_m)\big)$$

and we are done by the I.H.

(3) *Primitive recursion.*   This is the part that benefits from the work we put into 2.11.1.4. Here's why: Assume (I.H.) that the graphs of h and g are arithmetical, and let f be given for all $x, \vec{y}$ by

$$f(0, \vec{y}) = h(\vec{y})$$
$$f(x + 1, \vec{y}) = g(x, \vec{y}, f(x, \vec{y}))$$

Now, to state $z = f(x, \vec{y})$ is equivalent to stating

$$(\exists m_0)(\exists m_1)\cdots(\exists m_x)\Big(m_0 = h(\vec{y}) \wedge z = m_x \wedge$$
$$(\forall w)\big(w < x \rightarrow m_{w+1} = g(w, \vec{y}, m_w)\big)\Big) \tag{i}$$

The trouble with the "relation" (i) above is that it is not a relation at all, because it has a variable-length prefix: $(\exists m_0)(\exists m_1)\cdots(\exists m_x)$. We invoke coding to salvage the argument. Let us use a single number,

$$m = p_0^{m_0} p_1^{m_1} \cdots p_x^{m_x}$$

to represent all the m_i, for $i = 0, \ldots, x$. Clearly,

$$m_i = \exp(i, m), \text{ for } i = 0, \ldots, x$$

We can now rewrite (i) as

$$(\exists m)\Big(\exp(0, m) = h(\vec{y}) \wedge z = \exp(x, m) \wedge$$
$$(\forall w)\big(w < x \rightarrow \exp(w + 1, m) = g(w, \vec{y}, \exp(w, m))\big)\Big) \tag{ii}$$

The above is arithmetical because of the I.H. Some parts of it are more complicated than others. For example, the part

$$\exp(w + 1, m) = g(w, \vec{y}, \exp(w, m))$$

is equivalent to

$$(\exists u)(\exists v)\big(u = \exp(w + 1, m) \wedge v = \exp(w, m) \wedge u = g(w, \vec{y}, v)\big)$$

The above is arithmetical by the I.H. and the preceding lemma. This completes the proof.   □

2.12   ADDITIONAL EXERCISES

1. Prove (without looking up Euclid's proof) that there are infinitely many primes.

   Hint. See Example 2.1.2.40 and Exercise 2.1.2.42.

2. Modify the pairing function of Grzegorczyk (1953) (2.1.4.5) to make it onto.

3. Give a direct proof that the J in 2.1.4.6 is 1-1.

4. Give a direct proof that the J in 2.1.4.8 is 1-1.

5. Show that J given by

$$J(x,y) = \left\lfloor \frac{(x+y)(x+y+1)}{2} \right\rfloor + y$$

   is an onto pairing function in $\mathcal{PR}$. Show that its projections are in $\mathcal{PR}$.

   Hint. View $\mathbb{N}^2$ is the union of all the finite groups of pairs, for $i = 0, 1, 2, \ldots$,

$$G_i = \{\langle x, y \rangle \in \mathbb{N}^2 : x + y = i\}$$

   Now enumerate $\mathbb{N}^2$ by listing pairs $\langle x, y \rangle$, first, by ascending order (with respect to i) of their group-number i, and then, within each group G_i, listing them in ascending order of the y-component. Show that the position $n = 0, 1, 2, \ldots$ of $\langle x, y \rangle$ in this enumeration is precisely $J(x, y)$, which settles onto-ness. Of course, projections K and L exist (why?). Now observe that $J(x, y) \geq x$ and $J(x, y) \geq y$ and use this fact to show that the projections K and L are primitive recursive.

6. Prove that every finite set is primitive recursive.

7. Prove that $\mathbb{N}$ is primitive recursive.

8. Verify the claims made in Remark 2.1.4.10.

9. Prove that $\lambda x.x!$ (factorial) is primitive recursive.

10. Without using the **if-then-else** function or definition by cases, prove that $\lambda xy. \max(x, y)$ and $\lambda xy. \min(x, y)$ are primitive recursive.

11. Prove that if we know that (1) g is primitive recursive; (2) $f(\vec{x}) \leq g(\vec{x})$, for all $\vec{x}$; and (3) $\lambda z\vec{x}.z = f(\vec{y})$ is in $\mathcal{PR}_*$, then f is primitive recursive.

12. Are the conditions (1) and (2) above necessary in order to arrive to the same conclusion from just (3)?

13. What end-values do X, Y, Z, W hold in Example 2.2.0.13 if the variables initially hold, in order, a, b, c, d?

14. Prove by an appropriate induction that the claim regarding the end-values of X and Y in 2.2.0.15 are correct.

15. Write a loop program that computes $\lambda x. \lfloor x/k \rfloor$.

16. Write a loop program that computes $\lambda x. rem(x, k)$.

17. Redo the previous two problems to ensure that you do *not* nest the **Loop-end** instructions.

18. Prove that the function $\lambda x. \|x\|$, where $\|x\|$ here denotes the number of decimal digits of $x \in \mathbb{N}$, is in $\mathcal{PR}$.

19. Define

$$(\overset{\circ}{\mu}y)_{\leq z} f(y, \vec{x}) \overset{\text{Def}}{=} \begin{cases} \min\{y : y \leq z \land f(y, \vec{x}) = 0\} \\ 0, \text{ if the min does not exist} \end{cases}$$

Prove that $\mathcal{PR}$ is closed under $(\overset{\circ}{\mu}y)_{\leq z}$.

20. Prove that if a class of functions $\mathcal{C}$ is closed under $(\overset{\circ}{\mu}y)_{\leq z}$ and *substitution*, then its corresponding class of relations $\mathcal{C}_* = \{f(\vec{x}) = 0 : f \in \mathcal{C}\}$ is closed under $(\exists y)_{\leq z}$.

21. Is the result of the previous exercise still valid if we replace $(\overset{\circ}{\mu}y)_{\leq z}$ by $(\mu y)_{\leq z}$?

22. Refer to Subsection 2.2.2. Prove that neither of the two relations $\lambda xy. f_x(y) = 0$ and $\lambda xy. f_x(y) \neq 0$ is primitive recursive.

23. Once again, refer to Subsection 2.2.2 where we constructed the "universal" two-argument function $\lambda yx. f_y(x)$ that enumerates all one-argument primitive recursive functions. Prove

- For all $\lambda x. h(x) \in \mathcal{PR}$, there is an m such that $h(x) < f_m(x)$, for all x.
- Base on the preceding bullet a new proof of the fact that $\lambda yx. f_y(x) \notin \mathcal{PR}$.

24. Prove that it is impossible to form $\mathcal{PR}$ as the closure under *substitution* of some *finite* set of primitive recursive functions.

25. Prove that for $n \geq 0$, we have $A_n(x) < A_x(2)$ a.e. Use this fact to show

- For all $\lambda x. h(x) \in \mathcal{PR}$, we have $h(x) < A_x(2)$ a.e.
- $\lambda x. A_x(2) \notin \mathcal{PR}$.

26. Suppose that x, y, z are distinct variables. Show that $(\exists y)(\exists x)_{<z} Q \equiv (\exists x)_{<z}(\exists y) Q$.

Hint. $(\exists x)_{<z} Q \equiv (\exists x)(x < z \land Q)$. But unbounded $\exists$ commute.

27. Show that the set K_0 defined as $\{\langle x, y \rangle : \phi_x(y) \downarrow\}$ is semi-computable.

28. Prove the counterpart "definition by cases" theorem of 2.1.2.37 for $\mathcal{R}$ and $\mathcal{P}$. The assumptions are:

(1) For the $\mathcal{R}$ case, all the f_i are in $\mathcal{R}$, while for the $\mathcal{P}$ case they all are in $\mathcal{P}$.

(2) In both cases, the R_i are in $\mathcal{R}_*$.

The result to prove is that the defined f is in $\mathcal{R}$ and $\mathcal{P}$, respectively.

29. In 2.5.0.17 we saw that $\lambda x.\phi_x(x) + 1$ cannot be extended into a recursive (total) function. Prove that the same is true for $\lambda x.\phi_x(x)$.

30. Show that the set K_1 defined as $\{[x, y] : \phi_x(y) \downarrow\}$ is semi-computable.

31. Show that the set K_1 defined above is *not* recursive.

Hint. Caution: Do not confuse $[x, y]$ with $\langle x, y \rangle$. K_1 is $\{z : \phi_{(z)_0}((z)_1) \downarrow\}$—a set of numbers, not a set of pairs.

32. Show 2.5.0.20 directly from 2.5.0.3.

33. Explain precisely why the alternative "proof" of 2.5.0.7 that was suggested in the **Pause** following the corollary will not work.

34. Prove that $\lambda ixy.\Phi_i(x) = y$ is primitive recursive.

35. Prove that neither

$$f(x) = \begin{cases} 0 & \text{if } x \in K \\ 42 & \text{otherwise} \end{cases}$$

nor

$$g(x) = \begin{cases} 0 & \text{if } x \in K \\ x & \text{otherwise} \end{cases}$$

are in $\mathcal{P}$. This justifies our remarks in 2.5.0.30 that the best we can suggest as "output" in the "otherwise" case is $\uparrow$. In general.

Why "in general"?

36. This exercise attempts to contradict 35. So let $x \in K \equiv \psi(x) = 0$ for some $\psi \in \mathcal{P}$ (cf. 2.5.0.6).

But then $f(x) \simeq$ if $\psi(x) = 0$ then 0 else 42 and $g(x) \simeq$ if $\psi(x) = 0$ then 0 else x, which prove—via 2.1.2.6—that f and g are partial recursive. Is there something wrong with this, and if so what precisely?

37. Prove that a non-empty set is c.e. iff it is the range of some recursive function.

Hint. One direction is trivial.

38. Is the "proof" below correct? If not, where *exactly* does it go wrong?

"Let $y = f(\vec{x}_n)$ be r.e. Then $y = f(\vec{x}_n) \equiv \psi(y, \vec{x}_n) = 0$ for some $\psi \in \mathcal{P}$. Thus $g = \lambda \vec{x}_n.(\mu y)\psi(y, \vec{x}_n)$ is in $\mathcal{P}$. But $g = f$, since the unbounded search finds the y that makes $y = f(\vec{x}_n)$ true, if $f(\vec{x}_n) \downarrow$. Thus, $f \in \mathcal{P}$."

39. Prove that if f and g are in $\mathcal{P}$, then $f(x) = g(x)$ is c.e.

40. Prove that if f and g are in $\mathcal{P}$, then $f(x) \simeq g(x)$ is not necessarily c.e.

Hint. Choose carefully specific f and g for a definitive (non c.e.) example.

41. Prove Corollary 2.7.1.10.

42. Prove that the set $\{\langle x, y \rangle : \phi_x = \phi_y\}$ is not semi-recursive. Thus, as expected (given the result 2.5.0.22), the general "equivalence problem" of (URM-) computable functions is also not semi-decidable.

Hint. Fix ϕ_y to a conveniently simple function.

43. Prove, via Rice's lemmata, that the set of ϕ-indices of each computable function f is not c.e.

Hint. You may want to consider the total and nontotal cases separately.

The invocation of Rice's theorem is not permitted in Exercises 44–54.

44. Prove that the problem $\{0, 1, 5\} \subseteq \operatorname{ran}(\phi_x)$ is undecidable.

45. Prove that the set $\{x : W_x = \{0\}\}$ is not c.e.

46. Prove that the set $\{x : W_x = \{0, 1, 2\}\}$ is not c.e.

47. Prove that the set $\{x : W_x = \mathbb{N}\}$ is not c.e.

48. Prove that the set $\{x : W_x \in \mathcal{R}_*\}$ is not c.e.

49. Prove that the set $\{x : W_x = \text{the set of all even numbers}\}$ is not c.e.

50. Prove that the set $\{x : W_x \text{ is finite}\}$ is not c.e.

51. Prove that the set $\{x : W_x \text{ is infinite}\}$ is not c.e.

52. Prove that the set $= \{x : \operatorname{dom}(\phi_x) \text{ has exactly two elements}\}$ is not recursive. Is it c.e.? Why?

53. Explore and prove as needed

- the set $\{x : \operatorname{ran}(\phi_x) \text{ has exactly five distinct elements}\}$ is not recursive. (I.e., "$x \in A$ is unsolvable"). *Is it c.e.? Why?*

- the set $\{x : \phi_x \text{ is the characteristic function of some recursive set}\}$ is not recursive. *Is it c.e.? Why?*

- the set $\{x : \operatorname{ran}(\phi_x) \text{ contains only odd numbers}\}$ is not recursive. *Is it c.e.? Why?*

54. Prove that $\{x : \phi_x \in \mathcal{PR}\}$ is not c.e.

55. Can *every* infinite c.e. set be enumerated in *strictly increasing order* by a primitive recursive function? Why?

Hint. Bring the Ackermann function into the question.

56. Let

$$f = \lambda x. \text{if } f_R(x) = 0 \text{ then } g(x) \text{ else if } f_Q(x) = 0 \text{ then } h(x) \text{ else } \uparrow$$

where R, Q are c.e. (and mutually exclusive), and g, h, f_R, f_Q are partial recursive, where $R(x) \equiv f_R(x) = 0$ and $Q(x) \equiv f_Q(x) = 0$.

Is f partial recursive? *Why?*

Is the f' defined below the same as f? *Why?*

$$f'(x) = \begin{cases} g(x) & \text{if } R(x) \\ h(x) & \text{if } Q(x) \\ \uparrow & \text{otherwise} \end{cases}$$

Suppose you answered "no" to the last question. *Supplementary*: Is f' partial recursive? *Why?*

57. Prove that there is a primitive recursive τ such that for all x, we have $\text{ran}(\phi_x) = \text{ran}(\phi_{\tau(x)})$ and, moreover, if $\text{ran}(\phi_x) \neq \emptyset$, then $\text{ran}(\phi_{\tau(x)})$ is total.

Hint. Combine 2.7.3.5 with 2.7.3.3.

58. There is a primitive recursive function r such that $\mathbb{N} - \{x\} = W_{r(x)}$, for all x.

Hint. Consider $\psi(x, y) \simeq$ if $y \neq x$ then 0 else $\uparrow$.

59. There is a primitive recursive function q such that $\{x\} = W_{q(x)}$, for all x.

60. There is a primitive recursive function k such that $W_x \cup W_y = W_{k(x,y)}$, for all x and y.

61. (Selection Theorem) For each $n \geq 1$ there is a partial recursive function $\lambda i \vec{y}_n . Sel^{(n+1)}(i, \vec{y}_n)$ such that

(a) $(\exists x)\left(\phi_i^{(n+1)}(x, \vec{y}_n) \downarrow \right)$ iff $Sel^{(n+1)}(i, \vec{y}_n) \downarrow$

(b) $(\exists x)\left(\phi_i^{(n+1)}(x, \vec{y}_n) \downarrow \right)$ iff $\phi_i^{(n+1)}\left(Sel^{(n+1)}(i, \vec{y}_n), \vec{y}_n\right) \downarrow$

Hint. Imitate the dovetailing from the proof of 2.7.3.2.

62. Is there a partial recursive function $\lambda x. f(x)$ such that for all i

$$W_i \neq \emptyset \rightarrow f(i) \downarrow \wedge f(i) = \min\{y : y \in W_i\}$$

If you think that "yes", then you must give a proof.

If you think that "no", then you *must* give a definitive counterexample.

63. For your amusement: Write a "self-reproducing" program in your favourite programming language. This program, on every input, will just print itself—*exactly*; i.e., it prints nothing else—and then halts.

64. Prove that $\leq_m$ and $\leq_1$ are transitive relations.

65. Prove the claims made in 2.10.0.14.

66. Prove that a simple set is not recursive.

Hint. You want the set's complement to be c.e. (Why?) Can this be?

67. If f, of one variable, is recursive then there is an e such that $W_e = W_{f(e)}$.

68. If g, of two variables, is recursive then there is an 1-1 primitive recursive $\lambda x.h(x)$ such that $W_{h(x)} = W_{g(h(x),x)}$, for all x.

Hint. From $W_a = \text{dom}(\phi_a)$ and 2.9.0.30.

69. Let f be recursive. Is any of $\{x : \phi_{f(x)}(x) \downarrow\}$ and $\{x : \phi_x(f(x)) \downarrow\}$ a complete index set?

Hint. It depends on f.

70. Let a F be a computable function. Then the closure of $\mathcal{I} \cup \{F\}$ under composition, primitive recursion and unbounded search is a subset of $\mathcal{P}$ (indeed equal to it).

Hint. Do induction on said closure.

71. Prove the uniqueness of solution of the recurrence for F in 2.9.1.7.

72. Settle the unproved claims in Example 2.9.1.8.

73. Let g be recursive. Show that

$$h(n, \vec{x}_k) \simeq \begin{cases} 0 & \text{if } g(n, \vec{x}_k) = 0 \\ h(n+1, \vec{x}_k) + 1 & \text{otherwise} \end{cases}$$

has a computable h-solution $\phi_e^{(n+1)}$ that satisfies $\phi_e^{(n+1)}(0, \vec{x}_k) \simeq (\widetilde{\mu}y)g(y, \vec{x}_k)$, for all $\vec{x}_k$.

74. Answer the question in the **Pause** on p. 229.

75. Complete the proof given for 2.11.1.3.

CHAPTER 3

A SUBSET OF THE URM LANGUAGE; FA AND NFA

This chapter is presented for enhanced completeness of coverage, but is a mostly "how to" chapter, rather than one that poses and answers fundamental questions on the limitations of computing,[110] the latter being our central theme in this volume. Nevertheless, concepts and tools developed here are usable in the theory and practice of compiler writing, a principal area of application.

We will first introduce informally a modified and restricted URM. This new URM model will have explicit "read" instructions.[111]

Secondly, any specific URM under this model will only have one variable that we may call generically "**x**". This variable will always be of *type digit*; it cannot hold arbitrary integers, rather it can only hold digits as values. It has no **stop** instruction, nor instructions for adding/subtracting.

Pause. In the absence of a **stop** instruction, how does a computation halt? We postulate that our modified URMs halt simply by reading something *unexpected*, that

---

[110]However, the pumping lemma, 3.1.3.1, exposes a limitation of the model of computation discussed in this chapter.

[111]In 2.1.1.3 we explained why explicit *read* instructions are theoretically as redundant as explicit *write* instructions are.

is, an object that is *not* a member of the *input alphabet* of permissible digits. Such an illegal symbol serves as an end-marker of the useful stream digitss that constitute the *input string* over the given alphabet. As such it is often called an *"end-of-file"* marker, for short, *eof.*◄

Thus the modified URM halts if it runs out of input.

We next modify the permissible instructions of 2.1.1, displayed on p. 93. The *only* permissible instructions now is of the following type that we at first force into a "high level" programming language format, not aiming for elegance; we will achieve the latter once we formalize this model of computation that we will call a *finite automaton* (plural: *automata*)—acronym, *FA*.

Our insistence on a URM-like model for the automaton will be restricted to this brief motivational introduction and is only meant to illustrate the indebtedness of the finite automata model to the general URM model of Chapter 2, as promised above.

The typical instruction of an automaton.

$$
L: \begin{cases}
\textbf{read} \\
\textbf{if x} = a \textbf{ then goto } M' \\
\textbf{if x} = a' \textbf{ then goto } M'' \\
\vdots \\
\textbf{if x} = a^{(n)} \textbf{ then goto } M^{(n)} \\
\textbf{if x} = eof \textbf{ then halt}
\end{cases}
$$

where L and $M', \dots, M^{(n)}$ are labels—not necessarily distinct—and $a, a', \dots, a^{(n)}$ are all the possible digit values in the context of a specific URM program, that is, $\{a, a', \dots, a^{(n)}\}$ is the *input alphabet.*

The empty string, ϵ, will never be part of a FA's input alphabet.

The labels, in practice, are not restricted to be numerical or even consecutive (if numerical). However, *one* instruction's placement is significant. It is often identified by a label such as "0", or "q_0", or some such symbol and is placed at the very beginning of the program.

Pause. A finite automaton does not care where its other instructions fall, as they will be reachable by the goto-structure as needed, wherever they are.◄

The semantics of the "typical" instruction of above is first to assign to the variable **x** the just read value from some "external (to the URM) medium", and then to move to the next instruction as determined by the $a^{(i)}$s or the *eof* in the if-cases above.

Finally, we partition the instruction-labels—also called *states*—of any given restricted URM into two types: **accepting** and **rejecting**. Their role is as follows: Such a URM, when it has halted,

Pause. *When* or *if?*◄

will have finished scanning a sequence of digits—a string over its alphabet. This string is *accepted* if the program halted while in an accepting state, otherwise the input is *rejected.*

3.1 DETERMINISTIC FINITE AUTOMATA AND THEIR LANGUAGES

3.1.0.6 Example. Consider the restricted URM below that operates over the input alphabet $\{0, 1\}$

$$
0: \begin{cases} \textbf{read} \\ \textbf{if } \text{x} = 0 \textbf{ then goto } 0 \\ \textbf{if } \text{x} = 1 \textbf{ then goto } 1 \\ \textbf{if } \text{x} = eof \textbf{ then halt} \end{cases}
$$

$$
1: \begin{cases} \textbf{read} \\ \textbf{if } \text{x} = 0 \textbf{ then goto } 1 \\ \textbf{if } \text{x} = 1 \textbf{ then goto } 0 \\ \textbf{if } \text{x} = eof \textbf{ then halt} \end{cases}
$$

What does this program do? It has scanned a string of parity 0 (sum of its digits is even) and halted iff it halted while in state 0. This claim will be revisited after we formalize the automaton concept. □

3.1.1 The Flow-Diagram Model

The formalization is achieved by first abstracting a command

$$L: \ \textbf{read; if } \text{x} = a \textbf{ then goto } M \tag{1}$$

as the configuration below:
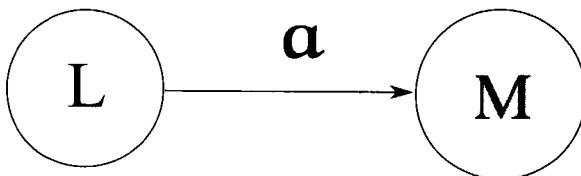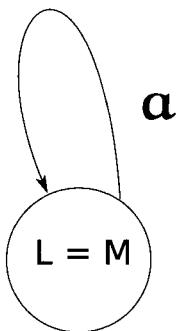
Figure capturing (1) above
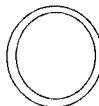
Thus the "read" part is implicit, while the labeled arrow that connects the states L and M denotes exactly the semantics of (1). Therefore, an entire automaton is a *directed graph*—that is, a finite set of (possibly) labeled circles, the *states*, and a finite set of arrows, the *transitions*, the latter labeled by members of the automaton's input alphabet. The arrows or *edges* interconnect the states. If $L = M$, then we have

the configuration
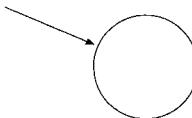
where the optional label could be L, or M, $L = M$ (as above), or nothing.

We depict the partition of states into *accepting* and *rejecting* by using two concentric circles for each accepting state as below.

The special start state is denoted by drawing an arrow, that comes from nowhere, pointing to the state.

To summarize and firm up:

3.1.1.1 Definition. (FA as Flow Diagrams) A *finite automaton*, for short, *FA*, over the *input alphabet* Σ is a finite directed graph of circular nodes—the *states*— and interconnecting edges—the *transitions*— the latter labeled by members of Σ. We impose a restriction to the automaton's structure: For every state L and every $a \in \Sigma$, there will be precisely *one* edge, labeled a, leaving L and pointing to some state M (possibly, $L = M$).

We say the automaton is *fully specified* (corresponding to the italics in the part "For every state L and every $a \in A$, *there will be* . . . ") and *deterministic* (corresponding to the italics in the part "there will be *precisely one* edge, . . . ").

This graph depiction of a FA is called its *flow diagram* and is akin to a programming "flow chart".   □

3.1.1.2 Remark. (1) Thus, full specification makes the *transition function total*— that is, for any state-input pair $\langle L, a \rangle$ as argument, it will yield some state as output. On the other hand, *determinism* ensures that the transition function is indeed a *function* (single-valued).
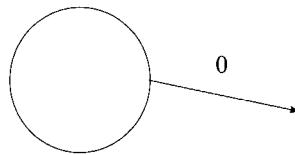
(2) Each "legal" input symbol is a member of the alphabet Σ, and vice versa. In the preamble of this chapter we referred to such legal symbols as "digits" in the interest of preserving the inheritance from the URM of Section 2.1.1, the latter being a number-theoretic programming language. But what is a "digit"? In binary notation it is one of 0 or 1. In decimal notation we have the digits $0, 1, \ldots, 9$. In *hexadecimal* notation[112] we add the "digits" a, b, c, d, e, f that have "values", in that order, $10, 11, 12, 13, 14, 15$. The objective is to have single-symbol, atomic, digits to avoid ambiguities in string notation (cf. 1.1.3.1). Thus, a "digit" is an atomic symbol (unlike "10" or "11").

We will drop the terminology "digit" from now on. Thus our automata alphabets are *finite sets of symbols*, period. □
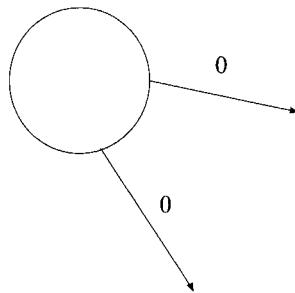
3.1.1.3 Example. Thus, if our alphabet is $A = \{0, 1\}$, then we cannot have the following configurations be part of a FA.

Nonototal Transition Function
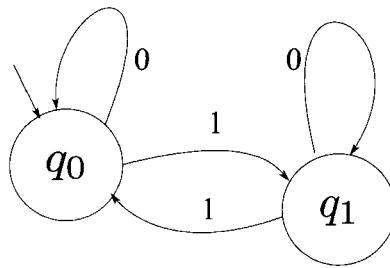
Non-determinism

□

---

[112]Base 16 notation.

3.1.1.4 Example. The FA of the example of 3.1.0.6, in flow diagram form but with no decision on which state(s) is/are accepting is given below:

We wrote q_0 and q_1 for the states "0" and "1" of 3.1.0.6.   □

Another way to define a FA without the help of flow diagrams is as follows:

3.1.1.5 Alternative Definition. (FA—Algebraically) A *finite automaton, FA,* is a toolbox $M = \langle Q, A, q_0, \delta, F \rangle$,[113] where

(1) Q is a finite set of states.

(2) A is a finite set of symbols; the *input alphabet.*

(3) $q_0 \in Q$ is the distinguished *start state.*

(4) $\delta : Q \times A \to Q$ is a total function, called the *transition function.*

(5) $F \subseteq Q$ is the set of accepting states; $Q - F$ is the set of rejecting states.   □

3.1.1.6 Remark. Let us compare Definitions 3.1.1.1 and 3.1.1.5.

(1) The set of states corresponds with the nodes of the graph (flow diagram) model. It is convenient—but not theoretically necessary in general—to actually *name* (label) the nodes with names from Q.

(2) The A in the flow diagram model is not announced separately, but can be extracted as the set of all edge labels.

(3) q_0—the start state by any name; q_0 being generic—in the graph model is recognized/indicated as the node pointed at by an arrow that emanates from no node.

(4) $\delta : Q \times A \to Q$ in the graph model is given by the arrow structure: Referring to the figure at the beginning of 3.1.1, we have $\delta(L, a) = M$.

   δ is sometimes given as a (finite) so-called transition matrix, which at row L and column a will hold M and nothing else in the illustrated case. δ being a

---

[113]"M" is generic; for "machine".

function guarantees determinism, that is, *at most one* entry in each location in the matrix. Totalness guarantees at least one entry, and hence exactly one entry, in each location.

(5) The members of F in the graph model are identified by being concentric circles.

☐

How does a FA compute? From the URM analogy, we understand the computation of a FA consisting of successive read moves, and attendant changes of state, until the program halts (by reading the *eof*). At that point we proclaim that the string formed by the stream of symbols read is *accepted* or *rejected* according as the halted machine is in an accepting or rejecting state.

To formalize this we use snapshots or IDs as we did in the full URM model of 2.1.1. The IDs of the FA are, however, very simple, since the machine (program) is incapable of altering the input stream.

3.1.1.7 Definition. (FA Computations; Acceptance) Let $M = \langle Q, A, q_0, \delta, F \rangle$ be a FA, and x be an input string—that is, a string over A that is presented as a stream of input symbols. An *M-ID* or simply *ID* related to x is a string of the form tqu, where $q \in Q$, and $x = tu$.

Intuitively, this means that the computing agent, the FA, is in state q and that the next input to process is the *first symbol* of u. If $u = \epsilon$—and hence the ID is simplified to tq—then M has halted (no more input).

Formally, an ID of the form tq has *no next ID*. We call it a *terminal ID*. However, an ID of form $tqau'$, where $a \in A$, has a *unique* next ID; this one: $ta\widetilde{q}u'$, *provided* $\delta(q, a) = \widetilde{q}$. We write

$$tqau' \vdash_M ta\widetilde{q}u'$$

or, simply (if M is understood)

$$tqau' \vdash ta\widetilde{q}u'$$

and pronounce it "(ID) $tqau'$ *yields* (ID) $ta\widetilde{q}u'$".

We say that M *accepts the string* x iff, for some $q \in F$, we have $q_0 x \vdash^*_M xq$.

The *language accepted by the FA* M is denoted generically by $L(M)$ and is the subset of A^* given by $L(M) = \{x : (\exists q \in F) q_0 x \vdash^*_M xq\}$.

An ID of the form $q_0 x$ is called a *start-ID*. ☐

3.1.1.8 Remark. (1) Of course, $\vdash^*_M$ is the *reflexive transitive closure* of $\vdash_M$ (cf. 1.6.0.23) and therefore $I \vdash^*_M J$—where I (not necessarily a start-ID) and J (not necessarily terminal) are IDs—means that $I = J$ or, for some IDs $I_m, m = 1, \ldots, k - 1$, we have

$$I \vdash_M I_1 \vdash_M I_2 \vdash_M I_3 \vdash_M \ldots \vdash_M I_{k-1} \vdash_M J \tag{1}$$

We call the disjunction of sequence (1) with "$I = J$", or the shorthand notation of this disjunction, $I \vdash^*_M J$, an *M-computation*, or simply a *computation*.
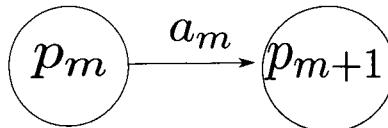
We may write (1) more informatively (but still suppressing reference to the I_j) in the shorthand $I \vdash^k J$, since there are exactly k occurrences of the relation $\vdash$ in (1).

The notations $I \vdash^{\leq n} J$ and $I \vdash^{<n} J$ mean $I \vdash^m J$ where, respectively, $m \leq n$ and $m < n$.

(2) Let $x = a_1 a_2 \cdots a_n$, where the a_j are symbols of the input alphabet $A = \{b_1, b_2, \ldots b_r\}$ of some FA M. Since

$$a_1 a_2 \cdots p_m a_m \cdots a_n \vdash a_1 a_2 \cdots a_m p_{m+1} a_{m+1} \cdots a_n$$
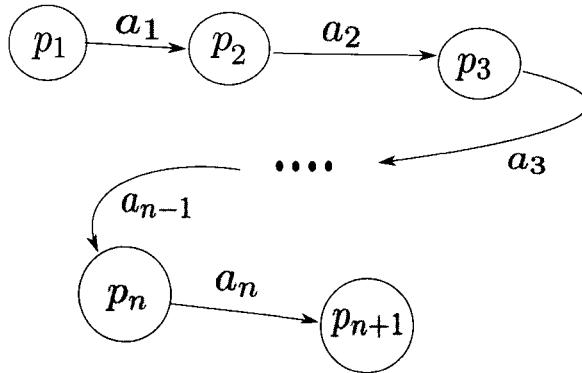
iff

is a transition that belongs to the FA, when viewed as a flow diagram, then the existence of a computation that starts at state p_1, ends at state p_{n+1} (not necessarily halting there[114]), which consumed (read) an input sequence x while doing so

$$p_1 a_1 \ldots a_n \vdash a_1 p_2 a_2 \ldots a_n \vdash a_1 \ldots p_3 a_3 \ldots a_n \vdash \text{etc}$$
$$\vdash a_1 \ldots p_m a_m \ldots a_n \vdash a_1 \ldots p_{m+1} a_{m+1} \ldots a_n \vdash \ldots \vdash a_1 \ldots a_n p_{n+1}$$

is equivalent to the existence of a labeled path—that we will aptly call a *computation path*— in the flow diagram M, from p_1 to p_{n+1} *whose labels, concatenated from left to right, form the string x*:

In particular, a string x over the input alphabet belongs to $L(M)$ iff it is formed by concatenating the labels of a path such as the above, where $p_1 = q_0$ (start state) and p_{n+1} is accepting. In this case we have an accepting path.

We see that the flowchart model of a FA is more than a static depiction of an automaton's "vital" parameters, Q, A, q_0, δ, F. *Rather, all computations, including accepting computations, are also encoded within the model as certain paths.*   □

---

[114]I.e., not necessarily meeting *eof*.

3.1.1.9 Proposition. *If M is a FA, then $\epsilon \in L(M)$ iff q_0 is an accepting state.*

Proof. First, say $\epsilon \in L(M)$. By 3.1.1.7, we have

$$q_0\epsilon \vdash^* \epsilon q \tag{1}$$

for some q that is accepting. Since $\langle q_0, \epsilon \rangle$ is not in the domain of δ (ϵ is not in the input alphabet), the only way to have (1) hold is as equality: That is, $q_0\epsilon = \epsilon q$; that is, $q_0 = q$.

Conversely, if $q_0 \in F$, then since $q_0\epsilon = q_0 = \epsilon q_0$, we have $q_0\epsilon \vdash^* \epsilon q_0$ and thus $\epsilon \in L(M)$. $\qquad\square$

3.1.1.10 Example. (Examples 3.1.0.6 and 3.1.1.4 Revisited) We prove here that if in 3.1.1.4 only q_0 is accepting, then $L(M)$ is the set of all 0-1-strings of 0 (even) parity, while if only q_1 is accepting, then $L(M)$ consists of all strings of parity 1 (odd parity).

We prove a bit more:

$$q_0 x \vdash^* x q_j, \text{ for } j = 0 \text{ or } j = 1, \text{ iff } x \text{ has parity } j \tag{1}$$

only if. We do induction on the length $|x|$ of x, to show that if

$$q_0 x \vdash^* x q_j \tag{2}$$

then x has parity j.

For $|x| = 0$ we have $x = \epsilon$, thus (2) yields $j = 0$. But indeed, the parity of ϵ is 0 as we needed to conclude.

Assume the claim for any x of length n. Let now $|x| = n + 1$.

Case where $x = y0$. From (2), we have $q_0 y0 \vdash^* yq_m 0 \vdash y0q_j$. By looking at the FA in 3.1.1.4 we see that input 0 does not cause state change, thus $m = j$.

By the I.H., y has parity m, but then so does $x = y0$; hence $j = m$ correctly gives this parity.

Case where $x = y1$. Once again, we have, from (2), $q_0 y1 \vdash^* yq_m 1 \vdash y1q_j$. By the I.H., the parity of y is m, thus the parity of $y1$ is $1 - m$. We will be done if $j = 1 - m$. This indeed is the case from the diagram of 3.1.1.4: Input 1 sends q_0 to q_1, but sends q_1 to q_0.

Conversely, the *if.* Suppose that x has parity j. By induction on the length n of x we prove that we will have (2). First, if $x = \epsilon$, then (2) becomes $q_0\epsilon \vdash^* \epsilon q_0$. Since the parity of this x is 0, we have landed on the correct state.
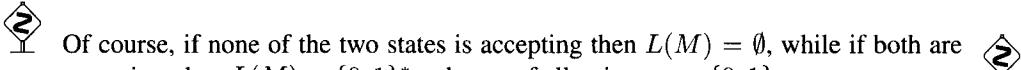
Assume now that if $|x| = n$ (fixed) has parity j, then we have (2).

Consider the case of $|x| = n + 1$, of parity j.

Case where $x = y0$. We have $q_0 y0 \vdash^* yq_m 0 \vdash y0q_r$. Now, the parity of y is j as well, so the I.H. yields that $m = j$. What is r? Well, each of q_0, q_1 is sent back to itself by input 0. Thus $r = m = j$.
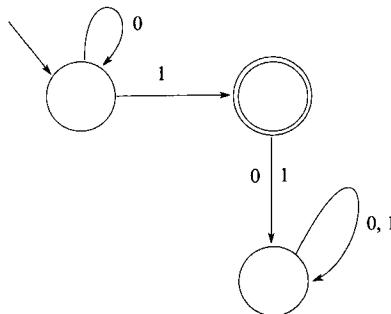
Case where $x = y1$. We have $q_0 y1 \vdash^* yq_m 1 \vdash y1q_r$. Now, parity of y is $1 - j$ thus the I.H. yields that $m = 1 - j$. What is r? Well, each of q_0, q_1 is sent to the other by input 1. Thus $r = 1 - m = j$, as needed. $\qquad\square$

Of course, if none of the two states is accepting then $L(M) = \emptyset$, while if both are accepting, then $L(M) = \{0,1\}^*$—the set of all strings over $\{0,1\}$.

3.1.1.11 Example. This example is much simpler than the preceding one. We readily see that the following automaton's only accepting paths will follow zero or more times the "loop" labeled 0 (attached to the start state), and then the edge labeled 1 to end up with an accepting state. Thus, its "$L(M)$" is the set $\{0^n1 : n \geq 0\}$, where we are reminded that for any string x, we have defined x^n by

$$x^0 = \epsilon$$
$$x^{n+1} = x^n x$$
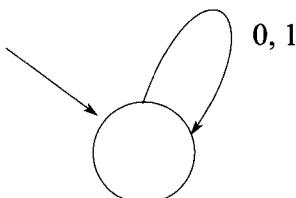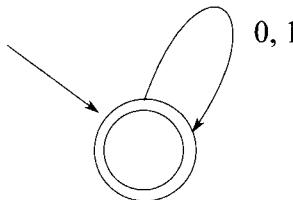
Note that we need not assign state names in this example in order to discuss what the FA does. The only purpose of the state below the accepting state is to ensure the transition function δ is total (the FA must be fully specified) as required by the definition. This state is one from which one cannot escape as once in it, all transitions lead back to it. For that reason it is often called a *trap state*.

The reader should note the use of two shorthand notations in labeling: One, we used two labels on the vertical down-pointing edge. This abbreviates the use of *two* edges going from the accepting to the trap state, one labeled 0, the other 1. We could also have used the label "0, 1" at the left or right of the arrow, "," serving as a separator. This latter notational convention was used in labeling the loop attached to the trap state.   □

3.1.1.12 Example. The two one-state FA over the input alphabet $\{0,1\}$ pictured here accept the languages $\emptyset$ and $\{0,1\}^*$, respectively.

$\square$

3.1.2 Some Closure Properties

When one introduces a class of *languages*,[115] like the sets $L(M)$, one next poses a number of interesting and fundamental questions about them. Such as, what (set-theoretic) closure properties do they have? Is the membership problem of such languages decidable?

The latter has rather a trivial answer for the $L(M)$. Given an $L(M)$—*finitely*, via a FA M—and a string, x, over the alphabet of M. The question "is $x \in L(M)$?" has an algorithmic solution: We just run M on input x. Since a FA halts on all inputs (by encountering *eof*), this task terminates. Then $x \in L(M)$ iff the state in which the computation halted is accepting.

How about closure under $\cup$? $\cap$? Complement?

3.1.2.1 Theorem. *Sets of the type $L(M)$ over a common input alphabet are closed under union.*

Proof. The proof is constructive. So let M and N be two FA over the input alphabet $A = \{a_1, \ldots, a_k\}$. Without loss of generality (since we do not have to name the states anyway), the states of M are $Q_M = \{q_0, q_1, \ldots, q_m\}$ while those of N are $Q_N = \{p_0, p_1, \ldots, p_n\}$ with q_0 and p_0 being the respective start states.

We build a new FA, let us call it K, over the same input alphabet A. Its state set is $Q_M \times Q_N$, thus the states of K are named by pairs $\langle q_i, p_j \rangle$ where $q_i \in Q_M$ and $p_j \in Q_N$.

K has a transition

$$\langle q_i, p_j \rangle \xrightarrow{a_r} \langle q_i', p_j' \rangle \tag{1}$$

iff M has the transition

$$q_i \xrightarrow{a_r} q_i' \tag{2}$$

and N has the transition

$$p_j \xrightarrow{a_r} p_j' \tag{3}$$

A state $\langle q, p \rangle$ of K is accepting, iff q is accepting in M, or p is accepting in N, or both.

---

[115]Sets of strings over an alphabet; cf. 1.1.3.2.

Let now $x = a_{j_1} a_{j_2} \ldots a_{j_i} \ldots a_{j_t}$ be an input string. Since the FA are fully specified (their δ are total), there are M- *and* N-paths

$$q_0 \xrightarrow{a_{j_1}} q_{j_1} \xrightarrow{a_{j_2}} q_{j_2} \longrightarrow \ldots \longrightarrow q_{j_{t-1}} \xrightarrow{a_{j_t}} q_{j_t} \tag{$2'$}$$

and

$$p_0 \xrightarrow{a_{j_1}} p_{j_1} \xrightarrow{a_{j_2}} p_{j_2} \longrightarrow \ldots \longrightarrow p_{j_{t-1}} \xrightarrow{a_{j_t}} p_{j_t} \tag{$3'$}$$

and a corresponding K-path labeled x, and this—by (1), (2) and (3)—is none other than

$$\langle q_0, p_0 \rangle \xrightarrow{a_{j_1}} \langle q_{j_1}, p_{j_1} \rangle \xrightarrow{a_{j_2}} \langle q_{j_2}, p_{j_2} \rangle \longrightarrow \ldots \longrightarrow \langle q_{j_{t-1}}, p_{j_{t-1}} \rangle \xrightarrow{a_{j_t}} \langle q_{j_t}, p_{j_t} \rangle \tag{$1'$}$$

Thus, we have the labeled paths $(2')$ and $(3')$ iff we have the labeled path $(1')$.

By the italicized sentence above concerning accepting states of K, we have that $(2')$ or $(3')$ (or both) are accepting paths iff so is $(1')$. That is, $L(K) = L(M) \cup L(N)$.

$\square$

3.1.2.2 Corollary. *Sets of the type $L(M)$ over a common input alphabet are closed under intersection.*

Proof. Modify the proof above so that accepting states of K are those pairs $\langle q, p \rangle$ such that q *and* p are accepting in M and N, respectively. $\square$
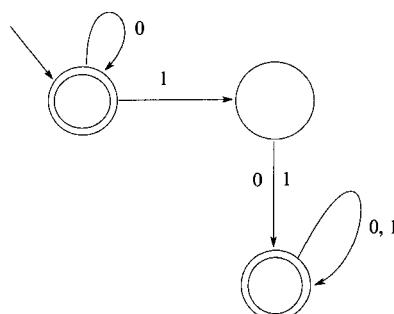
3.1.2.3 Proposition. *Sets of the type $L(M)$ over an input alphabet A are closed under complement. That is, for any M there is an N such that $L(N) = A^* - L(M)$.*

Proof. This proof is also constructive. Since $x \in L(N)$ iff $x \notin L(M)$ we simply need N to compute exactly as M—and hence to have structurally the same flow diagram as that for M—but have its accepting states be the rejecting states of M and vice versa. Indeed, under these circumstances, we have for the arbitrary input x a computation $q_0 x \vdash_N^* xq$ with q accepting iff we have $q_0 x \vdash_M^* xq$ with q rejecting.

Pause. Full specification guarantees a computation for any x.◄ $\square$

3.1.2.4 Example. The automaton that accepts the complement of the language of the FA in Example 3.1.1.11 is drawn without comment below.

$\square$

3.1.3   How to Prove that a Set is Not Acceptable by a FA; Pumping Lemma

Is there a FA M such that $L(M) = \{0^n 1^n : n \geq 0\}$? No? How can we be sure? The following theorem, known as the *pumping lemma* enables us to establish such "negative" results.

3.1.3.1 Theorem. (Pumping Lemma) *If* $S = L(M)$ *for some FA* M, *then there is a constant* C *that we will refer to as* a pumping constant *such that for any string* $x \in S$, *if* $|x| \geq C$, *then we can decompose it as* $x = uvw$ *so that*

(1) $v \neq \epsilon$

(2) $uv^i w \in S$, *for all* $i \geq 0$
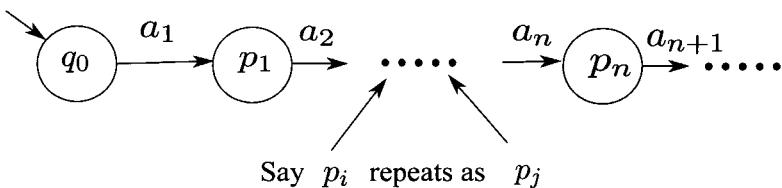
   and

(3) $|uv| \leq C$.

A pumping constant is *not* uniquely determined by S.

Proof. So, let $S = L(M)$ for some FA M of n states. We will show that if we take $C = n$[116] this will work.

Let then $x = a_1 a_2 \cdots a_n \cdots a_m$ be a string of S. As chosen, it satisfies $|x| \geq C$. An accepting computation path of M with input x looks like this:

Say p_i repeats as p_j
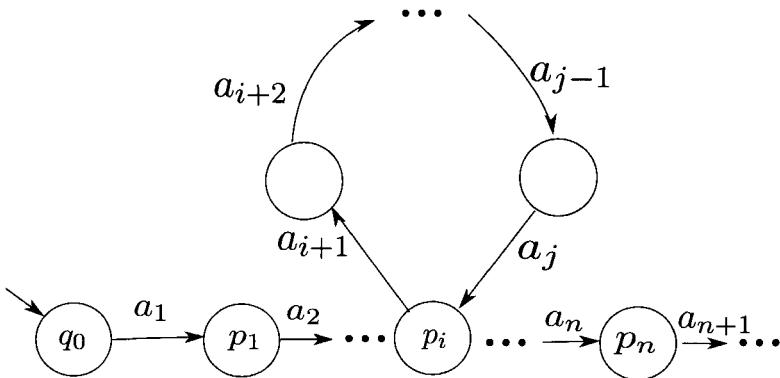
where $p_1, p_2, \ldots$ denotes a (notationally) convenient renaming of the states visited after q_0 in the computation. In the sequence

$$q_0, p_1, p_2, \ldots, p_n$$

we have named $n + 1$ states, while we only have n. Thus, at least two names refer to the same state.

---

[116] You see why C is not unique, since for any S that is an $L(M)$ we can have infinitely many different M that accept S. Can we not?

We may redraw the computation above as follows:

We can now partition x into u, v and w parts from the picture above: We set

$$u = a_1 a_2 \ldots a_i$$

$$v = a_{i+1} a_{i+2} \ldots a_j$$

and

$$w = a_{j+1} a_{j+2} \ldots a_m$$

Note that

(1) $v \neq \epsilon$, since there is at least one edge (a_{i+1}) emanating from p_i on the sub-path that connects this state to the (identical) state p_j.

(2) We may utilize the loop v zero or more times (along with u in the front and w at the tail) to always obtain an accepting path (cf. 3.1.1.8). Thus, all of uv^iw belong to $L(M)$—i.e., S.

(3) Since $|uv| = j \leq n$, we have also verified that $|uv| \leq C$.   □

The repeating pair p_i, p_j may occur anywhere between q_0 and p_n.

3.1.3.2 Example. The language $S = \{0^n 1^n : n \geq 0\}$ is not acceptable by any FA. By way of contradiction, suppose that it is, and let C be a pumping constant associated with it. Let $x = 0^C 1^C$. This is in S and satisfies $|x| \geq C$. By the pumping lemma, we have a decomposition $x = uvw$ with $|uv| \leq C$ and uv^iw in S for all $i \geq 0$. This cannot be, since uv is composed of 0s only due to its length restriction, and thus, for example, for $i = 0$, the string $uw \notin S$ as it has the wrong form: It has at least one less zero than it has ones (we miss all the zeros of v).   □

All proofs by 3.1.3.1 are by contradiction and they prove *non acceptability* by *any* FA (or, equivalently, NFA).

3.1.3.3 Example. We introduced FA as special URMs that cannot write. That in itself almost at once implies that they cannot do "arithmetic". For example, they cannot compute $\lambda x.x + 1$. "Trivially", you say, "how can they add 1 if they cannot write?"

Well, let us press on: How about accepting the language over $A = \{0, 1\}$, given by $T = \{0^n 10^{n+1} : n \geq 0\}$?

We would agree that if this is FA-acceptable, then in a roundabout way FA "know" how to add 1 and thus "compute" the *graph* of $\lambda x.x + 1$.

Alas, they cannot. Say T is FA-acceptable, and let C be an appropriate pumping constant. Choose $x = 0^C 10^{C+1}$. Splitting x as uvw with $|uv| \leq C$ we see that 1 is to the right of v. Thus, uw (using v^0) is not in T since the relation between the 0s to the left and those to the right of 1 is destroyed. This contradicts the assumption that T is FA-acceptable. □

3.1.3.4 Remark. Indeed FA cannot even compute the identity function, $\lambda x.x$, as it should be clear from 3.1.3.2: One can think that we get "n in (0s) and n out (1s)".

Another way to see that the identity function cannot be "computed" by FA is by proving, adapting the argument for T above, that $\{0^n 10^n : n \geq 0\}$ is not FA-acceptable. This coding of the identity uses the same input and output notation. □

3.1.3.5 Example. The set over the alphabet $\{0\}$ given by $P = \{0^q : q$ is a prime number$\}$ is not FA-acceptable.

Assume the contrary, and let C be an appropriate pumping constant. Let $Q \geq C$ be prime. We show that considering the string $x = 0^Q$ will lead us to a contradiction. Well, as x is longer than C, let us write—according to 3.1.3.1—$x = uvw$. By said theorem, we must have that all numbers $|u| + i|v| + |w|$, for $i \geq 0$ are prime. These numbers have the form

$$ai + b \tag{1}$$

where $a = |v| \geq 1$ and $b = |u| + |w|$. Can all these numbers in (1) (for all i) be prime?

Here is why not, and hence our contradiction: First, $b = 0$ is clearly impossible, since the numbers have the form ai, and, e.g., $a6$ is not prime.

Second, say $b \neq 0$. Now taking $i = b$, one of the numbers of the form (1) is $(a + 1)b$. So, if $b > 1$ then this is not prime (recall that $a + 1 \geq 2$). If however $b = 1$, then take $i = 2 + a$ to obtain the number (of type (1)) $a^2 + 2a + 1 = (a + 1)^2$. But this is not prime! □

The preceding shows that we can have a set that is sufficiently complex and thus fails to be FA-acceptable *even over a single-symbol alphabet*. Here is another such case.

3.1.3.6 Example. Consider $Q = \{0^{n^2} : n \geq 0\}$ over the alphabet $A = \{0\}$. It will not come as a surprise that Q is not FA-acceptable.

For suppose it is. Then, if C is an appropriate pumping constant, consider $x = 0^{C^2}$. Clearly, $x \in Q$ and is long enough. So, split it as $x = uvw$ with $|uv| \leq C$ and $v \neq \epsilon$.

Now, by 3.1.3.1,

$$uvvw \in Q \tag{1}$$

But
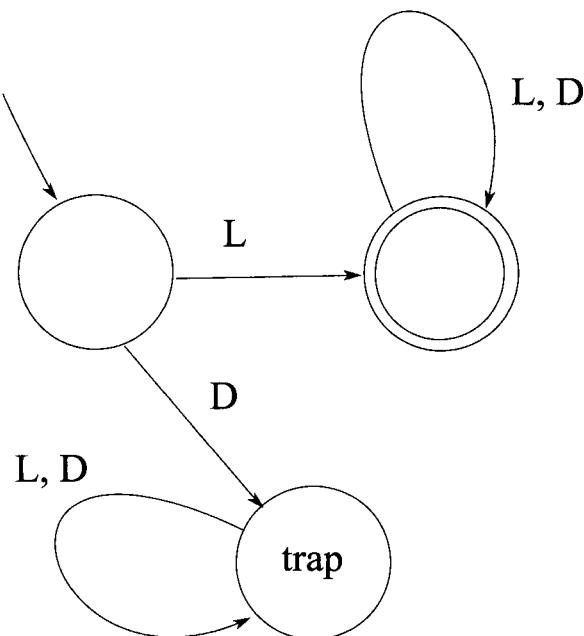
$$C^2 \leq |uvvw| \leq |uvw| + |uv| \leq C^2 + C < C^2 + 2C + 1 = (C+1)^2$$

Thus, the number $|uvvw|$ is not a perfect square being between two successive ones. But this will not do, because by (1), for some n, we have $uvvw = 0^{n^2}$ and thus $|uvvw| = n^2$—a perfect square after all! □

Hmm. Can we do *anything* useful with FA? Well, yes, for example, compilers of programming languages have an automaton front-end that will preprocess the input (which is a program written in some high level language) and extract all sorts of *tokens* such as, for example, *names of variables*. The *principle* of variable *naming* is captured by the following language (set of names) over $A = \{L, D\}$, where we chose "L" to suggest "letter" and "D" to suggest "digit": "A *name* for a variable is a string over A that starts with an L and continues with zero or more Ls or Ds".

This language is $V = \{L\}\{L, D\}^*$. An FA that accepts this language is the following:

But remember that this volume is about fundamental limitations of computing, not a how-to manual for, say, compiler writing. So we will leave this discussion at that.
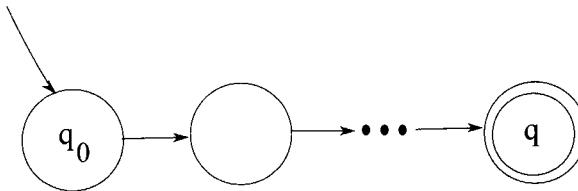
3.2   NONDETERMINISTIC FINITE AUTOMATA

The FA formalism provides us with tools to finitely define certain languages: Such a language—defined as an $L(M)$ over some alphabet A, for some FA M—contains a string x iff there is an *accepting computation*

$$q_0 x \vdash^*_M xq \tag{1}$$

or, equivalently an *accepting path* within the FA (given as a flow diagram) whose labels from left to right form x.
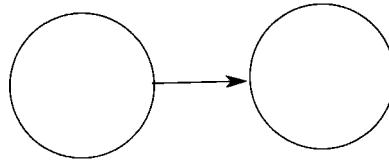
The computation (1) above, equivalently, the path labeled x within the FA, are uniquely determined by x since the automaton's δ—the *transition relation*—is a total *function*.

Much is to be gained in theoretical flexibility if we relax both the requirements that δ is single-valued (a function) and total.

This gives rise to a *nondeterministic* model of finite automata, a "NFA", that may accept string in more than one ways, that is, there may be more than one distinct paths from q_0 to an accepting state q, each labeled by the same string x.

Indeed, even more flexibility is attained if we also allow "*unconditional jumps*" from one state to another, such as
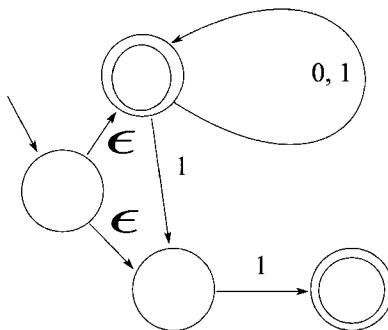
which, in the first approximation, will have unlabeled edges as above. However, in order to retain the central property that we may "read off" what string is accepted by any given accepting path, namely, *by simply concatenating the edge labels of the path from left to right*, we will label all unconditional jumps by a string that is *neutral with respect to concatenation*—that is, by ϵ. For this reason, unconditional jumps are also called "ϵ moves".

3.2.0.7 Example. The displayed flow diagram below, over the alphabet $\{0, 1\}$, incorporates all the liberties in notation and convention introduced in the preceding discussion.

We have two ϵ moves, and the string 1 can be accepted in two distinct ways: One is to follow the top ϵ move, and then go once around the loop, consuming input 1.

The other is to follow the bottom ϵ move, and then follow the transition labeled 1 to the accepting state at the bottom (reading 1 in the process).

Folklore jargon will have us speak of *guessing* when we describe what the diagram does with an input. For example, to accept the input 00 one would say that *the NFA guesses* that it should follow the upper epsilon, and then it would go twice around the top loop, on input 0 in each case.

This diagram is an example of a *nondeterministic finite automaton*, or NFA; it has ϵ moves, its transition relation—as depicted by the arrows—is not a function (e.g., the top accepting state has two distinct responses on input 1), nor is it total. For example, the bottom accepting state is not defined on any input, nor is the start state: ϵ is not an input! □

Returning to the issue of *guessing*, we emphasize that a NFA simply provides the *mathematical framework* within which *we* can formulate and verify an *existential statement* of the type

<center>*for some given input x, an accepting path* exists</center>

Given an acceptable input, the NFA does not actually guess "correct" moves (from among a set of choices), either in a hidden manner (consulting the Oracle in Delphi, for example), or in an explicit computational manner (e.g., parallelism, backtracking) toward *finding* an accepting path for said x. Simply, the NFA formalism allows *us* to *state*, and provides tools so that we can *verify*, the statement

<center>*for some given input x an accepting path* exists (3.2.0.9)   (1)</center>

just as the language of logic allows us to *state* statements such as $(\exists y).\mathscr{F}(y,x)$, and offers tools for their proof.

An independent agent, which could be ourselves or a FA—yes, we will see that every NFA can be can simulated by some FA!—can effect the verification that indeed an accepting path labeled x exists.

3.2.0.8 Definition. (NFA as Flow Diagrams) A *nondeterministic finite automaton*, or *NFA*, over the *input alphabet* Σ is a finite directed graph of circular nodes—the

states—and interconnecting edges, the *transitions*, the latter labeled by members of A. It is specified as in Definition 3.1.1.1 with some amendments:

- The restriction (for FA) that for every state L and every $a \in \Sigma$, there will be precisely *one* edge, labeled a, leaving L and pointing to some state M (possibly, $L = M$) is removed.

- The NFA *need not be fully specified*.

- It is allowed unconditional jumps, that is, edges labeled only by ϵ.

We will generically use names such as M, N, or K for NFA, just as we did for the case of FA. □

The above can be recast in an algebraic formulation, viewing a NFA from an alternative point of view as a tuple of ingredients Q, A, etc., just as we did for the FA. If one does so, one will relax the requirement that δ be a single-valued relation, and will also relax the requirement of totalness. One will also allow δ to have inputs of the type $\langle q, \epsilon \rangle$ making sure to view this *as an extension of what δ can deal with* (as inputs) rather than mistaking this as an extension of the input alphabet. Said alphabet cannot have the empty string as a member.

We will not pursue the algebraic model, as the flow diagram model will do all we want it to do.
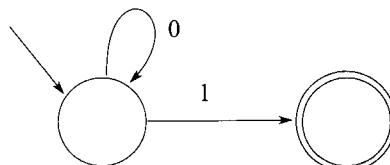
It is trivial that since a NFA is defined by relaxing requirements in 3.1.1.1, any FA is also a NFA, but not conversely, as the preceding example demonstrates.

The NFA "computes" as follows:

3.2.0.9 Definition. (NFA Computations; String Acceptance) Let M be a NFA over the input alphabet Σ. An *accepting path* is a path in M from the start state to some accepting state.
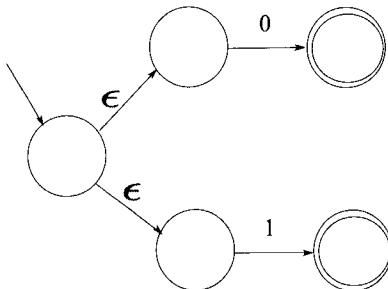
A string x over Σ is accepted by M iff x is obtained by concatenating the path labels from left to right in an accepting path. We say that x *names*, or is the name of, said accepting path. $L(M)$ denotes the set of all strings over Σ accepted by M. We say that M accepts $L(M)$. □

3.2.0.10 Example. The following is a NFA but not a FA (why?). It accepts the language $\{0\}^*\{1\}$ (cf. notation in 1.1.3.3).

□

3.2.0.11 Example. NFA are much easier to construct than FA, partly because of the convenience of the ϵ moves, and the lack of concern about single-valuedness of δ. Also, partly due to lack of concern for totalness: we do not have to worry about "installing" a trap state. For example, the following NFA over $A = \{0, 1\}$ accepts its alphabet A as we can trivially see that there are just two accepting paths: one named "0" and one named "1".

$\square$
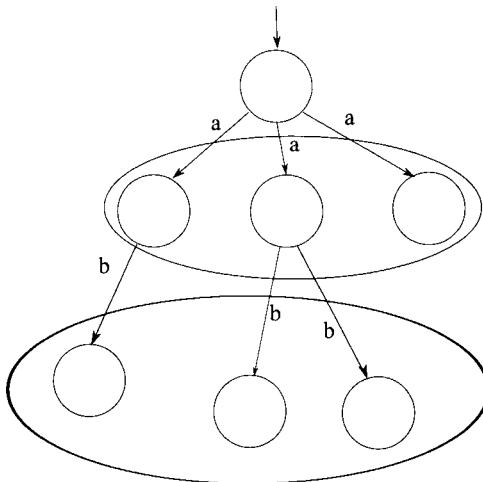
3.2.1   From FA to NFA and Back

We noted earlier that any FA is a NFA (compare Definitions 3.1.1.1 and 3.2.0.8), thus the NFA are at least as powerful as the FA. They can do all that the deterministic model can do. It is a bit of a surprise that the opposite is also true: For every NFA M we can construct a FA N, such that $L(M) = L(N)$.

Thus, in the case of these very simple machines, nondeterminism ("guessing") buys convenience, but not real power.

How does one simulate a NFA on an input x? The most straightforward idea is to trace *all possible paths* labeled x (due to nondeterminism they may be more than one—or none at all) *in parallel* and accept iff one (or more) of those is accepting.

The principle of this idea is illustrated below.

Say the input to the NFA M is $x = ab\ldots$ Suppose that a leads the start state—which is at "level 0"—to three states; we draw all three. These are at level 1. We repeat for each state at level 1 on input b: Say, for the sake of discussion, that, of the three states at level 1, the first leads to one state on input b, the second leads to two and the third leads to none. We draw these three states obtained on input b; they are at level 2, etc.

A FA can keep track of all the states at the various levels since they can be no more than the totality of states of the NFA M! The amount of information at each level is independent of the input size—i.e., it is a constant—and moreover can be coded as a single FA-state (depicted in the figure by an ellipse) that uses a "compound" name, consisting of all the NFA state names at that level. This has led to the idea that the simulating FA must have as states nodes whose names are *sets of* state names of the NFA. Clearly, for this construction, state names are important through which we can keep track of and describe what we are doing. Here are the details:

3.2.1.1 Definition. (a-**successors**) Let M be a NFA over an input alphabet Σ, q be a state, and $a \in \Sigma$. A state p is *an a-successor* of q iff there is an edge from q to p, labeled a. $\qquad\square$

In a NFA a-successors need not be unique, nor need to exist (for every pair $\langle q, a \rangle$). On the other hand, in a FA they exist and are unique.

3.2.1.2 Definition. (ϵ-**closure**) Let M be a NFA with state-set Q and let $S \subseteq Q$. The ϵ-closure of S, denoted by $\epsilon(S)$, is defined to be the smallest set that includes S but also includes all $q \in Q$, such that there is a path, named ϵ, from some $p \in S$ to q.
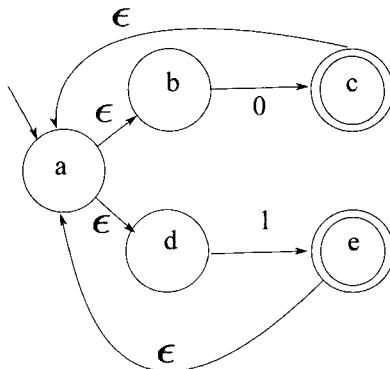
When we speak of the ϵ-closure of a state q, we mean that of the set $\{q\}$ and write $\epsilon(q)$ rather than $\epsilon(\{q\})$. $\qquad\square$

Note that a path named ϵ will have all its edges named ϵ.

3.2.1.3 Example. Consider the NFA below.

We compute some ϵ-closures: $\epsilon(a) = \{a, b, d\}$; $\epsilon(c) = \{c, a, b, d\}$.   □

3.2.1.4 Theorem. *Let M be a NFA with state set Q and input alphabet Σ. Then there is a FA N that has as state set a subset of $\mathcal{P}(Q)$—the power set of Q—and the same input alphabet as that of M. N satisfies $L(M) = L(N)$.*

We say that two automata M and N (whether both are FA or both are NFA, or we have one of each kind) are *equivalent* iff $L(M) = L(N)$. Thus, the above says that for any NFA there is an equivalent FA. In fact, this can be strengthened as the proof shows: We can *construct* the equivalent FA.

Proof. The FA N will have as state set some subset of $\mathcal{P}(Q)$, meaning that every state of N will have a compound name consisting of the names (in any order, hence *set* rather than sequence) of the members of some subset of Q. Moreover,

- The start state of N is $\epsilon(q_0)$, where q_0 is the start state of M.

- A state of N is accepting iff its name contains at least one accepting state of M.

- Let S be a state of N and let $a \in \Sigma$. The unique a-successor of S in N is constructed as follows:

  (1) Construct the set of *all* a-successors in M of *all* members of S. Call T this set of a-successors.

  (2) Construct $\epsilon(T)$; this is *the* a-successor of S in N.

As an illustration, we compute some *0-successors* in the FA constructed as above if the given NFA is that of Example 3.2.1.3.
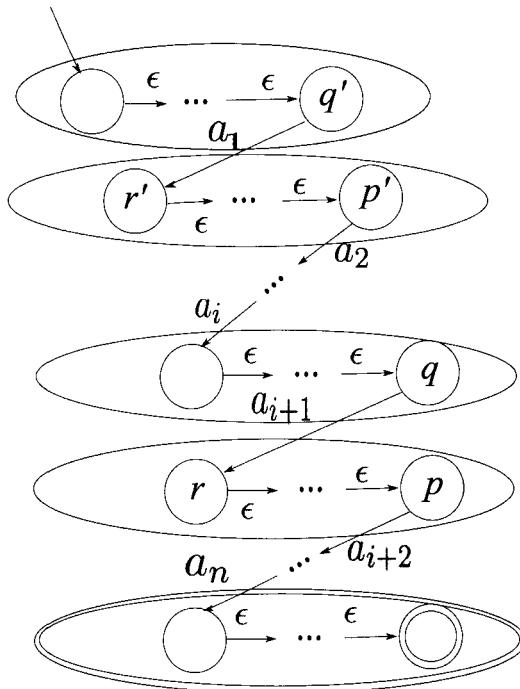
   (I) For state $\{a, b, d\}$ step (1) yields $\{c\}$. Step (2) yields the ϵ-closure of $\{c\}$: The state $\{c, a, b, d\}$ is the 0-successor.

   (II) For state $\{c, a, b, d\}$ step (1) yields $\{c\}$. Step (2) yields the ϵ-closure of $\{c\}$: The state $\{c, a, b, d\}$ is the 0-successor; that is, the 0-edge loops back to where it started.

We return to the proof. We will show that this construction of the FA N works. So let us prove first that $L(M) \subseteq L(N)$.

Let $x = a_1 a_2 \cdots a_n \in L(M)$. Without loss of generality, we have an accepting path in M that is labeled as follows:

$$\epsilon^{j_1} a_1 \epsilon^{j_2} a_2 \epsilon^{j_3} a_3 \cdots \epsilon^{j_n} a_n \epsilon^{j_{n+1}} \tag{3}$$

where each ϵ^{j_i} depicts $j_i \geq 0$ consecutive path edges, each labeled ϵ, *where $j_i = 0$ in this context means that the j_i group has no ϵ-moves.* We show this M-path graphically below as the zig-zag path with horizontal segments alternating with down-sloping segments. For easy reference, we assign the *level* number i to each horizontal part labeled $\epsilon^{j_{i+1}}$, which is the one that is followed immediately by a down-sloping edge labeled a_{i+1}.

Now the FA N that is constructed as detailed in the above three bullets will—its δ being total—have *a unique*[117] *path* labeled $a_1 a_2 \ldots a_n$ from its start state to some other state, its states being denoted by ellipses in the diagram.

The diagram *implicitly claims* that each N-state at level i, depicted as an ellipse in the diagram,

(A) contains *as part of its name* all the states that participate in the M-sub-path with name $\epsilon^{j_{i+1}}$, for $i = 0, 1, 2, \ldots$

---

[117]By determinism.

(B) the N-state at level i has the N-state shown, and *partially named*, at the next level as its a_{i+1}-successor.

We prove (A) by an easy induction on i and obtain (B) at once (bullet three on p. 262) as a side-effect. Indeed, for level $i = 0$ (basis), all the state names shown (from M) are in $\epsilon(q_0)$ and we are done by the first bullet on p. 262.

Taking as I.H. the validity of (A) for a fixed unspecified level i—where we have an N-state S—we look next at level $i + 1$, where we have an N-state T.

By definition of "level", level $i + 1$ is reached by the edge named a_{i+1}. By the assumption on how M accepts x, the edge named a_{i+1} bridges the two indicated states, q and r. From what we know about the components present in the name of S (I.H.), the third bullet that describes the transitions of N entails that all states in the ϵ-path from r to p are in the N-state T.

The string will be accepted by N iff the state pointed at by a_n—at level n—is *accepting in N*. This is so by the fact that the last state of the M-computation is accepting and—by (A)—is part of the last N-state in the above N-path.

We turn to the converse. So let $x = a_1 a_2 \ldots a_n \in L(N)$. We will argue that $x \in L(M)$.

We will reuse the above figure. Let us concentrate at first only on the elliptical states of the FA N and the indicated in the figure interconnecting transitions, which from top to bottom form the accepting N-computation path labeled $a_1 a_2 \ldots a_n$. We will want to produce a corresponding accepting M-path, that we will "fold" and fit its "horizontal" (ϵ-moves) parts inside appropriate ellipse states. This time it will be most convenient to do so starting at the bottom of the FA path and work backwards.

Thus the last (bottommost) state of N is two things:

(i) *accepting*; hence must contain an accepting M-state (as illustrated)

(ii) *the a_n-successor* of the preceding N-state (not illustrated). Therefore (p. 262), the latter *must* contain an M-state from which the a_n-edge emanates and this edge either points to the illustrated accepting M-state, or, *more generally*, to a different M-state, which is part of the last ellipse's name *and* is connected to the accepting M-state by an ϵ-path. *We always adopt the general case* (illustrated).

Continuing to build an M-computation path backwards, from an accepting state toward the start state—and "folding it" inside the ellipses of N as we go, leaving only the edges labeled a_j to connect levels—assume that we have reached the ellipse at level $i + 1$. This N-state acts on input a_{i+2} that emanates from the M-state p inside the ellipse. By the third bullet on p. 262, this p must have become part of the ellipse's name either by directly being pointed to by the a_{i+1}-edge—emanating from some M-state, that we will call q, inside the i-level ellipse—or the edge from q actually points to a state r inside the level-$i + 1$ ellipse, this r, in turn, pointing to p via an ϵ-path (the "general case"), as illustrated.
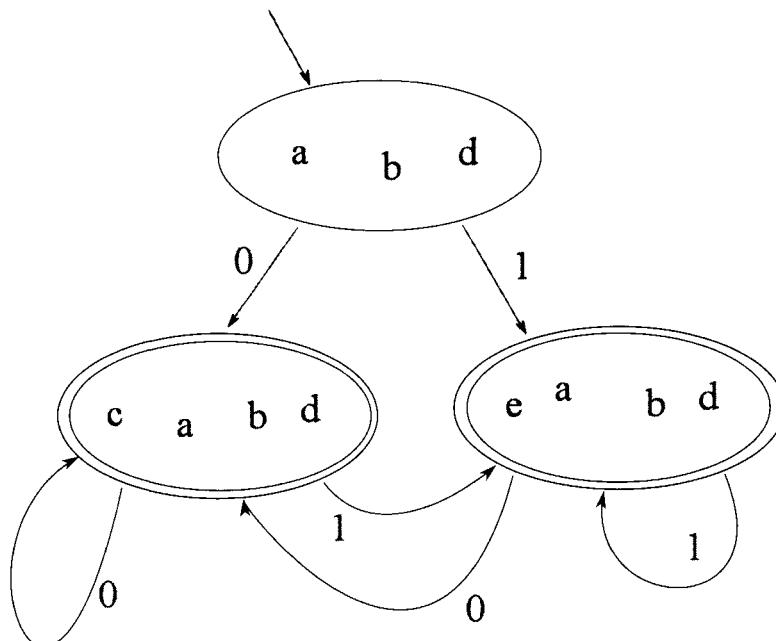
Once we reach level 1 in this way, we have a state p' (of M) as part of the level-1 ellipse's name, which—as in the general case at level $i + 1$—is either pointed to by the a_1-edge (emanating from q') directly, or indirectly via an ϵ-path from r', as

illustrated. Finally, at level 0, we have N's start state. Thus, either q' is the start state of M, or, more generally, as illustrated, q' is reachable from M's start state by an ϵ path. We have constructed an accepting path in M, labeled x. □

In theory, to construct a FA for a given NFA we draw all the states of the latter and then determine the interconnections via edges, for each state-pair and each member of the input alphabet Σ. In practice we may achieve significant economy of effort if we start building the FA "from the start state down": That is, starting with the start state (level 0) we determine *all* its a-successors, for each $a \in \Sigma$. At the end of this step we will have drawn all states at "level 1". In the next step for each state at level 1, draw its a-successors, for each $a \in \Sigma$. And so on.

This sequence of steps terminates since there are only a finite number of states in the FA and we cannot keep writing *new* ones; that is, sooner or later we will stop introducing new states: edges will point "back" to existing states. See the following example.

3.2.1.5 Example. We convert the NFA of 3.2.1.3 to a FA. See below, and review the above comment and the proof of 3.2.1.4, in particular, the three bullets on p. 262, to verify that the given is correct, and follows procedure.

You will notice the aforementioned economy of effort achieved by our process. We have only three states in the FA as opposed to the predicted 32 ($= 2^5$) of the proof of Theorem 3.2.1.4. But what happened to the other states? Why are they not listed by our procedure?

Because the procedure only constructs FA states that *are accessible by the start state via a computation path*. These are the only ones that can possibly participate in an accepting path. The others are irrelevant to accepting computations—indeed to any computations that start with the start state—and can be omitted without affecting the set accepted by the FA.   □

3.2.1.6 Example. Suppose that we have converted a NFA M into a FA N. Let a be in the input alphabet. What is the a-successor of the state named $\emptyset$ *in N*? Well, there is no M state q that connects some state in $\emptyset$ to q with label a. Thus, the set of a-successors (*in M*) of states from $\emptyset$ is itself the empty set. In other words, the a-successor of $\emptyset$ *in N* is $\emptyset$. The edge labeled a loops back to it.

Therefore, in the context of the NFA-to-FA conversion, $\emptyset$ is a *trap state* in N.   □

3.3 REGULAR EXPRESSIONS

There is a very useful alternative (to FA and NFA) way to *finitely represent* the $L(M)$-sets via a *system of notation*, or *naming*, that is called *regular expressions*. Regular expressions are familiar to users of the UNIX operating system. They are more than "just names" as they embody enough information—as we will see—to be *mechanically transformable* into a NFA (and via Theorem 3.2.1.4 to a FA).

3.3.0.7 Definition. (Regular Expressions over Σ) Given the alphabet Σ, we form the extended alphabet

$$\Sigma \cup \{\emptyset, +, \cdot, *, (,)\} \tag{1}$$

where the symbols $\emptyset, +, \cdot, *, (,)$ (not including the comma separators) are all abstract or *formal*[118] and *do not occur in* Σ. In particular, "$\emptyset$" in this alphabet is just a symbol, and so are "$+$", "$\cdot$", "$*$", and the brackets. All these symbols will be *interpreted* shortly.

The set of *regular expressions over* Σ is a set of strings over the augmented alphabet above, given as the closure $\mathrm{Cl}(\mathcal{I}, \mathcal{O})$, where

$$\mathcal{I} = \Sigma \cup \{\emptyset\}$$

and $\mathcal{O}$ contains three operations

(1) From strings α and β form the string $(\alpha + \beta)$

(2) From strings α and β form the string $(\alpha \cdot \beta)$

(3) From string α form the string (α^*)

The letters α, β, γ are used as *metavariables* (*syntactic variables*) in this definition. They will stand for *arbitrary regular expressions* (we may add primes or subscripts to increase the number of our metavariables).   □

[118]Employed to define form or structure.

3.3.0.8 Remark.

(i) We emphasize that regular expressions are built starting from the *objects* contained in $\Sigma \cup \{\emptyset\}$. We also emphasize that we have not talked about semantics yet, that is, we did not say what *sets* these expressions will represent or *name*, nor, what "+", "·", and "*" mean.

(ii) We will often omit the "dot" in $(\alpha \cdot \beta)$ and write simply $(\alpha\beta)$.

(iii) We assign the highest priority to *, the next lower to ·, and the lowest to +. We will let $\alpha \circ \alpha' \circ \alpha'' \circ \alpha'''$ group ("associate") from right to left for $\circ \in \{+, \cdot, ^*\}$.

Given these priorities, we may omit some brackets, as is usual. Thus, $\alpha + \beta\gamma^*$ means $\left(\alpha + \left(\beta(\gamma^*)\right)\right)$ and $\alpha\beta\gamma$ means $(\alpha(\beta\gamma))$. $\quad\square$

We next define what sets these expressions name (semantics).

3.3.0.9 Definition. (Regular Expression Semantics) We define the semantics of any regular expression over Σ by recursion over the set of all such regular expressions. We use the notation $L(\alpha)$ to indicate *the set named by* α.

(1) $L(\emptyset) = \emptyset$, where the left "$\emptyset$" is the symbol in the augmented alphabet (1) above, while the right "$\emptyset$" is the name of the empty set.

(2) $L(a) = \{a\}$, for each $a \in \Sigma$

(3) $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$

(4) $L(\alpha \cdot \beta) = L(\alpha)L(\beta)$—cf. Definition 1.1.3.3.

(5) $L(\alpha^*) = \left(L(\alpha)\right)^*$

A *language* over Σ (cf. 1.1.3.2) obtained as $L(\alpha)$ from some α over Σ is called a *regular language*. $\quad\square$

3.3.0.10 Example. Let $\Sigma = \{0, 1\}$. Then $L\left((0 + 1)^*\right) = \Sigma^*$. Indeed, this is because $L\left(0 + 1\right) = L(0) \cup L(1) = \{0\} \cup \{1\} = \{0, 1\} = \Sigma$. $\quad\square$

3.3.0.11 Example. We note that $L(\emptyset^*) = \left(L(\emptyset)\right)^* = \emptyset^* = \{\epsilon\}$ (cf. p. 40). $\quad\square$

Of course, two regular expressions α and β over the same alphabet Σ are equal, written $\alpha = \beta$, iff they are so *as strings*. We also have another, *semantic*, concept of regular expression "equality":

3.3.0.12 Definition. (Regular Expression Equivalence) We say that two regular expressions α and β over the same alphabet Σ are *equivalent*, written $\alpha \sim \beta$, iff they *name the same language*, that is, iff $L(\alpha) = L(\beta)$. $\quad\square$

3.3.0.13 Example. Let $\Sigma = \{0, 1\}$. Then $(0 + 1)^* \sim \big(0^*1^*\big)^*$.

Indeed, $L\big((0+1)^*\big) = \Sigma^*$, by 3.3.0.10. It suffices to show that $L\big((0^*1^*)^*\big) = \Sigma^*$ as well. To this end, the inclusion $L\big((0^*1^*)^*\big) \subseteq \Sigma^*$ is trivial, as the left hand side is a set of strings over Σ.

We turn to $L\big((0^*1^*)^*\big) \supseteq \Sigma^*$. Now let us write $A = L(0^*)$ and $B = L(1^*)$. Thus the left hand side is $(AB)^*$. Given that $A = \{0\}^* = \{0^n : n \geq 0\}$ and $B = \{1\}^* = \{1^m : m \geq 0\}$,

$$(AB)^* = \{0^n1^m : n \geq 0 \wedge m \geq 0\}^* \tag{1}$$

Let now $x \in \Sigma^*$. We show by induction on the length, k, of x that x belongs to the left hand side. For $k = 0$ we have $x = \epsilon$ and the claim follows from the definition of the Kleene star (p. 40) of any set X (here $X = AB$).

Assume the claim for a fixed k (this is the I.H.). The case for $k + 1$ has two sub-cases: First, $x = 0y$. By the I.H. and by (1), y, which has length k, has the form below.

$$0^{n_1}1^{m_1}0^{n_2}1^{m_2}0^{n_3}1^{m_3} \ldots 0^{n_r}1^{m_r}$$

But x has the same form, hence is in the left hand side.

Second, let $x = 1z$. By the I.H. and by (1), z (of length k) has the form above. But x has the same form, since it is obtained by adding 0^01^1 to the left of (1). Hence x is in the left hand side. $\square$

By the above example, $\alpha \sim \beta$ does not imply $\alpha = \beta$.

3.3.1   From a Regular Expression to NFA and Back

There is a mechanical procedure (algorithm), which from a given regular expression α constructs a NFA M so that $L(\alpha) = L(M)$, and conversely: Given a NFA M constructs a regular expression α so that $L(\alpha) = L(M)$.

We split the procedure into two directions. First, we go from a regular expression to a NFA.

3.3.1.1 Remark. Every NFA M can be considered, without loss of generality, to have exactly one accepting state. This means that, if it does not, we can *construct* an *equivalent* NFA, M', that does. Indeed, if M has no accepting states at all then we just add *one* accepting state to it, to obtain M'. *We add no edges.* Thus, just like the original, M' has no accepting paths. That is, $L(M) = L(M') = \emptyset$.

Suppose now that M has *several accepting states*,

$$P_1, P_2, \ldots, P_n \tag{1}$$

We form M' by doing this:

- add a *new* state, H, and designate it *accepting*

- connect each original accepting state to H via an ϵ-move; we add *no other edges*

- make all the P_j in (1) *rejecting*.

Trivially, M' has just one accepting state. On the other hand, say $x \in L(M)$. Then there is an accepting path in M that ends, say, with P_k, for some k. This path is labeled x. But then so is the path that we obtain by following the empty move to H: it is labeled $x\epsilon = x$. Thus, $x \in L(M')$.

Conversely, let $z \in L(M')$. Thus there is an accepting path that *must* end with H. The label of this path is z. Moreover, the path passes through, say, some P_m as this is the only way to reach H, and indeed the latter is reached from P_m via a single ϵ-move.
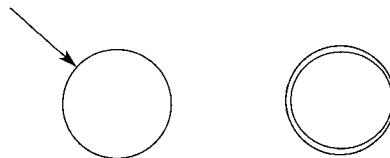
Thus, if y is the label of the portion of the path from start state to P_m, then $z = y\epsilon = y$. In other words, $z \in L(M)$, since P_m is accepting in M and therefore $y \in L(M)$. □
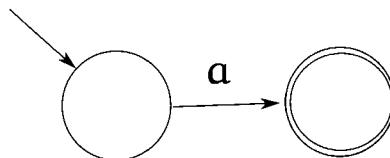
3.3.1.2 Theorem. (Kleene) *For any regular expression α over an alphabet Σ we can construct a NFA M with input alphabet Σ so that $L(\alpha) = L(M)$.*

Proof. Induction over the closure of Definition 3.3.0.7—that is, on the formation of a regular expression α according to the said definition. For the basis we consider the cases

- $\alpha = \emptyset$; the NFA below works

- $\alpha = a$, where $a \in \Sigma$; the NFA below works
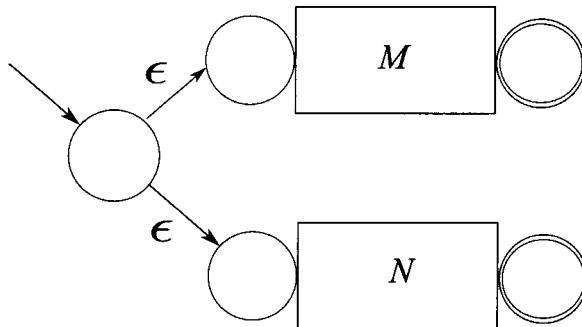
Both of the above NFA have the form guaranteed by Remark 3.3.1.1. All the NFA we construct in this proof will have that form, namely,
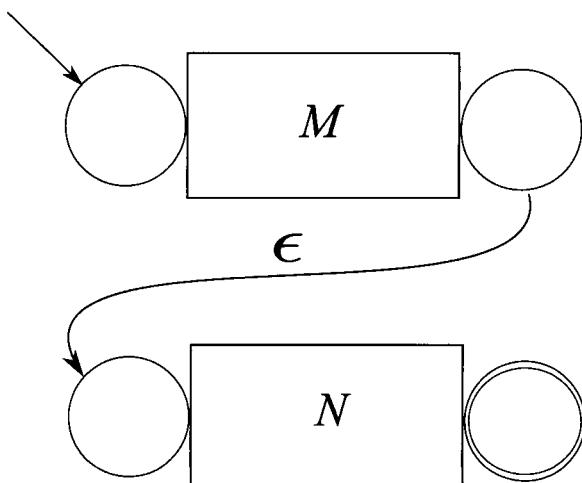
Assume now (the I.H. on regular expressions) that we have built NFA for α and β—M and N—so that $L(\alpha) = L(M)$ and $L(\beta) = L(N)$. *Moreover, these M and N have the form above.* For the induction step we have three cases:
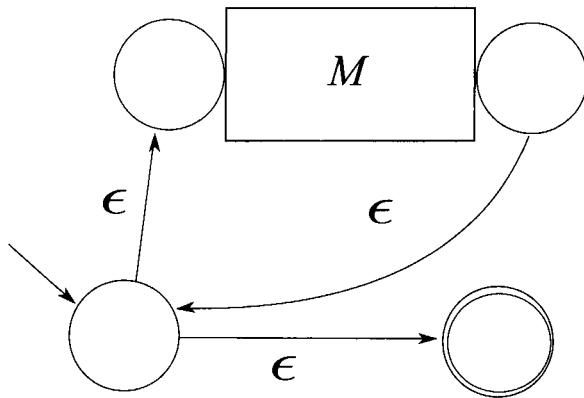
- To build a NFA for $\alpha + \beta$, that is, one that accepts the language $L(M) \cup L(N)$. The NFA below works since the accepting paths are precisely those from M and those from N. However, to maintain the single accepting state form, we apply Remark 3.3.1.1 to the NFA below.

- To build a NFA for $\alpha\beta$, that is, one that accepts the language $L(M)L(N)$. The NFA below works—since the accepting paths are precisely those formed by concatenating an accepting path of M [labeled by some $x \in L(M)$] with an ϵ-move and then with an accepting path of N [labeled by some $y \in L(N)$], in that left to right order. The ϵ that connects M and N will not affect the path name: $x\epsilon y = xy$.

- To build a NFA for α^*, that is, one that accepts the language $L(M)^*$. The NFA below, that we call P, works. That is, $L(P) = L(M)^*$.

($\supseteq$ part)   Indeed, let $x \in L(M)^*$. Then either $x = \epsilon$, or

$$x = z_1 z_2 \ldots z_k \tag{1}$$

$$\text{where each } z_j \in L(M) \tag{2}$$

In the former case, since x labels the one-edge path from the start state of P to its accepting state, it is in $L(P)$.

If, on the other hand, x is given by (1), then again P accepts it, since it has a path from its start state to the accepting state, labeled x. Here is why: this path starts at the start state of P, follows the ϵ-move to the start state of M, and then follows a path labeled z_1 in M. The path returns to the start state of P via the ϵ-move.

We repeat this path-building process, but this time we will traverse a path in M labeled z_2. We continue like this until we traverse a path labeled z_k in M and then follow the ϵ-move back to the start state of P. From here we move to the accepting state of P via an ϵ-move.

All the paths that we said we traverse in M exist by virtue of (2).

The entire path traversed via this process—in P—is labeled by

$$\epsilon z_1 \epsilon \epsilon z_2 \epsilon \ldots \epsilon z_k \epsilon \epsilon$$

which is equal to x.

($\subseteq$)   Let now, for the converse inclusion, that $y \in L(P)$. This y *labels a path from the start state of P to its accepting state.*

There are only two kinds of such paths: One is the one-edge path labeled ϵ from the start state of P directly to its accepting state. Thus $y = \epsilon$. But then, $y \in L(M)^*$.

The other kind of path is one that follows the loop that contains M *one or more times*, finally to "exit" to the accepting state by following the ϵ-move in the figure above. If we denote by w_j the name of the path traversed in M—from M's start to its accepting state—*the j-th time around the loop*, and if y caused the computation to enter the loop r times in total, then

$$y = \epsilon w_1 \epsilon \epsilon w_2 \epsilon \epsilon w_3 \epsilon \ldots \epsilon w_r \epsilon \epsilon = w_1 w_2 w_3 \ldots w_r$$

But each w_j is in $L(M)$ by virtue of what path it names in M. Hence, $y \in L(M)^*$ as we needed to prove. □

3.3.1.3 Theorem. (Kleene) *For any FA or NFA M with input alphabet Σ we can construct a regular expression α over Σ so that $L(\alpha) = L(M)$.*

Proof. Given a FA M (if a NFA is given, then we apply 3.2.1.4 first). We will construct an α with the required properties. The idea is to express $L(M)$ in terms of simple (indeed, finite) sets of strings over Σ by repeatedly using the operations $\cdot$, $\cup$ and Kleene star, a finite number of times. It will be clear that a so constructed set can be named by a regular expression.

So let $Q = \{q_1, q_2, \ldots, q_n\}$ be the set of states of M, where q_1 is the start state. We will refer to the transition function of M as δ and to the set of its accepting states as F.

We next define several sets of strings (over Σ)—denoted by R_{ij}^k, for $k = 0, 1, \ldots, n$ and each i and j ranging from 1 to n.

$$R_{ij}^k = \{x \in \Sigma^* : q_i x \vdash_M^* x q_j \text{ and every } q_m \text{ in this path}, \tag{1}$$

$$\text{other than the the endpoints } q_i \text{ and } q_j, \text{ satisfies } m \leq k\}$$

A superscript of n removes the restriction on the path $q_i x \vdash_M^* x q_j$ since every state q_m satisfies $m \leq n$.

We first note that for $k = 0$ we get very small finite sets. Indeed, since state numbering starts at 1, the condition $m \leq 0$ is false and therefore in R_{ij}^0, if we have $i \neq j$, then the condition $q_i x \vdash_M^* x q_j$ can hold precisely when $x = a \in \Sigma$ for some a—that is, iff $\delta(q_i, a) = q_j$. The case $i = j$ also allows ϵ in the set, since $q_i \epsilon \vdash_M^* \epsilon q_i$ for all i. To summarize, for all i and j we have

$$R_{ij}^0 = \begin{cases} \{a \in \Sigma : \delta(q_i, a) = q_j\} & \text{if } i \neq j \\ \{\epsilon\} \cup \{a \in \Sigma : \delta(q_i, a) = q_j\} & \text{if } i = j \end{cases} \tag{2}$$
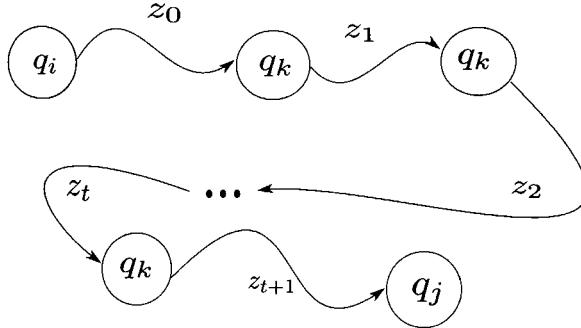
Since every *finite* set of strings can be named by a regular expression (cf. Exercise 3.5.26),

$$\text{there are regular expressions } \alpha_{ij}^0 \text{ such that } L(\alpha_{ij}^0) = R_{ij}^0, \text{ for all } i, j \tag{3}$$

Next note that the R_{ij}^k can be given recursively using k as the recursion variable and i, j as parameters, and taking (2) as the basis of the recursion.

To see this, consider a path labeled x in R_{ij}^k, for $k > 0$. It is possible that all q_m (other than q_i and q_j) that occur in the path have $m < k$. Then this x also belongs to R_{ij}^{k-1}.

If, on the other hand, we have q_k appear in the interior of the path labeled x once or more times, then we have the picture below:

where the q_k occurrences start immediately after the path named z_0 and are connected by paths named z_i, for $i = 1, \ldots, t$. Thus, $x = z_0 z_1 z_2 \ldots z_t z_{t+1}$. Noting that $z_0 \in R_{ik}^{k-1}$, $z_i \in R_{kk}^{k-1}$—for $i = 1, \ldots, t$—and $z_{t+1} \in R_{kj}^{k-1}$, we have that $x \in R_{ik}^{k-1} \cdot \left(R_{kk}^{k-1} \right)^* \cdot R_{kj}^{k-1}$.[119] We have established, for all $k \geq 1$ and all i, j, that

$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} \cdot \left(R_{kk}^{k-1} \right)^* \cdot R_{kj}^{k-1} \tag{4}$$

Now take the I.H. that for $k - 1 \geq 0$ (fixed!) and all values of i and j we have regular expressions α_{ij}^{k-1} such that $L(\alpha_{ij}^{k-1}) = R_{ij}^{k-1}$. We see that we can construct—from the α_{ij}^{k-1}—regular expressions α_{ij}^k for the R_{ij}^k. Indeed, using the I.H. and (4), we have, for all i, j and the fixed k,

$$\alpha_{ij}^k = \alpha_{ij}^{k-1} + \alpha_{ik}^{k-1} \left(\alpha_{kk}^{k-1} \right)^* \alpha_{kj}^{k-1} \tag{5}$$

Along with the basis (3) that the R_{ij}^0 sets can be named, this induction proves that *all* the R_{ij}^k can be named by regular expressions, which we may construct, from the basis up.

Finally, the set $L(M)$ can be so named. Indeed,
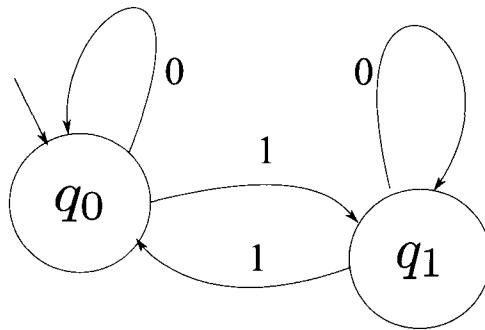
$$L(M) = \bigcup_{q_j \in F} R_{1j}^n$$

The above is a finite union (F is finite!) of sets named by α_{1j}^n with $q_j \in F$. Thus we may construct its name as the "sum" (using "+", that is) of the names α_{1j}^n with $q_j \in F$. $\qquad \square$

---

[119]If there is just one q_k in the figure, then this is captured by the $\left(R_{kk}^{k-1} \right)^0$ in the $\left(R_{kk}^{k-1} \right)^* = \bigcup_{i=0}^{\infty} \left(R_{kk}^{k-1} \right)^i$; cf. Subsection 1.1.3.

Because of the two Kleene theorems, *every* language that is accepted by a FA or NFA is a *regular language*.

3.3.1.4 Example. Consider the FA of Example 3.1.1.4, reproduced below.

We will *rename* its states q_0, q_1 to q_1, q_2 respectively (not shown), so that we can conform with the notation in the proof of Theorem 3.3.1.3.

We will compute regular expressions for:

- all sets R_{ij}^0

- all sets R_{ij}^1

- all sets R_{ij}^2

Recall the definition of the R_{ij}^k, here for $k = 0, 1, 2$ and i, j ranging in $\{1, 2\}$ (cf. proof of 3.3.1.3):

$\{x : q_i x \vdash^* xq_j$, where no state in this computation,

other than possibly the *end-points* q_i and q_j, has index higher than $k\}$

This leads to the recurrence:

$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}$$

Below I employ the abbreviated (regular expression) *name* "ϵ" for $\emptyset^*$.

R_{11}^0	$\epsilon + 0$
R_{12}^0	1
R_{21}^0	1
R_{22}^0	$\epsilon + 0$

Superscript 1 now:

	Direct Substitution
$R_{11}^1 = R_{11}^0 \cup R_{11}^0 (R_{11}^0)^* R_{11}^0$	$\epsilon + 0 + (\epsilon + 0)(\epsilon + 0)^*(\epsilon + 0)$
$R_{12}^1 = R_{12}^0 \cup R_{11}^0 (R_{11}^0)^* R_{12}^0$	$1 + (\epsilon + 0)(\epsilon + 0)^* 1$
$R_{21}^1 = R_{21}^0 \cup R_{21}^0 (R_{11}^0)^* R_{11}^0$	$1 + 1(\epsilon + 0)^*(\epsilon + 0)$
$R_{22}^1 = R_{22}^0 \cup R_{21}^0 (R_{11}^0)^* R_{12}^0$	$\epsilon + 0 + 1(\epsilon + 0)^* 1$

Using the previous table, the reader will have no difficulty to fill in the regular expressions under the heading "Direct Substitution" in the next table. To make things easier it is best to simplify the regular expressions of the previous table, meaning, finding simpler, equivalent ones. For example, $L\big(\epsilon + 0 + (\epsilon + 0)(\epsilon + 0)^*(\epsilon + 0)\big) = \{\epsilon, 0\} \cup \{\epsilon, 0\}\{\epsilon, 0\}^*\{\epsilon, 0\} = \{0\}^*$, thus

$$\epsilon + 0 + (\epsilon + 0)(\epsilon + 0)^*(\epsilon + 0) \sim 0^*$$

Superscript 2:

	Direct Substitution
$R_{11}^2 = R_{11}^1 \cup R_{12}^1 (R_{22}^1)^* R_{21}^1$	
$R_{12}^2 = R_{12}^1 \cup R_{12}^1 (R_{22}^1)^* R_{22}^1$	
$R_{21}^2 = R_{21}^0 \cup R_{22}^0 (R_{22}^1)^* R_{21}^1$	
$R_{22}^2 = R_{22}^1 \cup R_{22}^1 (R_{22}^1)^* R_{22}^1$	

□

3.3.1.5 Remark. We have the tools now to tackle the following question: Is there a converse to the pumping lemma (3.1.3.1)?

> *That is, if a language L "can be pumped", must it be regular?*

Of course, we can certify the regularity of a language L by verifying one of $L = L(M)$ or $L = L(\alpha)$, for some FA or NFA M, or some regular expression α; but *is* pump-ability an alternative tool for such certification?

First off, the colloquialism "can be pumped" means exactly this: For the given L there is a positive integer C—a pumping constant—such that, if $z \in L$ and $|z| \geq C$, then z can be decomposed as $z = uvw$ and all of (a)–(c) hold

(a) $v \neq \epsilon$
(b) $|uv| \leq C$
(c) $uv^i w \in L$, for all $i \geq 0$.

> *No. The pumping lemma goes "one way"!*

Here is why.

(i) First, we prove that the language L over $\Sigma = \{a, b, c\}$ given below

$$L = \{a^i b^j c^k : i \geq 0, j \geq 0, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\} \tag{1}$$

"can be pumped" in the sense above.

Proof.

To settle the part that a C with the required properties exists, we make an educated guess:[120] we *pick* $C = 3$, and show that this C works for the given L. That is, we prove the existential claim " there is a positive integer C, etc." *constructively*.

Pause. In earlier proofs by pumping lemma—cf. examples following 3.1.3.1— that a given language is *not* regular, we did not take care to hand-pick a C. We just said, "let C be an associated pumping constant". How come we now need to *pick* one that works?◄

We show now that all $z \in L$, where $|z| \geq 3$, "pump" in the sense (c) above. Let then z be *arbitrary* in L, and longer than 3.

    (I) Case: $z = a^1 b^j c^j$. Take $u = \epsilon$, $v = a$. Pumping up or down—i.e., $k > 0$ or $k = 0$ in $uv^k w$—we are still producing strings in L by the condition to the left of "and" in (1).

    (II) Case: $z = a^i b^j c^k$, $i \geq 2$. Take $u = aa$, v the first symbol after the second a (recall $|z| \geq 3$).

    (III) Case: z contains no a ($i = 0$). Take $u = \epsilon$, v the first symbol in the string.   □

Pause. Will either of the choices $C = 1$ or $C = 2$ work?◄

(ii) We next *prove* that L, notwithstanding that "it pumps", is **not** regular.

Pause. Hmm. How does one prove that L is not regular if the pumping lemma *does not help*? All the non regular languages in our earlier examples were proved as such by showing that "they cannot pump".◄

There are are a number of alternative techniques, that go like "if L is regular, then so is this L', obtained from L, by such and such operations that preserve regularity." We try to make L' such that its non regularity is amenable to a proof by pumping lemma.

This idea works here. *If* L is regular, then so is

$$L \cap L\left(ab^*c^*\right) = \{ab^n c^n : n \geq 0\}$$

by 3.1.2.2, using also Kleene's theorems. Now the standard pumping lemma proof shows that it is not; cf. Exercise 3.5.28.   □

Worth emphasizing. The pumping lemma is used for "negative" results of the type "L is not regular". It is used as follows:

---

[120]"Educated guesses" usually hide a lot of preliminary work and trial and error attempts; this kind of "preprocessing" is almost always omitted from proofs so that the argument is not obscured. They are not "guesses" at all.

"*If* L is regular, then it pumps". The lemma has no expectations or opinion on whether *non regular* languages are allowed to pump (or not). So, the existence of a non regular language L "that pumps" is fine, but the lemma *cannot* show *via direct application* the non regularity of this L.

3.4   REGULAR GRAMMARS AND LANGUAGES

There is yet another way to *finitely represent* a regular set: by a *grammar*—which will naturally be called a regular grammar. To motivate the core idea behind grammars, consider, for example, the (inductive) definition of formulae (2.11.0.27). Moreover, to simplify matters, let us stay in the Boolean domain—that is, we will include only the connectives $\neg$ and $\vee$ but no quantifiers—and we will also adopt as *atomic* formulae the set of Boolean variables,[121] generated by the symbol p with or without primes. Thus, the atomic formulae include

$$p, p', p''', p^{(n)}$$

where $p^{(n)}$ indicates p with n primes

$$p^{\overbrace{'' \ldots '}^{n\ \text{primes}}}$$

The alphabet over which we build these simplified well-formed (Boolean) formulae is

$$(,), \neg, \vee, p, p', p'', p''', \ldots$$

In the inductive clauses of 2.11.0.27 we have included "if $\mathscr{B}$ and $\mathscr{C}$ are formulae, then so is $(\mathscr{B} \vee \mathscr{C})$". In words this says that

> *One way to get a "complicated" formula is to take two formulae, and join them via a "$\vee$", adding outermost brackets after that.*

This generates this idea: Why not use a *metavariable*, named "$\langle$formula$\rangle$", *to stand for any formula*, generically, and retell the above italicized sentence as

$$\langle\text{formula}\rangle ::= (\langle\text{formula}\rangle \vee \langle\text{formula}\rangle)$$

The above is an instance of a *grammar rule* (or *grammar production*) in so-called *Backus-Naur Form* (or *BNF*). The symbol "::=" we read as "is defined as". The *syntactic variable* $\langle$formula$\rangle$ that names *any* formula is also called a *nonterminal symbol* or just a *nonterminal*. Symbols such as (,), and $\vee$ that name themselves are called *terminal symbols*, or just *terminals*.

A grammar is a *finite set* of BNF productions. Our entire grammar for the set of formulae has the structure below.

---

[121] A variable that, intuitively, assumes only the values *true* or *false*, the so-called Boolean values. Cf. also 1.1.1.26.

- $\langle\text{formula}\rangle ::= (\langle\text{formula}\rangle \vee \langle\text{formula}\rangle)$

- $\langle\text{formula}\rangle ::= (\neg\,\langle\text{formula}\rangle)$

- $\langle\text{formula}\rangle ::= p$

- $\langle\text{formula}\rangle ::= p'$

- $\langle\text{formula}\rangle ::= p''$

- $\langle\text{formula}\rangle ::= p'''$

$$\vdots$$

Hmm. The above listing of rules is *infinite*. How can we make it finite? By *finitely generating* the variables, rather than taking them from a ready to use, off-the-shelf infinite set! So let us use the *nonterminal* $\langle\text{var}\rangle$ to name any variable. A straightforward recursive definition, expressed as BNF rules, for this new metavariable is

$$\langle\text{var}\rangle ::= p$$

$$\langle\text{var}\rangle ::= \langle\text{var}\rangle'$$

BNF offers another notational convenience: We can group rules with the same left hand side—that is, *rules for the same nonterminal*—using the separator $\big|$ to separate right hand side alternatives. "$\big|$" is pronounced "or".

The final grammar therefore is over the alphabet

$$(,),\neg,\vee,p,{}'\qquad\qquad(*)$$

and its rules are

(1) $\langle\text{formula}\rangle ::= \langle\text{var}\rangle \,\Big|\, (\langle\text{formula}\rangle \vee \langle\text{formula}\rangle) \,\Big|\, (\neg\,\langle\text{formula}\rangle)$

(2) $\langle\text{var}\rangle ::= p \,\Big|\, \langle\text{var}\rangle'$

Every one of the two (grouped) rules defines a class of objects in the order listed: formulae, and variables. Since the "primary" object we are interested in defining is a *formula*, the nonterminal $\langle\text{formula}\rangle$ is the "important one". It is called the *start* (nonterminal) *symbol* of the grammar.

"Statically", the rules (1)–(2) recast Definition 2.11.0.27 (after the adopted above simplifications) in a new notation. Clearly, (1) says that a formula is either a variable, or is formed by using simpler (shorter) formulae. Option one is to use two simpler formulae, concatenate them so that we "sandwich" a "$\vee$" between them, and then enclose the result in brackets. Option two is to use one simpler formula, prefix it with a "$\neg$", and attach outermost brackets to the result. The subsidiary rule tells us how to finitely generate (define) variables.

Grammars do more than "statically" recasting in new notation inductive definitions such as 2.11.0.27, as we detail below.

Suppose we want to verify that a string over $(*)$ is a formula. We call such verification *parsing*. To fix ideas, say, that we are checking, specifically,

$$(p \lor (\neg p')) \tag{3}$$

We start by assuming that it *is* a formula and proceed to *verify* this *assumption* using the rules (1)–(2).

To begin with, we view that the string (3) is "*covered*" by the nonterminal $\langle$formula$\rangle$. At this highest level of abstraction (lowest level of detail), we just echo our working assumption.

We next *refine* this "covering" using a *well-chosen* rule from group (1). The choice is dictated here by the first couple of symbols in (3): The refinement or expansion of the original covering cannot be either $\langle$var$\rangle$ or the right hand side of the third rule in group (1), for the former requires the string to start with a "p" and the latter with a "$(\neg$".

The refinement yields the covering

$$(\langle\text{formula}\rangle \lor \langle\text{formula}\rangle) \tag{4}$$

We next, say, refine the left occurrence of "$\langle$formula$\rangle$" in (4), and the "goal", (3), suggests that we use the first rule of group (1), to obtain

$$(\langle\text{var}\rangle \lor \langle\text{formula}\rangle) \tag{5}$$

Let us refine "$\langle$var$\rangle$". The goal suggest we choose the first rule in group (2) to obtain

$$(p \lor \langle\text{formula}\rangle) \tag{6}$$

We have uncovered quite a bit of (3)!

Next we use the third rule of (1) to refine "$\langle$formula$\rangle$" in (7). Again, the context "$(\neg$" helps us to choose. We obtain

$$(p \lor (\neg \langle\text{formula}\rangle)) \tag{7}$$

This time we use group (1)—$\langle$formula$\rangle ::= \langle$var$\rangle$—to refine the above into

$$(p \lor (\neg \langle\text{var}\rangle)) \tag{8}$$

and then

$$(p \lor (\neg \langle\text{var}\rangle')) \tag{9}$$

using (2), second rule. One more application of (2), *first rule*, allows us to uncover our entire goal! So, indeed, (3) is a formula.

These refinements involved, in each case, the *rewriting* (replacing) of some *occurrence* of a nonterminal in some string by a right hand side of a rule for said nonterminal. For that reason we also call grammar rules *rewriting rules*.

We have a better, symbolic, way to depict the several acts of rewriting (refinement): Let us first introduce the relation "$\Longrightarrow$", pronounced *yields*, between strings over the mixed alphabet

$$(,), \neg, \vee, p, ', \langle \text{formula} \rangle, \langle \text{var} \rangle \qquad (**)$$

defined for all α, α' over the alphabet $(**)$ by

$$\alpha R \alpha' \Longrightarrow \alpha \beta \alpha' \text{ iff } R \in \{\langle \text{formula} \rangle, \langle \text{var} \rangle\} \text{ and } R ::= \beta \text{ is a rule} \qquad (\dagger)$$

Then we can summarize our rewriting sequence, from $\langle \text{formula} \rangle$ to (3), via (9), as

$$\langle \text{formula} \rangle \Longrightarrow (\langle \text{formula} \rangle \vee \langle \text{formula} \rangle) \Longrightarrow (\langle \text{var} \rangle \vee \langle \text{formula} \rangle) \Longrightarrow$$
$$(p \vee \langle \text{formula} \rangle) \Longrightarrow (p \vee (\neg \langle \text{formula} \rangle)) \Longrightarrow (p \vee (\neg \langle \text{var} \rangle)) \Longrightarrow$$
$$(p \vee (\neg \langle \text{var} \rangle')) \Longrightarrow (p \vee (\neg p'))$$

The sequence above is a *derivation*.

While our goal-driven choices worked *deterministically* in this example, it is not guaranteed that an arbitrary grammar allows us the luxury of deterministic choices during every derivation that attempts *to verify that a given string is derivable* (parsing). In general, we may have to "guess" which rule to apply in each step in order to reach our goal. Such guessing will usually be implemented by backtracking, that is, every time we make a choice that further down gets us stuck (a "bad" choice), then we go all the way back to said bad choice and make a different choice from among the available options, until we get it right—or until we have verified that the string is not derivable.

While BNF is very helpful in the definition of specific programming languages via a grammar, in particular, allowing us to utilize mnemonics for nonterminals, such as $\langle \text{if stmt} \rangle$, $\langle \text{begin block} \rangle$, etc.,[122] the metatheory of grammars benefits from a more abstract, and simpler, notational convention as detailed in the definition below.

3.4.0.6 Definition. (Grammars) A grammar $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$ is a toolbox where $\mathcal{V}$ is a finite set of *nonterminals*, Σ is a finite set of *terminals*, where $\mathcal{V} \cap \Sigma = \emptyset$.

$S \in \mathcal{V}$ is the *start symbol*, and $\mathcal{R}$ is the set of *rewriting rules* or *productions*. The symbols in $\mathcal{V}$ are denoted by upper case (*single*) Latin letters. The symbols of Σ are single-character symbols that are *not* Latin capital letters. The rules are of the form $A \rightarrow \alpha$, where $\alpha \in (\mathcal{V} \cup \Sigma)^*$. Note that in the domain of the metatheory of grammars we have simplified the BNF notation "$::=$" to "$\rightarrow$".

Strings over $\mathcal{V} \cup \Sigma$ will have the generic names $\alpha, \beta, \gamma, \delta$ with or without primes, while strings over Σ will have the generic names x, y, z, u, v, w with or without primes. Consistent with this convention, symbols of Σ—if they are not non-letters, such as $0, 1, 9, \dagger, \#$, etc.—will be denoted by a, b, c, d with or without primes.

---

[122]BNF also allows multi-character mnemonics for terminals, using a variety of notations. For example, the terminal "then" of Algol might be denoted by **then**, or *then*, or <u>then</u>.

We define a relation $\Longrightarrow$ on $(\mathcal{V} \cup \Sigma)^*$, pronounced *yields*, by $\alpha \Longrightarrow \beta$ iff, for some γ and γ', we have $\alpha = \gamma A \gamma'$ and $\beta = \gamma \delta \gamma'$ where $A \to \delta$ is in $\mathcal{R}$.
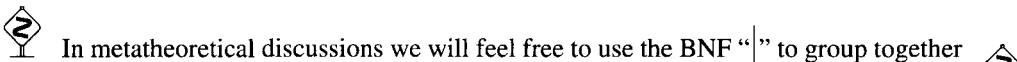
The language generated by G—in symbols $L(G)$—is the set $\{x \in \Sigma^* : S \Longrightarrow^* x\}$.[123]

An α such that $S \Longrightarrow^* \alpha$ is called a *sentential form*. A chain

$$A \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \alpha_3 \ldots \Longrightarrow \alpha$$

is a *derivation* of α *from* A. If A is the start symbol, then we just say "a derivation".

If $\alpha \in \Sigma^*$—in which case we prefer to write something like x instead of α—then it is called a *sentence* (of G). Thus, $L(G)$ is the set of all sentences of G. $\qquad \square$

In metatheoretical discussions we will feel free to use the BNF "$|$" to group together rules with the same left hand side.

3.4.0.7 Definition. The grammar defined above is called a *type-2* or *context free* grammar, for short, CFG. The corresponding language $L(G)$ is said to be a type-2 or *context free language*, for short CFL. The *type* of grammar is determined by the fact that the left hand side of every rule is a *single nonterminal*.

If the right hand side of every rule of a CFG is restricted to be of any of the four forms ϵ, a, B, or aB, where- $a \in \Sigma$ and $B \in \mathcal{V}$, then the grammar is said to be a *type-3* or *regular grammar*. $\qquad \square$

CFGs and CFLs will be studied further in Section 4.3.

3.4.0.8 Example. Let $\mathcal{V} = \{S\}$, $\Sigma = \{'\}$ and the rules are $S \to '\big|'S$. This grammar is regular. $L(G)$ is clearly

$$\Sigma^+ = \{', '', ''', \ldots\}$$

that is, the set of all non-empty strings of primes.

How "clearly"? Well, by $L(G)$'s definition, $L(G) \subseteq \Sigma^*$, but no rule can lead to the generation of ϵ (the length of right hand sides of the two rules is ≥ 1). Thus $L(G) \subseteq \Sigma^+$.

For the converse inclusion, $\Sigma^+ \subseteq L(G)$, we do induction on the length of strings $x \in \Sigma^+$. For the basis, $|x| = 1$, we are looking at $x = '$. But $S \Longrightarrow '$. Assume now (I.H.) that if $|x| = n$, then we have $S \Longrightarrow^* x$. What about a y of length $n + 1$? Well, $y = 'x$, with $|x| = n$. Thus,

$$S \overset{\text{rule } S \to 'S}{\Longrightarrow} 'S \overset{\text{I.H.}}{\Longrightarrow^*} 'x \qquad \square$$

---

[123]Cf. 1.6.0.20 and 1.6.0.23.

3.4.1   From a Regular Grammar to a NFA and Back

In this section we, *post facto*, justify the terminology *regular* grammar and *regular* language by proving the two theorems below, which relate the descriptive powers of NFA and type-3 grammars.

3.4.1.1 Lemma. *A sentential form $\alpha \notin \Sigma^*$ of a regular grammar $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$ has the form xA where $x \in \Sigma^*$ and A is a nonterminal.*

Proof. Induction on the length n of the derivation $S \Longrightarrow^n \alpha$ (cf. 1.2.0.21). The basis, for $n = 0$, is immediate since then $S = \alpha$. We take the obvious I.H. and consider

$$S \Longrightarrow^{n+1} \alpha \tag{1}$$

Now, (1) can be rewritten, using the I.H., as

$$S \Longrightarrow^n xA \Longrightarrow xaB$$

where $a \in \Sigma \cup \{\epsilon\}$ and B is a nonterminal.

   Pause. Why can it not be that the $(n + 1)$-st step used a rule of the type $A \to a$, where $a \in \Sigma \cup \{\epsilon\}$?◀

   We are done.   □

3.4.1.2 Theorem. *For any regular grammar $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$ we can construct a NFA $M = \langle Q, \Sigma, q_0, \delta, F \rangle$ such that $L(G) = L(M)$.*

Proof. We take $Q = \mathcal{V} \cup \{\checkmark\}$, where the "checkmark", $\checkmark$, is a *new* symbol (not in $\mathcal{V} \cup \Sigma$) that names the unique accepting state of M—that is, $F = \{\checkmark\}$. We take $q_0 = S$. The transitions (the δ) are chosen as follows. We have a transition

(1)
$$A \xrightarrow{a} \checkmark$$

   iff $A \to a$ is a rule, where $a \in \Sigma \cup \{\epsilon\}$

(2)
$$A \xrightarrow{\epsilon} B$$

   iff $A \to B$ is a rule

(3)
$$A \xrightarrow{a} B$$

   iff $A \to aB$ is a rule, where $a \in \Sigma$

We will show that $L(G) = L(M)$. First we address $\subseteq$. We will assume that $x \in L(G)$ and prove that $x \in L(M)$. As it is natural to do induction on the derivation length of x, it is more convenient to show instead (cf. preceding lemma) that, for any $y \in \Sigma^*$ and $A \in \mathcal{V}$,

$$\text{if } S \Longrightarrow^n yA, \text{ then there is a path in } M, \text{ labeled } y, \text{ from } S \text{ to } A \tag{4}$$

The case for $n = 1$ means that we have a grammar rule $S \rightarrow yA$, where $y \in \Sigma \cup \{\epsilon\}$ and A is a nonterminal. If y is a terminal then we have a NFA move of type (3) above, else we have one of type (2). We have settled the basis since in either case we have a one-edge path from S to A labeled y.

We now fix n and take (4) as the I.H. Let us next consider a derivation of length $n + 1$:

$$S \Longrightarrow^n yA \Longrightarrow yaB$$

By the I.H. there is a path in M, labeled y, from S to A. This path continues from A to B, with an edge labeled a ((2) or (3) above). Overall, we have a path from S to B labeled ya. We are done in our task.

If now $S \Longrightarrow^* x \in \Sigma^*$, then (as $S \neq x$) we must have

$$S \Longrightarrow^* yA \Longrightarrow ya$$

where $x = ya$ and $a \in \Sigma \cup \{\epsilon\}$. The last step of the derivation must be due to the rule $A \rightarrow a$. By what we proved above we have a path labeled y in M from S to A. The rule $A \rightarrow a$ contributes to the tail-end of the path the edge (1) above. The augmented path is labeled x and it ends at the accepting state. Done.

For the other direction, let

$$x \in L(M) \tag{5}$$

We want to show that $x \in L(G)$. Correspondingly with the previous direction, we will prove the following converse of (4), by induction on computation path length in M, for all $y \in \Sigma^*$ and $A \in Q - F$.

if there is a path of length n in M labeled y, from S to A, then $S \Longrightarrow^* yA$ (6)

For $n = 1$, there is *one* edge in M

$$S \xrightarrow{y} A$$

thus $y = a \in \Sigma$ or $y = \epsilon$. By (2) and (3) above, $S \rightarrow yA$ is a rule, thus $S \Longrightarrow yA$. We next fix the n and take (6) as the I.H. Suppose now that there is a path of length $n + 1$ from S to A in M, labeled x. Let a be the label of the *rightmost edge* of the path, connecting B to A, where $a \in \Sigma \cup \{\epsilon\}$. By (2) and (3),

$$B \xrightarrow{a} A \text{ is a rule} \tag{7}$$

and, by I.H., we have $S \Longrightarrow^* yB$, where $x = ya$. Since by (7) we have $B \Longrightarrow aA$, we get a derivation

$$S \Longrightarrow^* yB \Longrightarrow yaA$$

as needed.

With the induction completed, let us return to the hypothesis (5). It means that we have a path from S to $\checkmark$, labeled x. If the path has just one edge, then we have $S \rightarrow x$ as a rule [cf. (1) above]. Hence $S \Longrightarrow x$ as needed. If the path has length at least 2, then its nodes are $S, \ldots, A, \checkmark$, for some $A \in Q - F$. Say, the edge from A to $\checkmark$ is

labeled $a \in \Sigma \cup \{\epsilon\}$—thus $A \to a$ is a rule. By what we just proved by induction, it is $S \Longrightarrow^* yA$, where $x = ya$, and, of course, $yA \Longrightarrow ya$. Thus $S \Longrightarrow^* x$ and we are done. $\square$

3.4.1.3 Theorem. *For any NFA* $M = \langle Q, \Sigma, q_0, \delta, F \rangle$ *we can construct a regular grammar* $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$ *such that* $L(G) = L(M)$.

Proof. First off, without loss of generality the names of the rejecting states are capital Latin letters, with S as the start state that was generically denoted as "q_0" in the toolbox of M above. We also assume that we have "preprocessed" the given NFA by *adding a new, unique* accepting state that we name $\checkmark$ (cf. Remark 3.3.1.1). *We do so even if the original NFA had a unique accepting state.*

The construction of G and the proof that it behaves as stated is entirely analogous to that of 3.4.1.2. Thus, we take $\mathcal{V} = Q - F$.

The rules of G are chosen exactly as in (1)–(3) in the preceding proof, reordering the two sides of each "iff" to emphasize that it is the NFA that is *given* this time: We have a rule

(1) $A \to a$ is a rule, where $a \in \Sigma \cup \{\epsilon\}$, iff

$$A \xrightarrow{\ a\ } \checkmark$$

is a transition

(2) $A \to B$ is a rule iff

$$A \xrightarrow{\ \epsilon\ } B$$

is a transition

(3) $A \to aB$ is a rule, where $a \in \Sigma$, iff
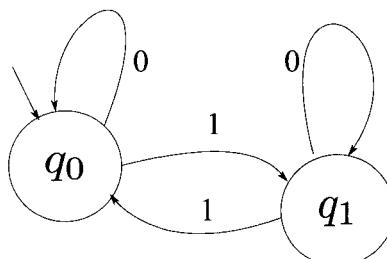
$$A \xrightarrow{\ a\ } B$$

is a transition
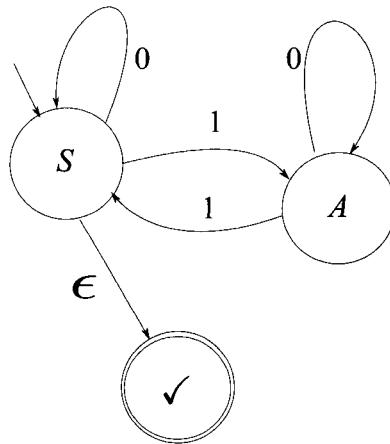
The proof that $L(G) = L(M)$ can now be read off the proof of 3.4.1.2. $\square$

Since every NFA is also a FA, the above construction, 3.4.1.3, works if we start with a FA.

3.4.1.4 Example. Let us construct a regular grammar for the FA of Example 3.1.1.4, which we reproduce below:

To fix ideas, let us assume that we have designated q_0 as accepting and q_1 is rejecting. Preprocessing gives the NFA below:

The proof of 3.4.1.3 gives us the rules

$$S \to \epsilon \,\big|\, 0S \,\big|\, 1A \text{ and } A \to 0A \,\big|\, 1S \qquad\qquad \square$$

3.4.2   Epilogue on Regular Languages

We look at some ad hoc additional results on regular languages, now that we have three equivalent ways to finitely define them: automata, regular expressions, and regular grammars. We will present these in the format of examples and will make no effort to be "complete" given our philosophy that was outlined in the Preface of this volume. The reader who wants to explore more on automata and their languages may wish to refer to any of Sipser (1997), Lewis and Papadimitriou (1998), and Hopcroft *et al.* (2007).

3.4.2.1 Example. Every regular language L over an alphabet Σ is closed under *taking prefixes* of its members. That is, the language

$$Init(L) = \{x : \text{for some } y, xy \in L\} \tag{1}$$

is also regular. To show regularity, we will look for an α or an M or a G and show that the language in (1) is finitely defined by one of these. Which one we choose is simply a matter of conveniewnce!

So let us take the regular expression avenue toward the proof, and translate the assumption to

For some α over Σ, we have $L = L(\alpha)$

By induction on the formation of α (cf. 3.3.0.7 and 1.6.0.13) we argue that we can find (*for any*) α such that $L = L(\alpha)$ a $\widetilde{\alpha}$ such that $Init(L) = L(\widetilde{\alpha})$.

Basis.

1. $\alpha = a \in \Sigma$. Then $\widetilde{\alpha} = \alpha + \emptyset^*$.[124]

2. $\alpha = \emptyset$. Then $\widetilde{\alpha} = \alpha$.

Induction steps. If $\alpha = \beta + \gamma$, then $\widetilde{\alpha} = \widetilde{\beta} + \widetilde{\gamma}$.
   If $\alpha = \beta \cdot \gamma$, then $\widetilde{\alpha} = \widetilde{\beta} + \beta \cdot \widetilde{\gamma}$.
   If $\alpha = \beta^*$, then $\widetilde{\alpha} = \beta^* \widetilde{\beta}$. $\square$

Another closure property is left as an exercise (Exercise 3.5.30).

3.4.2.2 Example. Given a regular language in one of the three ways that we know of, can we check algorithmically whether it is equal to $\emptyset$? Yes! We use the FA formalism to describe it (if it is not given that way we can certainly convert to this representation) and then check to see if there is any path from the start state to any accepting state. The language is empty iff no such path exists. $\square$

3.4.2.3 Example. Given a regular language, L, can we check algorithmically whether it is finite or not?
   Yes! We assume without loss of generality that the language has been given as $L = L(M)$ for some FA M. This allows us to determine a pumping constant C (it equals the number of states in M; cf. proof of 3.1.3.1).
   The language is infinite iff it contains strings with lengths greater than or equal to C. But how can we tell whether it does?
   Well, if we have infinitely many strings in L, then L *must* have strings of arbitrarily large lengths, hence, in particular, a string z of *smallest possible length*, but nevertheless $|z| \geq 2C$. By 3.1.3.1, we can write this z as uvw with $|uv| \leq C$ and such that $uv^i w \in L$ for $i \geq 0$. In particular, $uw \in L$.
   As z had least length above $2C$, we have that

$$|uw| < 2C \tag{1}$$

Now, adding to $2C \leq |uvw|$ the inequality $-C \leq -|v|$ [cf. 3.1.3.1, (3)] we get

$$C \leq |uw| \tag{2}$$

So, if L is infinite, then we will be able to find an $x \in L$ of length—by (1) and (2)

$$C \leq |x| < 2C \tag{3}$$

Of course we will have to *examine only finitely many strings*, those in the length-range (3), to find such an x.
   But is finding an x in the language L, in the length-range (3), *sufficient* in order for us to proclaim that L is infinite? Yes, because the left inequality of (3) and 3.1.3.1

---

[124]The set of prefixes of members of $\{a\}$ is $\{\epsilon, a\}$. But $\{\epsilon\}$ is given by the name $\emptyset^*$—cf. 3.3.0.11.

imply that this x "pumps"; it leads to infinitely many strings in L. So this is the algorithm:

Using the FA M, check all strings in the range (3) for membership in L. If any is found, then L is infinite; otherwise it is finite.   □

3.5 ADDITIONAL EXERCISES

1. Describe in set-theoretic notation the language of the automaton in 3.1.2.4. Independently, describe set-theoretically the complement $\{0^n 1 : n \geq 0\}$ and verify that your two answers are equivalent.

2. Construct a FA M over $A = \{0, 1\}$ such that $L(M) = \{\epsilon\}$.

3. Construct a FA M over $A = \{0, 1\}$ such that $L(M) = \{0, 1\}^+$, that is, $L(M) = A^* - \{\epsilon\}$.

4. Construct a FA that accepts $L = \{00, 11, 10\}$ over $A = \{0, 1\}$.

5. Let $\Sigma = \{0\}$. Which of the following languages over Σ is regular, *and why*?

   (a) $\{x : |xx| \text{ is odd}\}$
   (b) $\{x : |x| \text{ is odd}\}$
   (c) $\{x : |xx| \text{ is not a prime}\}$
   (d) $\{x : |x| \text{ is not a prime}\}$
   (e) $\{x : |x| \text{ is a perfect cube}\}$

6. Find a finite automaton that accepts the language over $A = \{0, 1\}$ that contains precisely the strings that have no three consecutive 0s.

7. Find a finite automaton that accepts the language over $A = \{0, 1\}$ that contains precisely the strings that end in precisely three consecutive 0s.

8. Find a finite automaton that accepts the language over $A = \{0, 1\}$ that contains precisely the strings that end in at least three consecutive 0s.

9. Design a FA over $\{0, 1\}$ that accepts exactly all the strings of *length* $3k + 1$ for some natural number k.

   E.g., 0, 0110, 0000 are all in. 00, 000, 01101 are not.

10. Build a NFA that accepts precisely all the strings over $\{0, 1\}$ of length ≥ 5 that contain at least one "1" among their last five symbols.

    You should argue the correctness of your design in general terms, not by example.

11. Design a FA over $\{0, 1\}$ that accepts exactly all the strings whose digits have *sum* equal to $3k + 1$ for some natural number k.

For example, 1, 100, 1111 are in. 11, 0, 111 are not.

You must prove that your automaton works!

12. Convert to NFA (all are over $\{0, 1\}$) without comment:

   - 01*

   - (0+1)01

   - 00(0+1)*

13. Convert the previous last two NFA to a FA.

14. For any string x over $\Sigma = \{0, 1\}$, let x^R mean its *reversal* (i.e., x^R reads right-to-left exactly as x does left-to-right), or, mathematically,

$$x^R = \begin{cases} x & \text{if } x = \epsilon \\ y^R a, \text{ for all } a \in \Sigma & \text{if } x = ay \end{cases}$$

Is $\{xwx^R : x \in \Sigma^* \wedge w \in \Sigma^*\}$ regular?

A proof should be provided in support of either of the two possible answers.

15. *This is a different question than the previous one!*

   Let L over $\Sigma = \{0, 1\}$ be given by $L = \{xyx^R : x \in \Sigma^* \wedge y \in \Sigma\}$.

   Prove or *disprove*: L is regular.

16. Prove that the language over $\{0, 1\}$,

   $L = \{w : w$ contains an equal number of occurrences of the substrings 01 and $10\}$

   is regular.

   Overlaps are allowed in the occurrences of 01 and 10. E.g., 010 is in. 0110 is in too, but 1010 is not.

   "Prove" is *not* fulfilled by just writing down a NFA (or FA or a regular expression). The question remains: *Why* does the proposed finite description represent L?

   Put positively, you need first to prove that L, as given above, can be *redefined* by some *much simpler* description. Then it will be easy to give a NFA or FA or a regular expression that can readily be seen to "work"!

17. Prove that the language A below, over the alphabet $\Sigma = \{0, 1, +, =\}$,

   $A = \{x + y = z : x, y, z$ are binary integers, and z is the sum of x and $y\}$

   is not regular.

18. *A counterexample to the Pumping Lemma*: "We know that $L = \{00, 01\}$ over $A = \{0, 1\}$ is regular. Let then $C > 0$ be a pumping constant for L, and take a $z \in L$ with $|z| \geq C$. Then we can write $z = uvw$—where $|v| \neq 0$—so that all the strings $uv^i w$, for $i \geq 0$, are in L. But this is impossible! L is finite!"

The reasoning in quotes invalidates the pumping lemma! *Do you believe this?*

(i) We said "We know that $L = \{00, 01\}$ over $\Sigma = \{0, 1\}$ is regular". How do we *know* this? Explain precisely.

(ii) Explain precisely *where* (and why) the counterexample argument is wrong.

19. Is the language $L = \{a^{2n} : n \geq 3\}$ over $\{a\}$ regular? Why?

20. Is the language $L = \{a^{2^n} : n \geq 3\}$ over $\{a\}$ regular? Why?

21. Is the language $L = \{a^x b^y c^z : x \geq y \vee y \geq z\}$ regular? Why?

22. Is the language $L = \{a^n (b^i a^j)^k b^n : i, j, k \geq 0 \wedge n \geq 1\}$ regular? Why?

23. Let M be a NFA. "To get a NFA that accepts the complement of $L(M)$ it suffices to swap accepting and rejecting states".

If you believe this prove it. If not, find a counterexample, that is, a NFA M on which the above suggested swapping does not work: It produces a NFA that accepts something other than the complement of $L(M)$.

24. Define a new type of a nondeterministic automaton exactly as in Section 3.2, *except* that the new model—in general—has *more than one* start state.

A string x is accepted by this model iff there is a path with label x *from some initial state* to some state in F.

Prove that this model still recognizes exactly the regular languages.

25. For any regular expressions α and β over the alphabet Σ, we have $(\alpha + \beta)^* \sim (\alpha^* \beta^*)^*$.

26. Prove that every finite set of strings over some alphabet A can be named by a regular expression.

27. There were a number of "pauses" in 3.3.1.5 posing questions to the reader. Provide all needed answers.

28. Prove that $\{ab^n c^n : n \geq 0\}$ over $\Sigma = \{a, b, c\}$ is not regular.

29. Prove the result in Example 3.4.2.1 using the NFA, rather than regular expressions, as the formalism.

30. Prove that regular languages are *closed under reversal*. This is understood as follows: The *reversal* of a string x over Σ, denoted by x^R, is defined in Exercise 14 above.

For any language L, its *reversal*, denoted by L^R is defined as $L^R = \{x^R : x \in L\}$. The exercise asks to prove that if L is regular, then so is L^R.

Hint. A technique similar to that used in 3.4.2.1 is recommended.

31. Prove that a CFG G over Σ, whose rules are of the types $A \to B$, $A \to a$, and $A \to Ba$—where $a \in \Sigma \cup \{\epsilon\}$—is regular.

Hint. This means that an α, or a FA M, or a regular grammar G' exist, such that the generated language equals $L(\alpha)$ [or $L(M)$, or $L(G')$].

32. Prove (by an appropriate example) that we cannot normally mix rules of the types $A \to Ba$ and $A \to aB$ and expect to generate a regular language. More precisely, find a CFG over Σ with rules of types $A \to a$ (where $a \in \Sigma \cup \{\epsilon\}$), $A \to B$, $A \to aB$, and $A \to Ba$ that generates a language that we *know* is not regular.

33. Provide an algorithm that checks whether or not

$$(\exists x)(x \notin L_1 \cup L_2)$$

for any given regular languages L_1, L_2 and string x, all over some fixed Σ.

34. We know that if L and L' are regular languages over Σ then so is $L \cup L'$. By induction on n prove that this extends to any number of regular languages $L_1, \ldots, L_n$.

35. Does the above extend to infinitely many regular languages L_i? Will $\bigcup_{i=1}^{\infty} L_i$ be regular?

Thoroughly justify your answer (a proof, if you said "yes"; a counterexample if you say "no").

36. Prove that the equivalence problem of regular expressions, that is the question, "$\alpha \sim \beta$?", for any two α and β over some Σ, is decidable (algorithmically solvable).

Hint. Start by thinking set-theoretically.

37. Prove that *universal FA do not exist*. Let us make the preceding statement precise.

We can easily see that *all* possible FA with tape alphabet $\{0, 1\}$ can be coded as strings over a fixed alphabet to be specified below:

Indeed, let us code a FA M: Each instruction $\delta(q_i, a) = q_j$ is represented by the string $q\tilde{i} * a * q\tilde{j}$, where $a \in \{0, 1\}$ and $\tilde{i}$ is the decimal representation of the number i. Thus, adding ";" to the alphabet as a *new* symbol, we represent the automaton by "gluing" the instruction-representations, one after the other, using ";" as inter-instruction glue, and appending at the end of the sequence the string "; $q\tilde{m}; q\tilde{n}; \ldots; q\tilde{k};$" which indicates that q_m is the *initial* state and $q_n, \ldots, q_k$ are the *accepting* states.

Any automaton, such as M, has more than one string representation (due to the fact that permutations of states and/or instructions are possible) over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, q, *, ; \}$.

$R(M)$ will denote *all* the representations of M. End of the description of how to code a FA.

Thus, you are asked to prove that the language

$$L = \big\{ x; y : (\exists M)\big(M \text{ is a FA } and \ x \in R(M) \wedge y \in L(M)\big)\big\}$$

is *not* regular, or, in plain English, "there is *no* FA U that can faithfully simulate an *arbitrary* FA M (the latter coded as x) on *arbitrary* input y over $\{0, 1\}$".

CHAPTER 4

ADDING A STACK TO A NFA: PUSHDOWN AUTOMATA

FA, and therefore NFA, accept some reasonably nontrivial languages over some alphabet Σ, yet they fail decisively on certain simple languages, such as $L = \{0^n 1^n : n \geq 0\}$ over $\Sigma = \{0, 1\}$, as we showed in 3.1.3.2. This language is an abstraction of a special case of "the set of strings of balanced brackets",[125] that is, $L_B = \{(^n)^n : n \geq 0\}$ over $A = \{(,)\}$. Balanced brackets play an important role in the definition of syntax, and subsequently in the syntactic analysis of formulae, and of programming language constructs. In the latter domain, the result of 3.1.3.2, for example, makes it impossible for an automaton to *recognize* and *pass* string constants to the language translator of Algol 60. The reason is that such strings have a balanced bracket structure—where the brackets in this context are left and right quotes, " and ". This does not happen with other major programming languages that do not define strings this way.

In this chapter we will add slightly to the power of the NFA, without going all the way back to the full-fledged URM, so that languages such as L and L_B are acceptable.

---

[125] This special case includes strings such as $((()))$ but not $((()))(())$.

4.0.0.4 Remark. Translating the notation of 1.6.0.10 to grammar notation, we see at once that $L = L(G)$, where G is the CFG over $\Sigma = \{0, 1\}$ with just two rules:

$$S \to \epsilon \,\Big|\, 0S1$$

That is, this language is context free (type-2). □

We will see that the augmented NFA of this chapter accept precisely the CFL.

4.1 THE PDA

Computer science students almost certainly become familiar with the concept of a *stack* by the time they enroll in a course on the (meta)theory of computation. Let us think of a stack as a *string* γ, over some alphabet Γ, whose length we are allowed to vary by adding to it or deleting from it *exactly one symbol* at a time. We have some *restrictions to our access* to the stack γ:

(1) Adding takes place *only* at the *left* end of γ. This end is called the *top of the stack*. Thus, if $A \in \Gamma$, we can perform a *Push* operation and go from γ to $A\gamma$. We say that we *pushed A into the stack*. One often writes $\gamma \downarrow A$ to indicate that we pushed A into γ.

(2) The reverse operation, deleting, is called a *Pop* operation and it also takes place *only* at the *left* end of γ. It is defined only if $\gamma \neq \epsilon$. Say then $\gamma = B\gamma'$. Then a pop operation on this γ yields the string γ'. One often writes $\gamma \uparrow$ to indicate that we popped the top of γ (assuming we knew $\gamma \neq \epsilon$).

(3) *Read access* is allowed *only* at the top of γ.

(1) We endanger no confusion between the notations $\gamma \downarrow$ and $f(a) \downarrow$ as one is about a string (stack) and the other about a function (cf. p. 43). One often encounters the notations $\gamma \Leftarrow$ and $\Leftarrow \gamma$ for $\gamma \downarrow$ and $\gamma \uparrow$, respectively. The danger of confusing this symbol is greater in this context, where we are using its long form, $\Longrightarrow$, for the yields-relation in (grammar) derivations.

(2) A so-augmented NFA in essence has the ability—additionally to the NFA's ability to read input, one symbol at a time, as well as the ability to make an ϵ-move *without reading input*—to record an *unbounded*, that is, *dependent on input size*, amount of information γ in a *stack variable*. In the preamble of Chapter 3 we introduced the FA as a restricted URM with only one variable that can hold a single digit at a time. The PDA model of this section will additionally have another variable that can hold any string (equivalently, any natural number, just as the variables of the unrestricted URM do), but we will put restrictions on the modes of *access* to this string-variable. In what follows we will not refer to this variable explicitly any more than we do so for the "input variable" of an NFA.

Are we deviating from our "program" to introduce only *subsidiary* formalisms to that of the URM? What of "string variables" and, especially, "restricted access" to them? Notwithstanding the comment that we often make that strings and natural numbers are interchangeable objects since the latter are coded as strings anyway (e.g., decimal notation) and, conversely, they can code strings, can we claim that a URM can perform those string-specific operations, such as "push" and "pop"? Yes. However, we do not want to digress on this issue here. See Chapter 5.

4.2 PDA COMPUTATIONS

4.2.0.5 Definition. (PDA) A *pushdown automaton* or *PDA*, M, is a NFA equipped with a stack variable whose contents we will generically denote by lowercase Greek letters from the beginning of the alphabet (α, β, γ) with or without primes.

Algebraically speaking, M is a toolbox
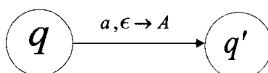
$$M = \langle \Sigma, \Gamma, Q, \delta, q_0, F \rangle$$

where the finite set Σ is the input alphabet; Q is a finite set of states; δ is the "program", that is, the transition *relation*; q_0 (generic name!) is a distinguished member of Q, the start state; F is a finite set of accepting states.
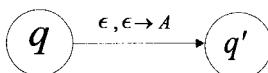
Γ is new: It is a finite alphabet of stack symbols. The stack variable takes values from Γ^*. It *is* allowed to have $\Sigma \cap \Gamma \neq \emptyset$.

The set of instructions (program) is encoded into the relation δ. A PDA has only *four* types of possible instructions, given below in flow-diagram form:
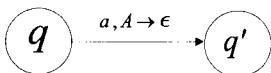
Semantics
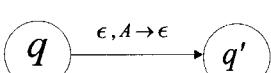
q: Read a, Push A; goto q'

q: Push A; goto q'

q: Read a, Pop A; goto q'

q: Pop A; goto q'

where, generically, an "A" denotes a member of Γ and an "a" denotes a member of the input alphabet Σ.[126] The first and third are "mixed" instructions (both input and stack activity), while the second and fourth are "pure" (stack activity only). The second is a "pure push instruction".

Mixed instructions "consume input" while pure instructions do *not*. For the latter note the analogy with a NFA's ϵ-moves.

Note that instructions of type 3 and 4 are *constrained* by what is the leftmost ("top") symbol of the stack: A *must* be the symbol at the top of the stack; *otherwise the instruction is not applicable.*   □

Very important. Jargon such as "consume input", and "push" into or "pop" from a stack, help our intuition and guide us toward the next definition, which details how PDAs compute. Such terminology has no formal status nor is it needed, except to serve pedagogy. At the end of the day, a PDA, just like a NFA, is a *string transformation formalism*—we are given a set of rules and a methodology on how to *construct certain sequences of strings.* Such formalisms allow us to "correctly" write down finite sequences such as

$$I \vdash_M I' \vdash_M I'' \vdash_M I''' \vdash_M \cdots$$

which we have seen before (FA and NFA). We also met such formalisms under a different guise—that of grammars— which led to a mechanism that allows us to write down *derivations* $\alpha \implies \alpha' \implies \alpha'' \implies \cdots$. Above all, we have met such string transformation formalisms way before all this: mathematical proofs (cf. 1.1.1.34).

Thus the above "semantics" embodies only *stated intentions and guidelines* no more. Once we define "computations" the above semantics will become formal.
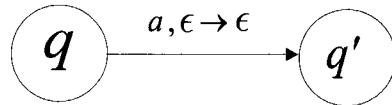
4.2.0.6 Example. We often allow, for the sake of convenience, the following *derived* (simulated) types of moves that were not among the primitive (primary) moves of Definition 4.2.0.5. Let us adopt in what follows that $a \in \Sigma \cup \{\epsilon\}$.
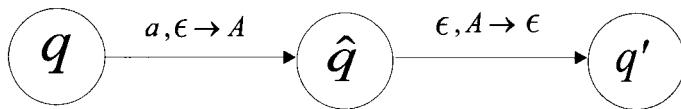
The first is an "ignore-the-stack" move (and it will also ignore the input, if $a = \epsilon$). In the simulation, A is a *new* stack symbol and $\widehat{q}$ a *new* state. This means that if I "program" a PDA, M, with a "macro" like $a, \epsilon \to \epsilon$, then the expansion (implementation) of the macro is done by *adding* a *new* state $\widehat{q}$ and a *new* stack symbol A to the respective alphabets (Q and Γ) of my original M and by using the

---

[126]If instead we opted for the understanding that $a \in \Sigma \cup \{\epsilon\}$, then we could conflate instructions one and two, and three and four.

three-state sequence below to implement the derived move.
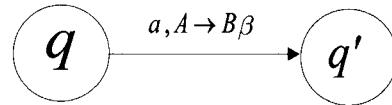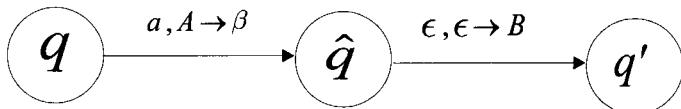
The next is also very useful when programming PDAs, e.g., when constructing examples. The suggested implementation takes care of the general case in a compact (recursive) manner. It says that "if you know how to macro expand (simulate) '$a, A \to \beta$', then here is how to do '$a, A \to B\beta$' ". The "basis" of the recursive construction is when $\beta = \epsilon$, which is covered by one of our basic moves (Definition 4.2.0.5).

Note that $a, A \to \beta$ means to read input a (if $a \neq \epsilon$) and also to pop A, *provided it is the top-of-stack symbol*, and then to push all of β into the stack—*without consuming any more input*. More precisely, if γ, the stack contents, is the string $A\gamma'$ before the effect of the instruction, it will change to $\beta\gamma'$ after the instruction is performed.
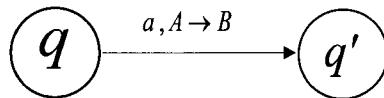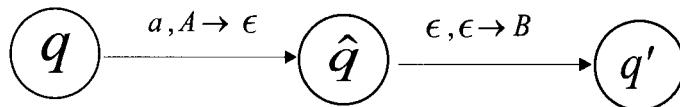
4.2.0.7 Example. Specifically, we simulate "$a, A \to B$" as follows

Simulated by

$\square$

There are a number of concepts we need towards defining a *computation* of a pushdown automaton, and eventually string acceptance.

4.2.0.8 Definition. (Configurations or IDs) A *configuration* is a snapshot in "time", also called an *instantaneous description*, for short, *ID*, of a PDA *computation*.[2] It is a *triple* (q, w, γ), where q denotes the current state, w is the *unspent* ("unexpended" or unprocessed, or, unread-so-far) input, and γ is the total contents of the stack, read from top to bottom (recall that the leftmost symbol of γ is the topmost symbol of the stack).

From the description of w follows that the *next input* that will be read, *if* we perform an instruction of types 1 or 2 (4.2.0.5) on this configuration, will be the leftmost symbol of w. $\square$

4.2.0.9 Definition. (Moves) If I and J are configurations of a PDA M, then $I \vdash_M J$ or simply $I \vdash J$ if M is understood from the context—pronounced "I yields J"— means that *there is a move*[127] of M that transforms the ID I into J, *in one step*. The foregoing is "English" for the following mathematically precise specification by cases:

For all $a \in \Sigma \cup \{\epsilon\}$, $y \in \Sigma^*$ and $\gamma \in \Gamma^*$,

$$(q, ay, A\gamma) \vdash (q', y, \gamma)$$

iff instruction 3 (Def. 4.2.0.5, case $a \in \Sigma$), or 4 (Def. 4.2.0.5, case $a = \epsilon$) is available.

---

[2]The reader should not overlook the *fact* that we do *not* need to *know* what a "computation" is before we know what an "ID" is! Indeed, we will define a computation as an appropriate sequence of IDs.

[127]Note the *there is*. I am *not* saying that it is the *only* move, or that I uniquely determines J.

For all $a \in \Sigma \cup \{\epsilon\}$, $y \in \Sigma^*$ and $\gamma \in \Gamma^*$,

$$(q, ay, \gamma) \vdash (q', y, A\gamma)$$

iff instruction 1 (Def. 4.2.0.5, case $a \in \Sigma$), or 2 (Def. 4.2.0.5, case $a = \epsilon$) is available.

$\square$

4.2.0.10 Definition. (Initial, Terminal IDs) A configuration is *initial* iff it has the form (q_0, x, ϵ). This captures the intended semantics that computations with *input* x start at state "q_0" (generic initial state) and with an empty stack.

A configuration (q, ϵ, γ) is *terminal* or *final* (I did *not* say "accepting!") iff there is *no* defined next move. That is,

$$\neg(\exists J)\Big((q, \epsilon, \gamma) \vdash J\Big) \qquad \qquad \square$$

4.2.0.11 Definition. (PDA Computations) A *PDA computation* is a sequence of IDs, $I_0, \ldots, I_n$ such that

(1) I_0 is initial

(2) I_n is terminal (final)

(3) For $i = 0, \ldots, n-1$, $I_i \vdash I_{i+1}$.

As usual, we write $I_0 \vdash^* I_n$ (denoting 0 or more occurrences of "$\vdash$") and say "$I_0 \vdash^* I_n$ is a computation", which is a slight abuse of language: We should have said—if we do not want to mention $I_1, \ldots, I_{n-1}$—that "there is a computation with I_0 as initial and I_n as terminal IDs".

In this context it is sometimes best to write more specifically $I_0 \vdash^n I_n$, which is explicit that there were n *steps* (applications of $\vdash$) in the computation, or that the computation has *length* n, as we say. $\square$

4.2.0.12 Remark. Thus, we require our computations to be terminating. This is implicit in PDA models that one sees in the literature. For example, in the approach taken in Hopcroft *et al.* (2007), a stack move of the type $\epsilon \to A$ (push) is *not* allowed. Instead, the only stack moves allowed are of the type $A \to \alpha$ (that is, "the stack must always be consulted *before* moving ahead").

The PDAs in loc. cit., which simulate $\epsilon \to A$ by an instruction of the type $X \to AX$ and accept their input by "empty stack" (see below), end up in a terminating configuration, since no move is permitted without consulting the stack.[128]

4.2.0.13 Definition. (Modes of Acceptance: ES, AS, and ES+AS) The string x *is ES-accepted* [129] by a PDA, M, iff there is a computation $(q_0, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$ for some $q \in Q$.

---

[128]To allow such PDAs to *begin* a computation, the stack is "externally"—i.e., *not* by a PDA instruction—initialized with a special "bottom of the stack 'initial' symbol".

[129]By Empty Stack.

For ES-acceptance, the set F of accepting states is irrelevant, and is often taken to be $\emptyset$.

The string x is *AS-accepted* [130] by a PDA, M, iff there is a computation $(q_0, x, \epsilon) \vdash^*$ (q, ϵ, γ) for some $q \in F$. The stack contents, γ, at computation's end are irrelevant to AS-acceptance.

The string x is *ES+AS-accepted* [131] by a PDA, M, iff there is a computation $(q_0, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$ for some $q \in F$.

If the context helps, we may simply say M accepts x, without having to say in what mode (ES, AS, or ES+AS). □

4.2.0.14 Definition. (Acceptance of Languages) If M is a PDA, then $L(M)$ denotes the set $\{x \in \Sigma^* : x \text{ is accepted by } M\}$ (by ES, AS, or ES+AS acceptance, as the case may be).

$$L \text{ is ES- (AS-, or ES+AS-) } accepted$$

iff there is a PDA M that accepts by ES (respectively, AS or ES+AS) such that $L = L(M)$. □

4.2.1 ES vs AS vs ES+AS

4.2.1.1 Theorem. (AS Can Simulate ES) *If M is a PDA accepting by ES, then we can construct a PDA N that accepts by AS, so that $L(M) = L(N)$.*

Proof. We refer to the figure below. Let

$$M = (\Sigma, \Gamma, Q, \delta, q_0, F)$$

where $F = \emptyset$, be our ES-PDA. We build N as it is (partially) suggested in the figure below.

$$N = (\Sigma, \overline{\Gamma}, \overline{Q}, \overline{\delta}, \overline{q_0}, \overline{F})$$

where $\overline{Q} = Q \cup \{\overline{q_0}, \overline{q}\}$, $\overline{F} = \{\overline{q}\}$, and $\overline{\Gamma} = \Gamma \cup \{\$\}$. The symbols $\overline{q_0}, \overline{q}$ and $ are new and $\overline{\delta}$ is δ, augmented by the *new* instructions pictured below. The one involving $\overline{q_0}$ is self-explanatory. The "pure pop" instructions that lead to $\overline{q}$

$$\text{are only defined on states } q \text{ that have no pure push moves} \tag{1}$$
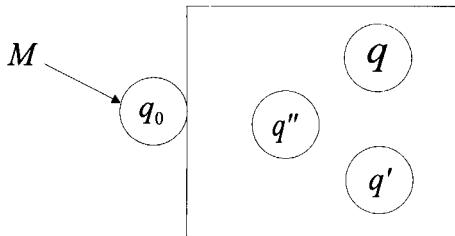
Such states exist, unless M has no terminating computations, in which case $L(M) = \emptyset$ and the result is trivial.

---

[130]By Accepting State.
[131]By Empty Stack *and* Accepting State. We may also say "AS+ES-accepted".
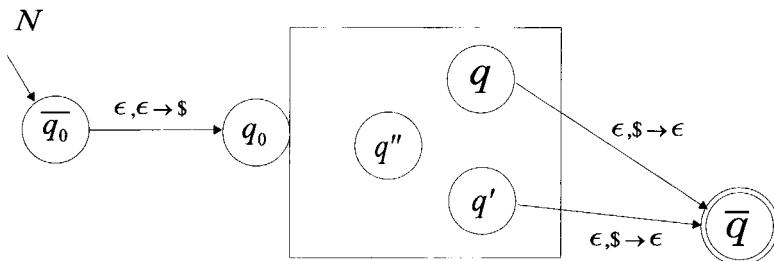
Pause. Why "trivial"?◄

M accepts by ES

N accepts by AS

We now proceed to prove that

$$L(M) = L(N) \tag{2}$$

For $\subseteq$:   Let $x \in L(M)$. Then we have an M-computation

$$(q_0, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$$

Thus,

$$(\overline{q_0}, x, \epsilon) \vdash (q_0, x, \$) \vdash^* (q, \epsilon, \$) \vdash (\overline{q}, \epsilon, \epsilon) \text{ is an } N\text{-computation}$$

Clearly the last ID is terminating, since, by construction of N, $\overline{q}$ has no moves at all.
    Let us justify the last "$\vdash$": Since (q, ϵ, ϵ) is terminal in M, q cannot have pure push moves, else the computation *could* continue from (q, ϵ, ϵ), a fact that would render this ID nonterminal; not possible (cf. Definition 4.2.0.10). Thus, this q *is* connected to $\overline{q}$ as shown in the figure above. For short, $x \in L(N)$.
    For $\supseteq$:   Let $x \in L(N)$. Then

$$(\overline{q_0}, x, \epsilon) \vdash^* (\overline{q}, \epsilon, \gamma) \text{ is an } N\text{-computation} \tag{3}$$

The *only way* to reach $\overline{q}$ is to reach it—in one move—from some q (of the original M) that has no pure push moves (in M) (see figure above). Thus, (3), in some more

detail, is the N-computation below, where the first step is obvious (and inevitable) due to the construction of N:

$$(\overline{q_0}, x, \epsilon) \vdash (q_0, x, \$) \vdash^* (q, \epsilon, \gamma') \vdash (\overline{q}, \epsilon, \gamma) \tag{4}$$

Now, the only way for the last $\vdash$ to be valid (see Definition 4.2.0.9 and the construction of N) is that $\gamma' = \$\gamma$. Moreover, since N can only write \$ once, *in the very first move* (and the M-part cannot write \$ at all), we conclude that $\gamma = \epsilon$.

Now we see (4) more clearly:

$$(\overline{q_0}, x, \epsilon) \vdash (q_0, x, \$) \vdash^* (q, \epsilon, \$) \vdash (\overline{q}, \epsilon, \epsilon)$$

Thus, forgetting the first and last moves, we obtain $(q_0, x, \$) \vdash^* (q, \epsilon, \$)$ and hence[132]

$$(q_0, x, \epsilon) \vdash^* (q, \epsilon, \epsilon) \text{ is an } M\text{-computation} \tag{5}$$

Note that the ID (q, ϵ, ϵ) is terminal in M. Thus, since M is an ES machine, $x \in L(M)$. $\square$

4.2.1.2 Theorem. (ES Can Simulate AS) *If M is a PDA accepting by AS, then we can construct a PDA N that accepts by ES, so that $L(M) = L(N)$.*

Proof. We refer to the figure below. Let

$$M = (\Sigma, \Gamma, Q, \delta, q_0, F)$$

be our AS-PDA. Without loss of generality we assume that $F = \{q\}$ and that q has no moves. Indeed, if it is not so designed, then we modify the original as follows:

(i) we add a new a state q and we designate it as accepting.

(ii) for each (original) accepting state q', we add a move "$\epsilon, \epsilon \to \epsilon$" (recall our macros in 4.2.0.6) from q' to q.

(iii) we give *no moves* to q.
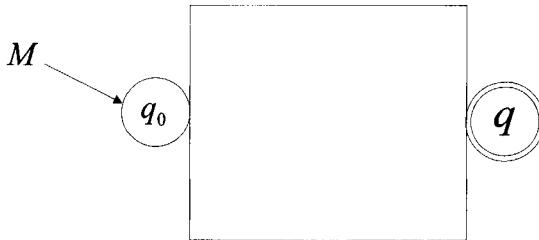
We build N as it is (partially) suggested in the figure below.

$$N = (\Sigma, \overline{\Gamma}, \overline{Q}, \overline{\delta}, \overline{q_0}, \overline{F})$$
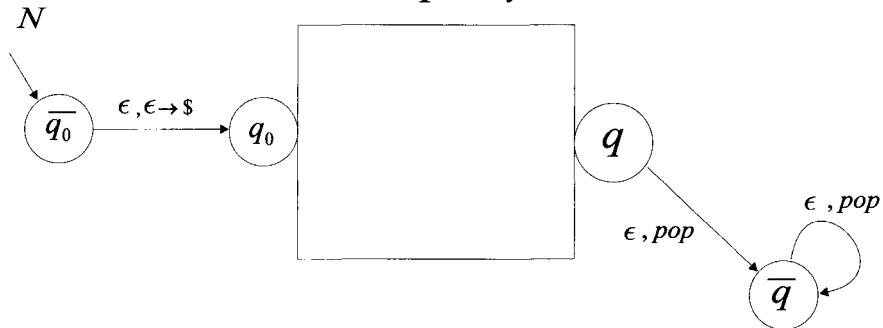
where $\overline{Q} = Q \cup \{\overline{q_0}, \overline{q}\}$, $\overline{F} = \emptyset$ (thus, we have removed accept-status from q), and $\overline{\Gamma} = \Gamma \cup \{\$\}$. The symbols $\overline{q_0}, \overline{q}$ and \$ are new and $\overline{\delta}$ is δ, augmented by the *new* instructions pictured below.

---

[132]In $(q_0, x, \$) \vdash^* (q, \epsilon, \$)$, that is, in $(q_0, x, \$) \vdash I_1 \vdash I_2 \vdash \cdots \vdash I_r \vdash (q, \epsilon, \$)$ each I_i has \$ at the bottom of its stack. In (5) we left all else the same, but we removed all the \$'s.

M accepts by AS

N accepts by ES

We now prove that $L(M) = L(N)$.

For $\subseteq$:   Let $x \in L(M)$. Then

$$(q_0, x, \epsilon) \vdash^* (q, \epsilon, \gamma) \text{ is an } M\text{-computation}$$

Then, trivially,

$$(\overline{q_0}, x, \epsilon) \vdash (q_0, x, \$) \vdash^* (q, \epsilon, \gamma\$) \vdash^* (\overline{q}, \epsilon, \epsilon) \text{ is an } N\text{-computation}^{133}$$

Thus, $x \in L(N)$.

For $\supseteq$:   Let $x \in L(N)$. Then

$$(\overline{q_0}, x, \epsilon) \vdash^* (\overline{q}, \epsilon, \epsilon) \text{ is an } N\text{-computation} \tag{1}$$

Pause. Wait a minute! Why is $\overline{q}$ the state in the last ID of (1)? Well, because *no other state* can contemplate an empty stack. The only states that can erase $\$$ are q and $\overline{q}$, *each by a move that leads to $\overline{q}$.*◄

---

[133]Terminating, since $\overline{q}$ has no pure push moves.

As in the proof of the previous theorem, let us work back from the end: $\overline{q}$ consumes no input and is only accessible from state q (see figure). Thus, when q sent the computation to $\overline{q}$ (by popping once), *the input must have been already consumed.* This, coupled with the inevitable *first* move that places a $ in the stack—a symbol *which cannot be erased by "internal" M-moves*—gives a more "detailed picture" of (1), below:

$$(\overline{q_0}, x, \epsilon) \vdash (q_0, x, \$) \vdash^* (q, \epsilon, \gamma\$) \vdash^* (\overline{q}, \epsilon, \epsilon)$$

Now the part $(q_0, x, \$) \vdash^* (q, \epsilon, \gamma\$)$, after we strip it, throughout, of the bottom $ stack symbol becomes the (*terminating*—remember the fact that q has no moves in M) computation

$$(q_0, x, \epsilon) \vdash^* (q, \epsilon, \gamma)$$

Since $q \in F$, it follows that $x \in L(M)$. $\square$

4.2.1.3 Corollary. *All three acceptance modes are equivalent.*

Proof. By Theorems 4.2.1.1 and 4.2.1.2, ES and AS acceptance modes are equivalent. What about ES+AS?

Well, if M is ES, then an N that accepts by ES+AS can be build that simulates M. We achieved this (without drawing attention to the fact, until now) in the proof of Theorem 4.2.1.1.

Conversely, let M accept by ES+AS. We can then build an ES-PDA simulator, N. The construction and proof is exactly as in Theorem 4.2.1.2 since neither the construction nor the proof there made *any assumptions whatsoever* on what precisely was the γ that was left in the stack at the end of the computation. $\square$

4.2.1.4 Example. Here is how to accept $\{0^n 1^n : n \geq 0\}$ by ES.

Indeed, for $n > 0$,

$$(q_0, 0^n 1^n, \epsilon) \vdash^n (q_0, 1^n, 0^n) \vdash (q, 1^{n-1}, 0^{n-1}) \vdash^{n-1} (q, \epsilon, \epsilon)$$

where $\vdash^0$ is equality. Clearly, (q, ϵ, ϵ) is terminal. We also note that $(q_0, \epsilon, \epsilon) \vdash^*$ $(q_0, \epsilon, \epsilon)$, and $(q_0, \epsilon, \epsilon)$ is terminal. Thus ϵ is accepted.

On the other hand, if the input x is "illegal"—i.e., not of the form $0^k 1^k$—then it is *not* accepted. Here is why: *Such an input starts with a 0 or it starts with a 1:*

(1) $x = 0^n 1^n y$, where $y \neq \epsilon$. Then $(q_0, 0^n 1^n y, \epsilon) \vdash^* (q, y, \epsilon)$. But (q, y, ϵ) is terminal, and the input has not been consumed (for acceptance, the middle coordinate of the last ID *must* be ϵ).

(2) $x = 0^m 1^n y$, where $m > n$ and $y = 0z$, for some $z \in \{0, 1\}^*$. Then $(q_0, x, \epsilon) \vdash^*$ $(q, 0z, 0^{m-n})$. We ran out of moves and did not reach (q, ϵ, ϵ)!

(3) $x = 1z$, where $z \in \{0, 1\}^*$, clearly is not acceptable either; it gets the PDA stuck at q_0 before it can make any move at all! $\qquad\square$

4.3 THE PDA-ACCEPTABLE LANGUAGES ARE THE CONTEXT FREE LANGUAGES

We have defined grammars, in particular, CFG in 3.4.0.7. We will see in this section that CFG are an alternative formalism to that of PDA. They define exactly the same languages. First off we note that for any CFG, $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$, we can assume *without loss of generality* that its instructions have two possible forms: $A \to a$, where $a \in \Sigma \cup \{\epsilon\}$, and $A \to X_1 X_2 \cdots X_n$—the right hand side being a string of X_i, each of which is a nonterminal.

Indeed, if G does not already have the desired rule structure, we add *new* nonterminals, $Y^{(a)}$, one for each $a \in \Sigma$. We then *replace* each rule $A \to \alpha$—where α contains at least one nonterminal—with $A \to \alpha'$, where α' is the result of replacing *each* a occurring in α ($a \in \Sigma$) by $Y^{(a)}$. Finally, we add the rules $Y^{(a)} \to a$, for all $a \in \Sigma$.

We will prove that if the so constructed grammar is G', then $L(G) = L(G')$, which justifies the *"without loss of generality"* above. Indeed, let $x = a_1 a_2 \cdots a_k \in L(G)$, where $k = 0$ means that $x = \epsilon$. Thus we have

$$S \Longrightarrow^* a_1 a_2 \cdots a_k \tag{1}$$

in G. Let us replace, *everywhere*, every a_i that occurs in the derivation (1), by its "alias" $Y^{(a_i)}$. This will transform the G-derivation (1) to a G'-derivation

$$S \Longrightarrow^* Y^{(a_1)} Y^{(a_2)} \dots Y^{(a_k)} \tag{2}$$

Utilizing the rules of the type $Y^{(a)} \to a$ that we have placed in G' we obtain

$$Y^{(a_1)} Y^{(a_2)} \dots Y^{(a_k)} \Longrightarrow a_1 Y^{(a_2)} \dots Y^{(a_k)} \Longrightarrow a_1 a_2 Y^{(a_3)} \dots Y^{(a_k)}$$

$$\Longrightarrow^k a_1 a_2 \cdots a_k$$

Along with (2), this says that $x \in L(G')$.

Conversely, let $y \in L(G')$. Thus, we have a G'-derivation

$$S \Longrightarrow^* y \tag{3}$$

We will show that we also have a G-derivation of y. In order to use induction on derivation length, we prove a somewhat more general statement:

$$\text{if } A \Longrightarrow^n y \text{ in } G', \text{ where } y \in \Sigma^* \text{ and } A \in \mathcal{V}, \text{ then } A \Longrightarrow^* y \text{ in } G \qquad (4)$$

For $n = 1$, the G'-derivation in (4) can only be $A \Longrightarrow y$, where $y \in \Sigma \cup \{\epsilon\}$. But this is also a G-derivation, since $A \in \mathcal{V}$. We take as I.H. that the statement (4) holds for all derivation lengths $< n$. Note that the G'-derivation in (4) can be elaborated as:

$$A \Longrightarrow X_1 X_2 \cdots X_m \Longrightarrow^{n-1} u_1 u_2 \cdots u_m = y \qquad (5)$$

where for $i = 1, \ldots, m$, we have

$$X_i \Longrightarrow^{n_i} u_i \qquad (6)$$

and $\sum_{i=1}^m n_i = n - 1$. Thus

$$n_i < n, \text{ for } i = 1, \ldots, m \qquad (7)$$

Every X_i in (6) that is in $\mathcal{V}$ obeys the I.H. due to (7), thus

$$\text{for all such } X_i, \text{ (6) implies } X_i \Longrightarrow^* u_i \text{ in } G \qquad (8)$$

On the other hand, any X_j in (5) that is *not* in $\mathcal{V}$ is a Y^{a_j}. We replace each such X_j in (5) by a_j. *Thus the corresponding u_j is a_j.*

Let us call $X'_1 \ldots X'_m$ the so modified $X_1 \ldots X_m$. By the construction of G', $A \to X'_1 \ldots X'_m$ is a rule of G. This observation and (8) transform (5) into a G-derivation

$$A \Longrightarrow X'_1 X'_2 \cdots X'_m \Longrightarrow^* u_1 u_2 \cdots u_m = y$$

We can now apply what we just proved to (3).

We these preliminaries out of the way, we can now turn to the equivalence of the CFG and PDA formalisms.

4.3.0.5 Theorem. *For any CFG $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$ we can construct a PDA $M = \langle Q, \Sigma, \Gamma, q_0, \delta, F \rangle$ that accepts by ES such that $L(G) = L(M)$.*

Proof. Given the discussion preceding the theorem, without loss of generality, the rules of G are of either the type $A \to X_1 X_2 \cdots X_m$ or $A \to a$, where capital Latin letters denote, as usual, nonterminals and $a \in \Sigma \cup \{\epsilon\}$. We have drawn below the

PDA that we claim will work. *We note that in the diagram below $b \in \Sigma$.*

The idea of how M "parses" an input string x is already embodied in the preamble of Section 3.4. The stack is used to, well, stack "to do" items on top of each other in the proper sequence. The first "to do", given the input x, is to attempt to show that S—the start symbol—"covers" x, in the jargon used in the discussion of said preamble, or, technically, derives it. Now, if $S \rightarrow ABC$ is the right rule to use (nondeterminism!) next, we replace and refine the task for S by the subtasks that A, B, and C will have to cover appropriate parts of x: a prefix, a middle part, and the remaining tail. Note that A, appropriately, will be on top of the stack. Note that these tasks do not allow us to consume input. Input is consumed once a "b" (from Σ) is found on top of the stack; it is something we uncovered, and it better match the input symbol scanned at that point!

Technically, we will prove that, for any $z \in \Sigma^*$, $\gamma \in \Gamma^*$, $A \in \mathcal{V}$, and $x \in \Sigma^*$, we have

$$(p, xz, A\gamma) \vdash^* (p, z, \gamma) \text{ iff } A \Longrightarrow^* x \tag{1}$$

For the ($\rightarrow$) in (1) we do induction on n of

$$(p, xz, A\gamma) \vdash^n (p, z, \gamma) \tag{2}$$

For the basis, let $n = 1$. The only way for (2) to happen in one move is if the move involved is $\epsilon, A \rightarrow x$ where $x = \epsilon$. But then $A \Longrightarrow x$, as needed. Assume the claim for all values less than n. For $n > 1$ (the $n = 1$ case was done) we have two cases, one that does not need the I.H. and one that does.

- $n = 2$: The first move is "$\epsilon, A \rightarrow a$", where $x = a \in \Sigma$. Then the corresponding IDs are connected as follows: $(p, az, A\gamma) \vdash (p, az, a\gamma) \vdash (p, z, \gamma)$. The first move requires $A \rightarrow a$ to be a rule of G, hence we have $A \Longrightarrow a$, as needed.

- $n > 2$: We elaborate on (2) above: $(p, xz, A\gamma) \vdash (p, xz, X_1 X_2 \cdots X_m \gamma) \vdash^{n-1} (p, z, \gamma)$. We further elaborate on "$\vdash^{n-1}$": There is a decomposition of x as

$x = u_1 u_2 \ldots u_m$, such that

$$(p, u_1 \ldots u_m z, A\gamma) \vdash (p, u_1 \ldots u_m z, X_1 X_2 \cdots X_m \gamma) \vdash^{k_1}$$
$$(p, u_2 \ldots u_m z, X_2 \cdots X_m \gamma) \vdash^{k_2} (p, u_3 \ldots u_m z, X_3 \cdots X_m \gamma) \vdash^{k_3}$$
$$(p, u_4 \ldots u_m z, X_4 \cdots X_m \gamma) \vdash^{k_4} \ldots \vdash^{k_{m-1}}$$
$$(p, u_m z, X_m \gamma) \vdash^{k_m} (p, z, \gamma)$$

For each i it takes $k_i < n$ steps (cf. 4.2.0.11) to lose X_i from the stack top and to consume the portion u_i of the input, that is, we have

$$(p, u_i w, X_i \gamma') \vdash^{k_i} (p, w, \gamma') \tag{3}$$

By the I.H., we must conclude that

$$X_i \Longrightarrow^* u_i \tag{3'}$$

Coupling $(3')$ with the fact the first $\vdash$ in the second bullet above requires that $A \to X_1 \cdots X_m$ is a G-rule, we obtain $A \Longrightarrow X_1 \cdots X_m \Longrightarrow^* u_1 \ldots u_m = x$, as needed.

For the $(\leftarrow)$ in (1) we do induction on n of

$$A \Longrightarrow^n x \tag{4}$$

For $n = 1$ we have two cases:

- $x = \epsilon$. Then we have $A \to \epsilon$ is a rule, thus we have $(p, xz, A\gamma) \vdash (p, z, \gamma)$ using the move pictured at the top of the loop of the PDA (p. 307).

- $x = a$. Then we have that $A \to a$ is a rule, thus we have $(p, az, A\gamma) \vdash (p, az, a\gamma) \vdash (p, z, \gamma)$, using the moves pictured at the top and top right of the loop of the PDA.

For $n > 1$ we take the obvious I.H. and elaborate on (4):

$$A \Longrightarrow X_1 X_2 \cdots X_r \Longrightarrow^{n-1} u_1 u_2 \ldots u_m = x \tag{5}$$

where each X_j is responsible for the substring u_j of x. More precisely,

$$\text{for } i = 1, \ldots, r, X_i \Longrightarrow^{<n} u_i \tag{6}$$

where $\Longrightarrow^{<n}$ means $\Longrightarrow^k$, for some $k < n$. The I.H. is applicable to (6), thus, from (5) we get

$$(p, u_1 \ldots u_r z, A\gamma) \vdash (p, u_1 \ldots u_r z, X_1 \cdots X_r \gamma) \overset{(6)+I.H.}{\vdash^*}$$

$$(p, u_2 \ldots u_r z, X_2 \cdots X_r \gamma) \overset{(6)+I.H.}{\vdash^*} (p, u_3 \ldots u_r z, X_3 \cdots X_r \gamma) \overset{(6)+I.H.}{\vdash^*} \ldots \overset{(6)+I.H.}{\vdash^*}$$

$$(p, u_r z, X_r \gamma) \overset{(6)+I.H.}{\vdash^*} (p, z, \gamma)$$

With (1) proved, let $S \Longrightarrow^* x$. Then

$$(q, x, \epsilon) \vdash (p, x, S) \overset{\text{by (1)}}{\vdash^*} (p, \epsilon, \epsilon)$$

Note that (p, ϵ, ϵ) is terminal as all moves from p require a non-empty stack. Next, let $(q, x, \epsilon) \vdash^* (p, \epsilon, \epsilon)$. The only move for q is $(q, x, \epsilon) \vdash (p, x, S)$, thus our assumption implies $(p, x, S) \vdash^* (p, \epsilon, \epsilon)$. By (1), $S \Longrightarrow^* x$ holds. $\qquad\square$

4.3.0.6 Theorem. *For any PDA $M = \langle Q, \Sigma, \Gamma, q_0, \delta, F \rangle$ that accepts by $ES + AS$ we can construct a CFG $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$ such that $L(G) = L(M)$.*

Proof. Without loss of generality, $F = \{q_{acc}\}$. The nonterminals of G will be ordered pairs of states of M: $\langle q, p \rangle$ for all q, p in Q (think of them as BNF nonterminals!). We aim to design G so that we can prove

$$\langle q, p \rangle \Longrightarrow^* x \in \Sigma^* \text{ iff, for all } \gamma \in \Gamma^* \text{ and all } z, \text{ we have}$$
$$\text{a subcomputation } (q, xz, \gamma) \vdash^* (p, z, \gamma) \text{ that never pops from } \gamma \qquad (1)$$

To achieve this we include the rules

(i) $\langle q, q \rangle \to \epsilon$, for all $q \in Q$

(ii) $\langle q, p \rangle \to \langle q, r \rangle \langle r, p \rangle$, for all q, p, and r in Q

(iii) $\langle q, p \rangle \to a \langle r, s \rangle b$, for all q, p, r, and s in Q and a, b in $\Sigma \cup \{\epsilon\}$, *for all pairs of moves* such as the ones below. The first pushes some $X \in \Gamma$ to the stack on input a and the second, *later*, pops the *same* X, on input b.

(iv) The start symbol, S, is $\langle q_0, q_{acc} \rangle$.

Let us prove (1), splitting it into two directions:

($\to$): Assume $\langle q, p \rangle \Longrightarrow^n x$ and conclude the right hand side of "iff". We use induction on n. Given the kinds of rules that we adopted for G, the only string (over Σ) that can be produced for $n = 1$ is $x = \epsilon$. That means $\langle q, p \rangle \Longrightarrow \epsilon$, $\langle q, p \rangle \to \epsilon$

being the responsible rule. Therefore $p = q$. But $(q, \epsilon z, \gamma) \vdash^* (q, z, \gamma)$ holds since $\vdash^*$ includes equality. Of course, this subcomputation never pops γ.

Let $n > 1$. We assume the claim (1) for all derivation lengths $k < n$ (the I.H.) Now $\langle q, p \rangle \Longrightarrow^n x$ can be elaborated under two cases:

- For some $r \in Q$, we have

$$\langle q, p \rangle \Longrightarrow \langle q, r \rangle \langle r, p \rangle \Longrightarrow^{<n} u \langle r, p \rangle \Longrightarrow^{<n} uv = x \qquad (2)$$

  With reference to (2), the I.H. gives us the subcomputation

$$(q, uvz, \gamma) \vdash^* (r, vz, \gamma) \vdash^* (p, z, \gamma)$$

  where (still under the I.H.) none of the two "$\vdash^*$" pop from γ, hence nor does the entire subcomputation $(q, uvz, \gamma) \vdash^* (p, z, \gamma)$.

- $x = aub$—$\{a, b\} \subseteq \Sigma \cup \{\epsilon\}$—and, for some r and s in Q, we have

$$\langle q, p \rangle \Longrightarrow a \langle r, s \rangle b \Longrightarrow^{<n} aub \qquad (3)$$

  The first $\Longrightarrow$ entails that $\langle q, p \rangle \to a \langle r, s \rangle b$ is a rule in G, and this in turn means that, for some $X \in \Gamma$, we have the two moves in M, depicted in the preceding figure. On the other hand, by the I.H. we have, for any γ, z, and the X mentioned above,

$$(r, ubz, X\gamma) \vdash^* (s, bz, X\gamma)$$

  where the stack $X\gamma$ was never popped. Combining with the moves involving X, we obtain

$$(q, xz, \gamma) = (q, aubz, \gamma) \vdash (r, ubz, X\gamma) \vdash^* (s, bz, X\gamma) \vdash (p, z, \gamma)$$

  Clearly, since $X\gamma$ was never popped in the above, nor was γ.

This completes what we set out to do.

For the $(\leftarrow)$ direction of (1),

we assume that we have $(q, xz, \gamma) \vdash^n (p, z, \gamma)$—*where we never popped γ*   (4)

and prove $\langle q, p \rangle \Longrightarrow^* x$.

For $n = 0$ the assumption is $(q, xz, \gamma) \vdash^0 (p, z, \gamma)$. But $\vdash^0$ is the identity, therefore $q = p$ and $x = \epsilon$. Since $\langle q, q \rangle \to \epsilon$ is a rule, we have $\langle q, q \rangle \Longrightarrow \epsilon(= x)$, as it was needed.

We next fix $n > 0$, adopt the I.H. that the claim is true for all $k < n$, and address the case for n. We have two subcases

- Case where the computation in (4)—while it *never pops γ*—nevertheless "dips down" to (stack contents) γ in *at least one intermediate* instance (that is, it

pops from stack (contents) $Y\gamma$, for some Y. Say this happens for the first time after a prefix u of x has been consumed:

$$(q, xz, \gamma) = (q, uvz, \gamma) \vdash^{<n} (r, vz, \gamma) \vdash^{<n} (p, z, \gamma)$$

By the I.H. we have $\langle q, r \rangle \Longrightarrow^* u$ and $\langle r, p \rangle \Longrightarrow^* v$. Since $\langle q, p \rangle \to \langle q, r \rangle \langle r, p \rangle$ is a rule and thus $\langle q, p \rangle \Longrightarrow \langle q, r \rangle \langle r, p \rangle$, we conclude $\langle q, p \rangle \Longrightarrow^* x$, as was needed.

- Case where the computation in (4)—not only it *never pops* γ—but also never "dips down" to (stack contents) γ in *any* instance except *at the end of the very last move* that leads to ID (p, z, γ). Thus the first move *must* push in the stack some symbol X, and the last move of (4) *must pop that very same X*.

We are proving this *metatheoretical* result for the formal PDA model as this was defined in 4.2.0.5. As such, this type of PDA has only push or pop moves; it does not have "ignore the stack" moves—$\epsilon \to \epsilon$—that we introduced *post facto* as derived moves or *macros* in Example 4.2.0.6. Thus, pop being ruled out in the first move, by the conditions of this case, *push it is*!

Thus, in (4), $x = aub$ for some a and b in $\Sigma \cup \{\epsilon\}$ and we refine (4) as

$$(q, aubz, \gamma) \vdash (r, ubz, X\gamma) \vdash^{<n} (s, bz, X\gamma) \vdash (p, z, \gamma)$$

By I.H. we have $\langle r, s \rangle \Longrightarrow^* u$. Since the first and last move put the rule $\langle q, p \rangle \to a \langle r, s \rangle b$ in G, we have $\langle q, p \rangle \Longrightarrow^* x$ and we are done with the proof of (1).

With (1) settled, we have in particular

$$(q_0, x, \epsilon) \vdash^* (q_{acc}, \epsilon, \epsilon) \text{ iff } \langle q_0, q_{acc} \rangle \Longrightarrow^* x$$

and this concludes the proof that $L(M) = L(G)$. $\qquad\square$

The convenience stemming from Theorems 4.3.0.6 and 4.3.0.5 is significant as they allow us two distinct tools to use toward proving properties of PDA-acceptable languages: PDA and CFG.

4.3.0.7 Example. If $L = L(M)$ for some PDA M, then there is a PDA N such that $L^* = L(N)$. That is, "PDA-acceptable languages are closed under Kleene star".

We prove so (in outline) for CFLs instead!

Let then G be a CFG such that $L = L(G)$. We built—rather than building N—a CFG G' such that $L^* = L(G')$. In going from G to G' we just add a new nonterminal Z and make it the (new) start symbol. If S is the start symbol of G, we add the rules

$$Z \to \epsilon \,\Big|\, SZ$$

to G to finalize the construction of G'.

The reader can easily show that $Z \Longrightarrow^* S^n$, for all $n \geq 0$, from which the result follows. The details are left to the reader. $\square$

4.3.0.8 Example. Are CFL closed under intersection? It turns out that they are not. Consider $L = \{0^n 1^n 2^k : n \geq 0 \wedge k \geq 0\}$ over $\{0, 1, 2\}$. Also consider $L' = \{0^n 1^k 2^k : n \geq 0 \wedge k \geq 0\}$.

It is easily seen that they are CFL.

For L either give explicit rules, or, alternatively, note that it is the concatenation of two context free languages, L_1 and L_2 over $\{0, 1, 2\}$, and invoke Exercise 4.5.11.

$$L_1 = \{0^n 1^n : n \geq 0\}$$

and

$$L_2 = \{2^k : k \geq 0\}$$

(L_2 is actually regular, being $L(2^*)$)

One similarly sees that L' is a CFL. However, $L \cap L' = \{0^n 1^n 2^n : n \geq 0\}$ and we will see in the next section that this language is not context free. $\square$

4.4   NON CONTEXT FREE LANGUAGES; ANOTHER PUMPING LEMMA

In this section we prove a pumping lemma for CFL that is similar, but as expected from the richer structure of the PDA, more complex than the one for regular languages. We will benefit in our proof from the concept of *parse tree* for a CFG. The reader has most likely encountered *trees* in courses on discrete mathematics, data structures, etc. A tree is a structure like the one drawn below:

Since by default all edges point "downwards", one does not need to emphasize so by drawing them as arrows. The round nodes may or may not have names. Our parse trees will have all their nodes named. The precise angle or slope of the edges does not matter—meaning, we allow variation in drawings without changing the tree we want to depict. However, *left to right order matters*, thus the following is a different

tree from the one above. Our trees are *ordered*.

Of course, the reader has encountered many aspects of discrete mathematics before, but this did not stop me from reintroducing some such concepts in Chapter 1. So, let us revisit the definition of trees. A tree has both a "data" component (the node "contents", whatever they may be) and a *structural component* or geometry, that is, how the nodes are interconnected. This motivates us to opt for an abstraction toward a mathematical definition that wants a tree to be an ordered pair of a "data" part and a "structure" part, $\langle S, T \rangle$, where S is a set of objects and T is the "geometry" imposed upon them. To get quickly to our pumping lemma we present only as much dendrology[134] as necessary, thus we will rather define directly those trees that matter to us: *labeled trees* that are *parse trees* of some CFG G.

4.4.0.9 Definition. (Parse Trees) Given a CFG $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$. The set of all *parse trees* for G, and the associated concepts of *root*, *support*, and *yield*, are defined inductively.

> In what follows we assume an infinite supply of (unspecified) *objects*, a set N of *nodes*. We may visualize N as an infinite supply of "circles", such as the ones employed in the two illustrations above.

(1) *Basis—"smallest" parse trees.* $\mathcal{T} = n$ is a parse tree, if $n \in N$. This n is *labeled* by either some $A \in \mathcal{V}$ or some $a \in \Sigma \cup \{\epsilon\}$.

> We say that n is the *root* of $\mathcal{T}$, $sp(\mathcal{T}) = \{n\}$ is its *support*, and the string A (respectively a) is the *yield* of the tree.

> *Note that we use the* labels *in the definition of yield, but use the* node name *in the definition of root!*

> As the recursive definition unfolds, we will note at once that the support of a tree is simply the set of all nodes from N that we utilized to build it.

(2) Suppose that $\mathcal{T}_i$ are *all* parse trees, for $i = 1, \ldots, k$, where for each $i \neq j$ we have $sp(\mathcal{T}_i) \cap sp(\mathcal{T}_j) = \emptyset$[135]—i.e., the trees do not "share nodes".

---

[134] The study of (in our case, mathematical) trees.
[135] We say that the $sp(\mathcal{T}_j)$ are *pairwise disjoint*.

They *may* share labels though!

Assume further that $\mathcal{T}_i$ has as root r_i, and yield $\alpha_i \in (\mathcal{V} \cup \Sigma)^*$.

Let $r \in N$ be *new* relative to the $\mathcal{T}_i$ [i.e., $r \notin \bigcup_{i=1}^{k} sp(\mathcal{T}_i)$]. Then we can build a new parse tree $\mathcal{T} = \langle r, T_1, \dots, T_k \rangle$ with $sp(\mathcal{T}) = \{r\} \cup \bigcup_{i=1}^{k} sp(\mathcal{T}_i)$—this updates sp recursively and verifies the claim that we made about it above.

Building $\mathcal{T}$ has a restriction: if we want to use a $\mathcal{T}_i$ with root r_i that is labeled ϵ, then it *must* be that this is the only tree we use to build $\mathcal{T}$.

We can label the root r with *any* $A \in \mathcal{V}$, *provided* the rule $A \to \beta$ is in the grammar, where β is formed by concatenating the labels of the r_i in the left to right order $i = 1, 2, 3, \dots, k$.

The yield of the new parse tree is $\alpha_1 \alpha_2 \dots \alpha_k$.

The $\mathcal{T}_i$ are called (the) *subtrees* of r but also subtrees of $\mathcal{T}$.   □

In what follows we will often say *tree*, meaning *parse tree*. Parse trees are the only trees of interest in this volume. The root of any parse tree which has more than one node (in its support) *must* be labeled by a nonterminal, since only construction step (2) of the preceding inductive definition applies.

The following connects Definition 4.4.0.9 with the usual depiction of trees as drawings of nodes connected by edges ("graphs").

4.4.0.10 Informal Definition. We will draw a "physical" tree that corresponds to the inductive Definition 4.4.0.9. Via a recursive construction we associate a *figure* composed of *nodes* (circles) and *edges* (straight lines sloping downwards), the latter connecting pairs of nodes.

(1′) A parse tree of type (1) in Definition 4.4.0.9 has no edges. It is drawn as a single circle, labeled by the root label (written normally close to, or inside the circle).

(2′) Assuming that we have associated a (labeled) drawing with each of the subtrees $\mathcal{T}_i$ occurring in case (2) of Definition 4.4.0.9, we *draw* the parse tree for $\mathcal{T}$ as follows:

   (a) We draw a circle for r and label it A [cf. 4.4.0.9, (2)]

   (b) We introduce *precisely* k edges from the circle representing r (and named A) to each of the roots of the subtrees $\mathcal{T}_i$, which are drawn below r and *from left to right* in the order $i = 1, 2, 3, \dots, k$.   □

4.4.0.11 Example. Here are some parse trees for the CFG in 4.0.0.4.

All three have roots labeled S. The yields of the trees, from left to right, are S, ϵ, and $00S11$. ☐

4.4.0.12 Example. Often we want to discuss a parse tree without drawing a *specific* one. We draw generic trees by essentially drawing a root and then connecting it to its subtrees, the latter drawn as triangles with or without names (below we have called the subtrees $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$). We gave label A to the root.

☐

The following is easy but fundamental.

4.4.0.13 Proposition. *If $\mathcal{T}$ is a (parse) tree, then the root node is the only one pointed to by* no *edges. Every other node in $\mathcal{T}$—that is, $sp(\mathcal{T})$—is pointed to by precisely one edge.*

Proof. We do induction *over the set of* parse trees (4.4.0.9)—or *induction on trees*. If $\mathcal{T} = n \in N$, then the validity of the claim is immediate, since there is just a root node and nothing else. Let then $\mathcal{T} = \langle r, \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$ and assume the claim (I.H.) for all the $\mathcal{T}_i$. By 4.4.0.9, the root, r, of $\mathcal{T}$ will have no edges "coming in", and the *only edges added* are those from r to the r_i—roots of the $\mathcal{T}_i$ (4.4.0.10). Viewed from within $\mathcal{T}$, each of the r_i will now have precisely one edge pointing to them by I.H.

Moreover, since there have been no other edges added, the I.H. covers the case for all other non root nodes as well.   □

4.4.0.14 Definition. We introduce some standard terminology from graph theory: The number of edges pointing to a node in a tree is called the *indegree* of the node.

We say that a node n in a tree is *a child* of a node m iff there is an edge from m to n. Distinct children of m are called *sibling nodes*. We call m *the parent node* of n.

By 4.4.0.13, every non root node in a tree $\mathcal{T}$ has precisely one parent.

A sequence of nodes $a_1, a_2, \ldots, a_n$ in a tree is a *path* or a *chain* iff, for *all* $i = 1, 2, \ldots, n-1$, we have an edge from a_i to a_{i+1}. We say that this is a path *from* a_1 *to* a_n, and that a_n is a *descendant* of a_1 while a_1 is an *ancestor* of a_n. The path is said to have *length* $n-1$, that is, *one less than the number of its nodes*. This number coincides with the number of edges along the path.

A node is a *leaf* iff it has no children. A non leaf node is called an *internal node*.   □

We note that case (2) in Definition 4.4.0.9 dictates that every internal node *must* be labeled by a nonterminal.

We have the following corollaries of 4.4.0.13:

4.4.0.15 Corollary. *The root of a tree is an ancestor of every non root node of the tree.*

Proof. Induction on trees. For the basis, let $\mathcal{T} = n \in N$. The result is vacuously true.

Let us look at the case of $\mathcal{T} = \langle r, \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$, where we take as I.H. the truth of the claim for all $\mathcal{T}_i$. Let n be an arbitrary non root node in $\mathcal{T}$, and say, without loss of generality, that it is in $\mathcal{T}_1$. By the I.H. there is a path from r_1 (notation as in 4.4.0.9) to n within $\mathcal{T}_1$. The case rests since there is an edge from r to r_1.   □

4.4.0.16 Remark. (1) The path from r to n that establishes the above mentioned ancestry is unique, for if $a_1, \ldots, a_q$ and $b_1, \ldots, b_m$ are two distinct such paths where $a_1 = b_1 = r$ and $a_q = b_m = n$, then the paths have a *maximal* common part at the tail-end—they certainly share n— say, $c_1, \ldots, c_p$ where $c_p = n$ and each c_j is both an a_q and a b_t. Now, by assumption, $c_1 \neq r$. Thus (by maximality), it has indegree equal to ≥ 2. A contradiction.

(2) More generally, for a non root a_1, there is *at most one* path $a_1, \ldots, a_n$ that connects any a_1 to any a_n. Indeed, if there are two, we augment each by the path that connects the root r to a_1 and thus have contradicted what we just established above.

(3) A path $a_1, \ldots, a_n$ cannot contain points in more than one subtree, $\mathcal{T}_i$, of r, for, say, that distinct a_i and a_j are in distinct $\mathcal{T}_q$ and $\mathcal{T}_p$, in that order. But then we have two distinct paths from r to a_n, one via a_i and the other via a_j. These paths differ already in their first edge: One is from r to r_q while the other is from r to r_p.

(4) A *cycle* is a path $a_1, \ldots, a_n$ of *at least two* distinct nodes such that $a_1 = a_n$. *A tree can have no cycles.* By way of contradiction: If the above path is a cycle, then

it cannot contain the root, since there can be no edge pointing to the root. Consider then a path from r to a_n:

$$b_1, \ldots, b_m \qquad (*)$$

where $b_1 = r$ and $b_m = a_n$. Without loss of generality, the node b_{m-1} is not one of the a_i—otherwise, we just walk backwards along the path $(*)$ until we find the *first* b_j that is not an a_i—such a b_j exists (why?)—and use this b_j instead. But then the indegree of a_n (respectively, of b_{j+1}) is at least 2.   □

4.4.0.17 Definition. (Levels) We define *levels* for every node in a tree sequentially (iteratively): The root is assigned level 0. If b is any child of a and a has level i assigned, then b is assigned level $i + 1$.   □

Remark 4.4.0.16 ensures that in the process of assigning levels every node is processed, and hence assigned a level, only once, and therefore levels are uniquely determined.

4.4.0.18 Remark. The length of a path from the root of a parse tree to a leaf equals the *maximum level* possible along this path, for the path *cannot be extended* at the leaf, and hence this is the node with the *the maximum possible level along the path*.   □

4.4.0.19 Definition. (Height) The one-node tree of case (1) in 4.4.0.9 has *height* equal to 0. If the root of the tree $\mathcal{T}$ has subtrees $\mathcal{T}_i$ of heights h_i respectively, then the height of $\mathcal{T}$ is $1 + \max\{h_i : i = 1, \ldots, k\}$ [cf. 4.4.0.9, (2)].   □

4.4.0.20 Proposition. *The height of a tree equals the maximum level observed in the tree.*

The maximum level observed in the tree is the length of the longest path (from the root to a leaf) in the tree (cf. 4.4.0.18). Despite "the", a path that scores maximum length is not unique. For example, in 4.4.0.11, the third tree has *three* longest paths; all have length 2: (a) $S, S, 0$; (b) S, S, S; and (c) $S, S, 1$.

Thus, another way to state the proposition is that the height of a tree equals *the maximum path length among all paths from the root to a leaf*.

Proof. Induction on trees. A one-node tree has height 0 and the maximum observed level in it is 0.

Assume now the claim for the "small" trees $\mathcal{T}_i$, for $i = 1, \ldots, k$, of heights h_i, respectively, for $i = 1, \ldots, k$. Consider $\mathcal{T} = \langle r, \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$ of height h. Let l be the length of the longest path in $\mathcal{T}$. We want to show that $h = l$.

Now the *maximum* among all longest path lengths observed in the $\mathcal{T}_i$—of a path from r_i to a leaf—is one less that l since all r_i are children of r. Since this $l - 1$ equals $\max\{h_i : i = 1, \ldots, k\}$ by the I.H., we have that $l = 1 + \max\{h_i : i = 1, \ldots, k\} = h$.   □

4.4.0.21 Proposition. *Given a CFG G whose rules have right hand sides with length bounded by the number n. Then any parse tree of height h has a yield of length at most n^h.*

Proof. Induction on trees. For a one-node tree the height is 0. The yield has length 0 or 1 according as the label is ϵ or $a \in \Sigma$. That is, at most n^0.

Assume the claim for all subtrees $\mathcal{T}_i$, $i = 1, \ldots, k$ of the root. By assumption on the grammar rules, $k \leq n$. The yields of these subtrees satisfy

$$|\alpha_i| \leq n^{h_i}, \text{ for } i = 1, \ldots, k \tag{1}$$

The yield of the entire tree is $\alpha_1 \alpha_2 \cdots \alpha_k$ and the tree height is $1 + \max(h_i)$. Let, without loss of generality, $h_1 = \max(h_i)$. Notice that

$$|\alpha_1 \alpha_2 \cdots \alpha_k| = \sum_{i=1}^{k} |\alpha_i| \overset{\text{by (1)}}{\leq} \sum_{i=1}^{k} n^{h_i} \leq k(n^{h_1}) \leq n(n^{h_1}) = n^{h_1+1} = n^h \quad \square$$

4.4.0.22 Remark. What if $n = 1$ in 4.4.0.21? The analysis above works, noting $k = 1 = n$. In particular, the result means that such a grammar generates a finite set, indeed, a subset of $\Sigma \cup \{\epsilon\}$. $\quad \square$

4.4.0.23 Corollary. *Given a CFG G whose rules have right hand sides with length bounded by the number n. If the yield α of a parse tree of G has length $> n^C$, then the height of the parse tree is $> C$.*

Proof. A height $h \leq C$ leads to yields of length *at most* n^h, but $n^h \leq n^C$. $\quad \square$

Here is the import of parse trees: they depict derivations in a two-dimensional manner and help to simplify the analysis of derivations.

4.4.0.24 Theorem. *Let a CFG G be given. Then we have a parse tree with root label A and yield α iff we have $A \Longrightarrow^* \alpha$ in G.*

Proof. For the *only if*, let us have a parse tree as described. We proceed by induction on parse trees (cf. 4.4.0.9). The one-node case, combined with the assumption, gives us a tree with just one node labeled A. The yield is A. On the other hand, $A \Longrightarrow^* A$.

By taking the I.H. on subtrees, let our parse tree have subtrees $\mathcal{T}_i$, $i = 1, \ldots, k$, with root labels Ξ_i, $i = 1, \ldots k$, where each Ξ_i is in $\mathcal{V} \cup \Sigma \cup \{\epsilon\}$.[136] Let the respective yields be α_i. By 4.4.0.9,

$$\alpha = \alpha_1 \alpha_2 \ldots \alpha_k \tag{1}$$

By the I.H. we have

$$\Xi_i \Longrightarrow^* \alpha_i, \ i = 1, \ldots, k \tag{2}$$

---

[136]This explains the specific to this proof, and "nonstandard", choice of a non Latin capital letter, to allow this level of "inclusivity" in the notation.

where if $\Xi_i \in \Sigma \cup \{\epsilon\}$, then (2) holds as equality without needing any help from the I.H.

By 4.4.0.9, G contains the rule $A \to \Xi_1 \cdots \Xi_k$, hence, from (1) and (2), we obtain $A \Longrightarrow \Xi_1 \cdots \Xi_k \Longrightarrow^* \alpha$.

For the *if* we do induction on n in the hypothesis $A \Longrightarrow^n \alpha$. For $n = 0$ we have $\alpha = A$. Thus the one-node parse tree with root label A will do. We assume the claim (I.H.) for all $m < n$. Thus we look at

$$A \Longrightarrow \Xi_1 \cdots \Xi_k \Longrightarrow^{<n} \alpha \tag{3}$$

We have from above

$$\Xi_i \Longrightarrow^{<n} \alpha_i, \ i = 1, \ldots, k \tag{4}$$

where $\alpha = \alpha_1 \cdots \alpha_k$. Note that if $k > 1$, then no Ξ_i can be ϵ—we just don't write rules like this: $A \to \epsilon Ba$; we simply write $A \to Ba$ [cf. also the restriction in case (2) of 4.4.0.9].

With this understanding noted, we analyze (4) to see what kinds of parse subtrees we can obtain with root labels Ξ_i: If $k = 1$ and $\Xi_1 = \epsilon$, then we have a tree such as the first one in 4.4.0.11 with root label ϵ—that is, Ξ_1, and hence yield $\epsilon = \alpha_1$; the I.H. was not needed in this subcase. Similarly, if $\Xi_i = a \in \Sigma$, then again, without invoking the I.H., we have at once a two-node parse tree with root label a (this is Ξ_i) and with yield $a = \alpha_i$.

The I.H. implies that, in general, we have k parse trees $\mathcal{T}_i$, with roots labeled Ξ_i and yields α_i, for $i = 1, \ldots, k$. By 4.4.0.9 and the first $\Longrightarrow$ in (3), we select a *new* root node, we label it A, and we connect this root, from left to right, with the roots of the $\mathcal{T}_1, \ldots, \mathcal{T}_k$. The yield of the tree we have just built is $\alpha_1 \ldots \alpha_k$ (4.4.0.9). We are done. $\square$

We now turn to the main result of this section, a pumping lemma for the CFL.

4.4.0.25 Theorem. (The $uvwxy$-Theorem) *For any infinite CFL, L, there is a constant C* (not uniquely determined by L!) *such that if $z \in L$ and $|z| \geq C$ then we can partition z as $z = uvwxy$ so that the following hold*

(1) $uv^i wx^i y \in L$ *for all* $i \geq 0$

(2) $vx \neq \epsilon$

(3) $|vwx| \leq C$

Proof. First, let us fix a CFG $G = \langle \mathcal{V}, \Sigma, S, \mathcal{R} \rangle$ such that $L = L(G)$. Let n represent the maximum length of the right hand side of any rule of G. The assumption that L is infinite implies $n \geq 2$ (4.4.0.22). We will take $C = n^{|\mathcal{V}|+1}$, where by the symbol $|\mathcal{V}|$ we denote the number of nonterminals in $\mathcal{V}$. We next fix a $z \in L$ such that $|z| \geq C$. By the choice of z, we have $S \Longrightarrow^* z$. By Theorem 4.4.0.24, there is a parse tree with root S and yield z.

Of all such parse trees we pick one with the *least number of nodes*. $(*)$

We depict this parse tree—in a stylized manner, cf. 4.4.0.12—in the figure below. We have denoted some of its nodes (one labeled S and two labeled A) by big dots rather than by circles. The node labels are near these dots. Now, by 4.4.0.23, the height h of the tree that we have chosen satisfies $h > |\mathcal{V}|$. Moreover, by 4.4.0.20, there is a path of *longest possible* length *that has length precisely equal to the height of the tree*. We depict this path by the wavy line that goes from S, through the two A-labels, to the unlabeled dot that is part of the yield of the bottom "triangle" with root A. This unlabeled dot is labeled by a terminal, since the entire yield, z, is in Σ^*.

The longest path that we are referring to thus has length greater than $|\mathcal{V}|$, and therefore has more than $|\mathcal{V}| + 1$ nodes on it. Since *only* the dot in the yield is labeled by a terminal, this path has utilized *more* than $|\mathcal{V}|$ nonterminal labels. *Thus a nonterminal label repeats.*

We depict label A as repeating. Moreover, we assume that there is *no* label B, between the bottommost A and the last dot (on the yield) on the wavy path, that also repeats. That is, the pair of the two As is bottommost.

With these preliminaries out of the way, we draw attention to the substrings of z marked on the yield. Notice that if we remove the middle parse tree—of root (labeled) A and yield vAx— and join the first and last tree at A we obtain a parse tree with root S and yield uwy. Thus $S \Longrightarrow^* uwy$ and we have proved (1), case $i = 0$.

Let us next form a chain of triangles (parse trees), $\mathcal{T}_1, \ldots, \mathcal{T}_k$ using a *copy* of the middle triangle $k > 1$ times. For each i, we join $\mathcal{T}_i$ with $\mathcal{T}_{i+1}$, using as connecting point the node labeled A in the yield of the former with the root (also labeled A) of the latter. By 4.4.0.9, *the chain is a parse tree* with yield $v^k Ax^k$. We can now replace the single copy of the middle triangle of the preceding figure by the k-long triangle chain that we have just constructed. The root of the chain is glued to the A-node in the yield of the top triangle. The bottom triangle's root is attached to the bottom A of the chain (the one in the yield of the chain). By 4.4.0.9 we obtained a parse tree with a root labeled S and yield $uv^k wx^k y$. Thus $S \Longrightarrow^* uv^k wx^k y$, and have just proved (1) for $i > 1$. For $i = 1$, of course, we need do nothing.

Regarding (2), if $vx = \epsilon$, then the tree we got for $i = 0$ still has yield z. Since this has at least one node less than the original, we have contradicted our stipulation ($*$). Thus, at least one of v or x is not ϵ.

Finally, the wavy path from the first A, via the second A that ends on the yield of the bottom triangle, has the *longest length* in the parse tree, $\mathcal{T}'$, formed by the last two triangles (root labeled A, yield equal to vwy). If not, then there is a longer path in $\mathcal{T}'$, from the root, labeled A, to the tree's yield. Appending this path to the path in the top triangle that goes from S to A, we get an overall path in the original tree—from S to the yield z—that is longer than the original, and this contradicts our assumptions!

Now, *omitting the first A*, the resulting truncated path has length $l \leq |\mathcal{V}|$— otherwise we have a repeating nonterminal on it;[137] impossible by the way we chose the pair of A. Since $l + 1$ is the longest path length in $\mathcal{T}'$, it equals the height of $\mathcal{T}'$ (4.4.0.20). Thus, $l + 1 \leq |\mathcal{V}| + 1$ and hence the yield, vwx, satisfies $|vwx| \leq n^{l+1} \leq n^{|\mathcal{V}|+1} = C$—that is, (3) is true. $\square$

The condition "For any *infinite* CFL, L, etc." in the statement of the theorem was used to guarantee that a CFG for the grammar cannot have all its rules have right hand sides of length one. The proof is applicable for any CFL L that can be obtained as $L = L(G)$ where the CFG G has rules with right hand sides of length ≥ 2.

4.4.0.26 Example. We show that $L = \{0^n 1^n 2^n : n \geq 0\}$ is not a CFL. If it is, then the $uvwxy$-theorem applies. Let then C be as in the 4.4.0.25, and consider $z = 0^C 1^C 2^C$. This z must pump, i.e., there must be a partition as $z = uvwxy$, so that $vx \neq \epsilon$, and the $uv^i wx^i y$ are in L for all i. Seeing that $|vwx| \leq C$, we have two cases: v contains a 0. Then vwx contains no 2, hence pumping leaves 2^C invariant and the pumped string cannot possibly be in L. On the other hand, if v contains no 0, then the part 0^C remains invariant during pumping, and again the string cannot be in L. $\square$

4.4.0.27 Example. The language $L = \{zz : z \in \Sigma^*\}$ over $\Sigma = \{0, 1\}$ is not a CFL either.

Assume by way of contradiction that it is. So let C be chosen as in the $uvwxy$-theorem. We will show that certain strings of L that are longer than C "cannot pump", contradicting 4.4.0.25 (*we do not expect this contradiction to be demonstrated for all strings of L!*).

"Cannot pump" means that if we do pump them as in (1) of 4.4.0.25, then we end up with strings outside L.

So look at $t = 0^C 1^C 0^C 1^C$. Indeed, if we went along with the italicized hypothesis, then we should be able to write $t = uvwxy$ with $|vwx| \leq C$, and $uv^i wx^i y$ would all be in L, for $i \geq 0$. The latter cannot happen:

---

[137] The last node on the path is a terminal in w. A path of length l has $l + 1$ nodes.

In view of the maximum length of vwx, this string can have three general positions as a substring of t:

(1) It has no overlap with the last $0^C 1^C$. This is inconsistent with the specification of L, since pumping t down ($i = 0$) will change at least one of the leftmost 0^C or the leftmost 1^C but will leave the the last $0^C 1^C$ invariant.

(2) It has no overlap with either the first 0^C or the last 1^C. This is again inconsistent with the specification of L, since pumping t down will change at least one of the leftmost 1^C or the second 0^C but will leave the first 0^C and the last 1^C invariant.

(3) It has no overlap with the first $0^C 1^C$. For one last time, this clashes with the specification of L, since pumping t down will change at least one of the rightmost 0^C or the last 1^C but will leave the first $0^C 1^C$ invariant.

These being the only possible placements of vwx we showed that t "cannot pump", hence L is not a CFL.   □

The following is analogous to 3.4.2.3.

4.4.0.28 Example. The CFL version of the pumping lemma gives us a characterization of infinite (and hence of finite) CFL: Let L be a CFL given by some CFG G. If all the rules of G have right hand side lengths equal to one, then the language is finite, as we have already observed.

Let then this not be the case, and let C be the constant used in the proof of 4.4.0.25. Then if there is a string $z \in L$ such that $C \leq |z| < 2C$, L will be infinite.

This is hardly surprising, as it directly follows from "pumping" the string. However, we have a converse! If L is infinite, then there *will* be a $z' \in L$ such that $C \leq |z'| < 2C$.

So, let L be infinite, and let z—no prime—be in L, such that its length is smallest such that $|z| \geq 2C$. Let $z = uvwxy$ as in 4.4.0.25. We have that $uwy \in L$. By smallest length property of z, we obtain (recall, $vx \neq \epsilon$)

$$|uwy| < 2C \tag{1}$$

Moreover,
$$|uwy| = |z| - |vx| \geq |z| - |vwx| \geq 2C - C = C$$

We set $z' = uwy$ and we are done. The contrapositive of the characterization says that L is finite iff there are no $z \in L$ in the length-range $C \leq \cdots < 2C$.   □

4.5 ADDITIONAL EXERCISES

1. Modify the PDA in 4.2.1.4 to accept $\{0^n 1^n : n \geq 1\}$ by ES.

2. Modify the PDA in 4.2.1.4 to accept $\{0^n 1^n : n \geq 3\}$ by ES.

3. Build a PDA over an input alphabet Σ that accepts strings by ES, and (provably) accepts $\{xx^R : x \in \Sigma^*\}$.

4. Build a PDA for the language $\{0^n1^{5n} : n \geq 0\}$. You are free to choose the mode of acceptance.

5. Repeat the above task, but now do it in this way: First get a CFG for the language, and then construct a PDA for the language that accepts by empty stack (cf. 4.3.0.5).

6. Give a CFG G over $\Sigma = \{0, 1\}$ such that $L(G) = \{xx^R : x \in \Sigma^*\}$.

7. Give a CFG G over $\Sigma = \{(,)\}$ such that $L(G)$ is the full set of balanced brackets—that is, not only those of the form $((()))$ but also those like $((()))(())$.

8. Provide a complete proof for the claims in Example 4.3.0.7.

9. By induction on regular expression length prove: "For every α, there is a CFG, G, such that $L(G) = L(\alpha)$".

10. Prove that CFL are closed under union, that is, if L and L' are CFL, then so is $L \cup L'$.

11. Prove that CFL are closed under concatenation, that is, if L and L' are CFL, then so is LL'.

12. Prove that CFL are closed under reversal, that is, if L is a CFL, then so is L^R.

13. Prove that CFL are *not* closed under complement. That is, if L is a CFL over Σ, then *it is not necessarily the case* that $\overline{L}$, that is, $\Sigma^* - L$ is a CFL.

    Hint. Toward your counterexample begin by thinking set-theoretically.

14. Prove that the language over $\{a, b\}$ given by $L = \{a^nba^{n^2} : n > 0\}$ is *not* a CFL.

15. The language $L = \{1^p : p \text{ is a prime}\}$, over $\Sigma = \{1\}$, is not regular. Is it a CFL? Why?

16. The language $L = \{1^{n^3} : n \geq 0\}$, over $\Sigma = \{1\}$, is not regular. Is it a CFL? Why?

17. Prove that the intersection of a CFL L and a regular language L' is a CFL.

    Hint. Let M be a PDA for L and N a FA for L'. Assume that the PDA accepts *by accepting state*. Now imitate the construction of 3.1.2.2.

18. Prove that the language L of strings over $\{0, 1, 2\}$ that have an equal number of zeros, ones and twos is not a CFL.

    Hint. The pumping lemma will not work directly. However, $0^*1^*2^*$ defines a regular language.

19. Prove that the language over $\{0, 1, \#\}$ given by $\{z\#x\#y : \{x, y, z\} \subseteq \{0, 1\}^* \land z = x + y \text{ in binary}\}$ is not a CFL.

CHAPTER 5

COMPUTATIONAL COMPLEXITY

What do we mean by saying "this problem, $x \in A$, has no algorithmic solution"? And why is it that some problems do not have such solutions? How can we classify (compare) such undecidable problems? These are the fundamental questions of computability theory that we studied in Chapter 2.

Among the problems $x \in A$ that *do* have algorithmic solutions (decidable or solvable problems), why is it that some require enormous computational resources toward obtaining the answer? And how can we classify decidable problems according to their demand on computational resources? This is the domain of *computational complexity*, or just *complexity*, theory. This chapter discusses a few topics in complexity theory.

5.1   ADDING A SECOND STACK; TURING MACHINES

We introduced the PDA as a special case (albeit nondeterministic) of the URM, with a single number-type read/write variable—the stack variable. Actually, we viewed this variable as a *string-type variable* suppressing a detailed look into how a URM can effect string operations such as "pop" and "push", until now. As in Section 2.11, we can identify strings over a finite alphabet of m symbols with natural numbers—the

latter written in *notation base-*$(m + 1)$. The correct way to do this (cf. 2.11.0.32, III) is to *fix an order* of the alphabet and identify its members $\{a_1, a_2, \ldots, a_m\}$ with the "digits" $1, 2, \ldots, m$ (i.e., a_i with i). Thus a non-empty string

$$a_{j_r} a_{j_{r-1}} \cdots a_{j_0} \tag{1}$$

is uniquely *represented* by (and *represents*) the number

$$j_r(m + 1)^r + j_{r-1}(m + 1)^{r-1} + \cdots + j_1(m + 1) + j_0$$

0 corresponds to the string ϵ.

Assume now that the *stack variable* is $\mathbf{x}$, and that its *contents* is the string γ, which, in detail, is the string in (1). Moreover, let us have the stack top located at the *right end* of γ rather than the left end.

Pause. Why is this not an about face of any significance *vis à vis* our earlier conventions?◀

Then we can do a pop—and assign the popped symbol in the variable $\mathbf{z}$, if we wish—as follows

$$\mathbf{z} \leftarrow rem(\mathbf{x}, m + 1)\text{—cf. 2.1.2.40} \tag{2}$$

The stack (contents) change corresponds to the effect of

$$\mathbf{x} \leftarrow \left\lfloor \frac{\mathbf{x}}{m + 1} \right\rfloor \tag{3}$$

Both operations can be simulated by simpler URM operations (as the right hand sides are calls to primitive recursive functions).

To push the one-digit contents of a variable $\mathbf{w}$ into $\mathbf{x}$ one simply does

$$\mathbf{x} \leftarrow (m + 1)\mathbf{x} + \mathbf{w} \tag{4}$$

That is, a URM, despite being inherently a "number-processing" device (read: program), can perform all the string operations that are associated with stacks. Clearly, a URM can be written to handle *several* stack variables.

The converse is an important question: Is it possible to simulate *any* URM with one that *only* has stack variables and its *primary instructions*[138] are restricted to be "push" and "pop" (and checking for empty stack)? And if so, how many stacks are needed?

Suppose now that we add a second stack to a PDA. What more can such a turbo-charged PDA do with its two stacks?

Well, we will show that a URM with *just two* stacks—of string type—can simulate the operation of *any number-processing* URM! Here is how [cf. Minsky (1967) and Tourlakis (1984)]:

---

[138]In the standard URM the instructions $\mathbf{x} \leftarrow \mathbf{x} + 1$ and $\mathbf{x} \leftarrow a$ are primary. The instructions $\mathbf{x} \leftarrow \mathbf{y}$ and **goto** L are *derived*.

5.1.0.29 Example. To prepare for the justification of the above claim, let us see first that we can perform $x \leftarrow a x$ in a URM, using just one extra variable y, where $a > 0$. The following fits the specification:

$1 : y \leftarrow 0$
$2 : x \leftarrow x \dotminus 1$
$3 : y \leftarrow y + a$ {**NB.** This is $y \leftarrow y + 1$, a times}
$4 :$ **if** $x = 0$ **goto** 7 **else goto** 2
$5 : y \leftarrow y \dotminus 1$
$6 : x \leftarrow x + 1$
$7 :$ **if** $y = 0$ **goto** 8 **else goto** 5
$8 :$ **stop**

$\square$

5.1.0.30 Example. We next see how to do

$$x \leftarrow \textbf{if } rem(x, a) = 0 \textbf{ then } \lfloor x/a \rfloor \textbf{ else } x$$

in a URM with just two variables, where $a > 0$. Below we present a pseudo URM program for the task. For the simulation of "**goto** L" see 2.1.1.8.

$\qquad y \leftarrow 0$
$Loop :$ **if** $x = 0$ **goto** L_0 {case of $rem = 0$}
$\qquad x \leftarrow x \dotminus 1$
$\qquad$ **if** $x = 0$ **goto** L_1 {case of $rem = 1$}
$\qquad \vdots$
$\qquad x \leftarrow x \dotminus 1$
$\qquad$ **if** $x = 0$ **goto** L_{a-1} {case of $rem = a - 1$}
$\qquad x \leftarrow x \dotminus 1$
$\qquad y \leftarrow y + 1$
$\qquad$ **goto** $Loop$

At this point

$$y = \left\lfloor \frac{\text{original value of } x}{a} \right\rfloor$$

$L_0 \quad : x \leftarrow ay$
$\qquad$ **goto** L
$\qquad$ Cases where $rem(x, a) \neq 0$; we need to restore x.
$L_1 \quad : y \leftarrow ay$
$\qquad y \leftarrow y + 1$
$\qquad x \leftarrow y$
$\qquad$ **goto** L
$\qquad \vdots$

$$L_k \quad : \mathbf{y} \leftarrow a\mathbf{y}$$
$$\mathbf{y} \leftarrow \mathbf{y} + k$$
$$\mathbf{x} \leftarrow \mathbf{y}$$
$$\textbf{goto } L$$
$$\vdots$$
$$L_{a-1} : \mathbf{y} \leftarrow a\mathbf{y}$$
$$\mathbf{y} \leftarrow \mathbf{y} + a - 1$$
$$\mathbf{x} \leftarrow \mathbf{y}$$
$$L \quad : \textbf{stop}$$

□

5.1.0.31 Example. We can now show how two *number-type* variables are sufficient to simulate the function of a URM M that contains more than two variables,

$$\mathbf{z_0}, \ldots, \mathbf{z_m} \tag{1}$$

where $m > 1$. We build (in outline) a URM N that employs just two variables, $\mathbf{x}$ and $\mathbf{y}$. The first stores the values of all the variables in (1) using a modified prime power coding,

$$\mathbf{x} = p_0^{\mathbf{z_0}} \times \ldots \times p_m^{\mathbf{z_m}} \tag{2}$$

where $p_1 = 2, p_2 = 3$, etc.

To simplify matters, assume that M has only $\mathbf{z}_0$ as its input/output variable. The simulator N will have $\mathbf{x}$ as its only *input variable*, but it will "read" an input a *encoded* as the number 2^a—this means that the coding phase, going from a to 2^a is not part of the simulation. By our conventions on non-input variables, all of $\mathbf{z}_1, \ldots, \mathbf{z_m}$ are "implicitly" set to 0, that is why the correct initial value of $\mathbf{x}$ is indeed just 2^a. Furthermore, without loss of generality, we restrict M so that it has no instructions of the type $\mathbf{w} \leftarrow s$.

Pause. Why "without loss of generality"?◄

Once (2) has been initialized, an instruction such as $\mathbf{z}_i \leftarrow \mathbf{z}_i + 1$ is simulated by N doing $\mathbf{x} \leftarrow p_i\mathbf{x}$. On the other hand, $\mathbf{z}_i \leftarrow \mathbf{z}_i \dotminus 1$ will require N do $\mathbf{x} \leftarrow \textbf{if } rem(\mathbf{x}, p_i) = 0 \textbf{ then } \lfloor \mathbf{x}/p_i \rfloor \textbf{ else } \mathbf{x}$. Finally, the instruction **if** $\mathbf{z}_i = 0$ **goto** L' **else goto** L'' of M is simulated by N by, in a first approximation, the instruction **if** $rem(\mathbf{x}, p_i) = 0$ **goto** L' **else goto** L''. This can be implemented by a modification of the URM in 5.1.0.30, where the instructions under labels $L_1, \ldots, L_{p_i}$ transfer to L'' (rather than to the label of **stop**), and the instructions under label L_0 transfer to L' rather than to L. □

5.1.0.32 Remark. We can easily do what the previous three examples did with two *string-type* stack variables—"two stacks" as we say. Let us call these variables $\mathbf{L}$ and $\mathbf{R}$. They will be utilized as a simple *counters*. This means that the *length of the stored string* represents the *number* stored—*no matter what the actual symbols in the stack may be.*

Pause. Therefore a stack alphabet Γ of any one symbol, say, $\Gamma = \{\#\}$, will work.◄

Thus, "$\mathbf{L} \downarrow$ (push)" implements $\mathbf{L} \leftarrow \mathbf{L} + 1$ and "if $\mathbf{L} \neq \epsilon^{139}$ then $\mathbf{L} \uparrow$ (pop)" implements $\mathbf{L} \leftarrow \mathbf{L} \dotminus 1$. We also note that in the simulation in Examples 5.1.0.29, 5.1.0.30, and 5.1.0.31 we employed the variables $\mathbf{x}$ and $\mathbf{y}$ in what can be termed *push-pull*[140] mode, that is, when one popped ("pulled"), the other pushed—since we *paired* instructions

$$\mathbf{x} \begin{array}{c} \dotminus \\ + \end{array} 1$$

with

$$\mathbf{y} \begin{array}{c} + \\ \dotminus \end{array} 1$$

We have abstracted in the above observation the detail that sometimes we had a "$+ a$" paired with a "$\dotminus 1$", e.g., simulation of $\mathbf{x} \leftarrow a\mathbf{x}$. This is legitimate since a "push" like "$+ a$" can be thought of as one push, rather than a consecutive pushes, if we allow groups of symbols to be pushed at once. In the context of this simulation this can be "hardwired" into the simulating program since there are only a finite number of different "a": $2, 3, 5, \ldots, p_m$.

Thus two string-type stacks—even if they are restricted to *not* work independently but rather are constrained to work in push-pull mode—give the 2-stack PDA formalism at least as much power as that of the URM.

Conversely, and rather trivially, a URM can simulate any *deterministic* program written in this new formalism, since as we know (e.g., preamble of Section 5.1) the number-theoretic URM can pop, push, and check for empty stack (equal to 0, number-theoretically). □

What about nondeterminism? Does it give *more* computing power to the 2-stack PDA formalism vs. that of the URM?

It turns out that it does not.

This can be seen in at least one straightforward way. On one hand, a nondeterministic URM—which we will not define formally—is one that *allows choice for next instruction*. By preceding remarks, such a URM subsumes the new formalism completely.

On the other hand, a nondeterministic URM can be simulated by a deterministic URM—by one of our standard URM, that is. Why? Because we *can* do this:

- Recast Definition 2.3.0.5 for nondeterministic URM.

---

[139]If $\mathbf{L}$ were a numeric variable, then, of course, one could test for the condition $\mathbf{L} \neq 0$ (non-emptiness) since the number 0 corresponds to the string ϵ. On the other hand, a "string stack"—like the described here "counter"—will not *know* when it is empty. We can get around this by initializing the stacks with a special "bottom" symbol. For example, \$. Emptiness then is equivalent with \$ being the top symbol.

[140]A term that I borrowed from circuit design where two transistors are connected in push-pull mode if, in operation, and at any given instance in time, when one's current goes down by a certain amount, the other's increases by the same amount, and vice versa.

- Define acceptance of an input $\vec{x}_n$ to be equivalent to *input $\vec{x}_n$ leads to* at least one *terminating computation.*

- Define that a relation $R(\vec{x})$ is URM-acceptable iff, for some nondeterministic URM M, "$R(\vec{x})$" is equivalent to "$\vec{x}$ is accepted by M".

- Do the arithmetization of the nondeterministic URM, by trivially modifying all the work that we did following 2.3.0.5, which led to the normal form theorem.

- Prove a new "Kleene Normal Form Theorem" for nondeterministic URM, namely:

  For any nondeterministic URM M of code z, we have that M accepts input $\vec{x}_n$ iff $(\exists y)T_{ND}^{(n)}(z, \vec{x}_n, y)$—where, for every n, $T_{ND}^{(n)}$ is a primitive recursive "Kleene predicate" for nondeterministic URMs.

After all this work, which is tedious but easy, we have that the sets that are (nondeterministic) URM-acceptable are precisely the c.e. sets. Therefore, nondeterminism bought us nothing in terms of power in the URM formalism.[141] This is analogous with the situation of FA vs. NFA.

5.1.1 Turing Machines

Let us now carefully define the *Turing machine*—acronym *TM*—formalism of Turing (1936, 1937), since we want to prove a metatheorem about it. We will essentially identify the Turing machine with the 2-stack PDA but will offer some simplifications in the eventual definition—in particular, we will join the two stacks into a *single variable* of type "string". Let us make a few additional observations before we embark on definitions.

First, we adhere to the traditional point of view according to which "automata"—whether these are FA, NFA, PDA, or the 2-stack PDA—are *string-processing programs* where *concatenation* rather than "$+1$" or "$\dotminus 1$" are the *primitive* operations in the formalism. Each such automaton has an input alphabet Σ over which its string inputs are formed. For the 2-stack PDA, an *instantaneous description* of its computation is partly determined by the contents of the two stacks, $\mathbf{L}$ and $\mathbf{R}$, by the *current instruction number* or *state* that the program is at, and by the symbol on top of the stack.

Pause. Which one of the two stacks?◄

Since the stacks operate in push-pull, we imagine the ID as the quadruple

$$\langle \mathbf{l}, q, a, \mathbf{r} \rangle$$

where $\mathbf{l}$ and $\mathbf{r}$ are the contents of $\mathbf{L}$ and $\mathbf{R}$, respectively, q is the current state, and a is the *current symbol*, which we arrange always to be the top symbol of $\mathbf{R}$.

The top of $\mathbf{L}$ is its rightmost symbol, while the top of $\mathbf{R}$ is its leftmost symbol.

---

[141]It turns out that deterministic (1-stack) PDA are strictly less powerful than the (1-stack) PDA.

This ID ignores the input stream that is read via the input, read-only, variable of digit-type (single-symbol-type). The reason is that it turns out that the presence or absence of a read-only input variable, *provably*, does not add or subtract to the power of the model. Thus, without loss of generality we will henceforth assume that there is *no* input variable and that the input appears in the stack **R** *initially*.

We will not offer the proof that this simplification of the model indeed is "without loss of generality". It is well known that the TM (the 2-stack PDA) is quite oblivious to *large* variations in its definition, including the one we mentioned. Several startlingly different but equipotent variants of the TM are presented in, for example, Hopcroft *et al.* (2007). In particular, the reader can find in Tourlakis (1984) a full account of how the TM that we outlined earlier—that is, the PDA with two stacks that are implemented as counters—can be modified, still within the TM model as it is defined in 5.1.1.1 below, to simulate a URM, including the part of transforming the input from a to 2^a (the "input format" 2^a was used in 5.1.0.31), this being done explicitly this time as part of the computation.

At long last, we collect what we have said so far in this chapter, and define the TM by actually "gluing" together the two stacks, *top to top*, an operation that makes sense due to their push-pull operation: *Thus we have just one read-write string variable* in a TM, to which the literature most often refers as the "*tape*" [a notable exception is Papadimitriou (1994)].

5.1.1.1 Definition. (The Turing Machine) A *Turing machine* (abbreviation *TM*) is a collection of tools, $M = \langle Q, T, \Sigma, \Gamma, q_0, F, \mathcal{I} \rangle$, as follows: Q is a finite set of states, that is, *instruction labels*. T is the only variable, a string-type variable with special access mode (cf. 5.1.1.2 below). Most of the literature refers to is as "the *tape*" of the TM.

$\Gamma = \{B, \ldots\}$ is a finite alphabet over which the contents of the tape are formed. The symbol B occurs in all TM alphabets, is called the *blank symbol*, and is *special*. More on it later (5.1.1.2).

Σ is the *input alphabet*; it satisfies $\Sigma \subseteq \Gamma - \{B\}$. q_0 is the label of the instruction that must be executed first—the *start state*. F is a possibly empty, finite set of accepting states.

Finally, $\mathcal{I}$ is a finite set of *instructions* of the forms (1)–(3) below, where our notation adheres to the rules that a, b, c, possibly with primes or subscripts, denote generically members of Γ; p, q, r, possibly with primes or subscripts, denote members of Q. The TM can access at any *one* step of the *computation* ("computation" to be defined in 5.1.1.2) *one* symbol of the string stored in T—*any* one symbol—which is referred to as the "current" symbol.

(1) "$qabq' \rightarrow$"; this instruction is *applicable only if* the current instruction is (labeled) q and the current symbol is a. The execution of the instruction causes the symbol a to change to b. *Note that a and b may denote the same symbol!*

It also causes q' to be the next instruction. *Note that q and q' may denote the same state!* The new scanned symbol will be the one immediately to the right of a (which has changed to b).

(2) "$qabq'$ $\leftarrow$"; this instruction is applicable *only if* the current instruction is q and the current symbol is a. The execution of the instruction causes the symbol a to change to b. It also causes q' to be the next instruction. The new scanned symbol will be the one immediately to the left of a (which has changed to b).

(3) "$qabq'$"; this instruction is applicable *only if* the current instruction is q and the current symbol is a. The execution of the instruction causes the symbol a to change to b. It also causes q' to be the next instruction. The new scanned symbol will be the b, which is in the place of the original a.

If there are *no* two *distinct* instructions in $\mathcal{I}$ that start with the same state and symbol pair, "qa", then the TM is *deterministic*; otherwise, it is *nondeterministic*.   □

It is immediate that every deterministic TM is also nondeterministic, but not vice versa. (This does *not* mean that we cannot simulate any nondeterministic TM by a deterministic TM. This is only a comment about the *structure* of Turing machines.)

5.1.1.2 Definition. (IDs and Computations) Given a TM M as in 5.1.1.1. An instantaneous description or ID of a computation of M[142] is a string

$$tqau$$

where a is the scanned symbol, q is the current instruction label (state) and tau is the string stored in the variable T—the total "tape contents".

Connecting with the discussion that led us here, t is the content of the stack variable **L** (top at the right end of t) and au is the content of the stack variable **R** (top at the left end of au).

Two IDs, $tqau$ and $t'q'a'u'$ are related via the relation $\vdash$,

$$tqau \vdash_M t'q'a'u', \text{ where } M \text{ is omitted if it is understood from the context} \quad (1)$$

in words, $tqau$ yields $t'q'a'u'$, iff one of (i)–(iii) holds:

(i) $t = t'$, $u = u'$, $a' = b$ and $qabq'$ is an instruction

(ii)   This is "push in **L**, pull (pop) from **R**"

  • $t' = tb$, $u = a'u'$, with $a' \in \Gamma$, and $qabq' \rightarrow$ is an instruction

  • $t' = tb$, $u = \epsilon$, $u' = \epsilon$, $a' = B$, and $qabq' \rightarrow$ is an instruction

(iii)   This is "push in **R**, pull from **L**"

  • $t = t'a'$, $u' = bu$, with $a' \in \Gamma$, and $qabq' \leftarrow$ is an instruction

  • $t' = B$, $t = \epsilon$, $a' = B$, $u' = bu$, and $qabq' \leftarrow$ is an instruction

---

[142] Again, no circularity here!

The second bullet in each of (ii) and (iii) shows the special nature of the blank symbol B. Intuitively each bullet formalizes how each half of the "tape"—that was obtained by joining two stacks, top to top, as you recall—can be *extended* when the computation reaches either of the two extremes and "wants" to go further. In stack terms, the two stacks never "*underflow*". [143]

An ID of the form $q_0 x$, where $x \in \Sigma^+$ is the input string, is called *initial*. An ID $tqau$ that has *no next IDs*, that is, $\neg(\exists J)(tqau \vdash_M J)$, is called *terminal* or *final*. An ID $tqau$ is called *accepting* iff it is terminal *and* $q \in F$.

A *computation* of M—an M-computation—is a finite sequence of IDs J_i, for $i = 0, \ldots, m$, such that

(1) J_0 is initial

(2) J_m is terminal

(3) $J_i \vdash_M J_{i+1}$, for $i = 0, \ldots, m-1$

A subsequence $(J_i)_{i=r}^k$ is a *subcomputation* iff it satisfies (3) for all pairs of consecutive terms.

A computation *accepts the input* x iff its initial ID is $q_0 x$ and its terminal ID is accepting. The symbol $L(M)$, where M is a TM, denotes the *language accepted by* M, that is, $\{x \in \Sigma^+ : x$ is accepted by $M\}$.

The *complexity* or *run time* of a computation $J_0, \ldots, J_m$ is the number m—which equals the number of "steps", $\vdash_M$, applied in (3) above. $\square$

Since TMs can *write* on their *tape* (string variable) they do not need accepting states to indicate acceptance of a string—for example, just prior to halting, they may erase the contents of the variable, and replace them with the word "accept", or the word "true", or the number 1, to indicate that the input was accepted. Nevertheless, the presence of accepting states makes arguments about the behavior of TMs—such as the proof of Cook's theorem in this section—easier.

Turing machines can do more than produce yes/no output. They can compute functions just like the URM does. Indeed, the two formalisms have the same computing power (Examples 5.1.0.31 and 5.1.0.32).

However, we will *not* explore the Turing machine in its function-computer role. *All our Turing machines will be acceptors.*

The reader who wants to see how computability can be founded on the TM formalism may consult Davis (1958) and Tourlakis (1984).

5.1.1.3 Definition. (The Classes $\mathcal{NP}$ and $\mathcal{P}$) Given a function $\lambda n.T(n)$. We say that the language L has *deterministic $T(n)$-time complexity* iff there is a *deterministic* TM M such that $L = L(M)$ and the run time of *every computation* of M is *bounded by $T(|x|)$*, where $|x|$ is *the length of the input x*.

---

[143] A stack "underflow" is the state where a stack is empty and yet we attempt to pop.

We say that the language L' has *nondeterministic $T(n)$-time complexity* iff there is a *nondeterministic* TM N such that $L' = L(N)$ and *every* acceptable *input x of N* has at least one computation of run time $\leq T(|x|)$.

We often use the terminology "M (respectively, N) *accepts x within time $T(|x|)$*". We say that L has *deterministic polynomial-time (or poly-time) complexity* iff it has deterministic $T(n)$-time complexity for some polynomial $T(n)$. We say that L' has *nondeterministic poly-time complexity* iff it has nondeterministic $T(n)$-time complexity for some polynomial $T(n)$.

The class of all languages that have *deterministic* poly-time complexity is denoted by $\mathscr{P}$. The class of all languages that have *nondeterministic* poly-time complexity is denoted by $\mathscr{NP}$.   □

What the *nondeterministic* machine does *when it does* not *accept x*—it might even compute forever for some rejected x—has no bearing on the complexity of L'.

In essence, a *nondeterministic machine* recognizes *just* the positive instances of a relation of the form

$$(\exists y)R(x, y) \tag{1}$$

by accepting all x that make (1) true—but no other inputs. We may imagine that this is done as follows:

- It first *nondeterministically* "writes down" ("guesses") a value of y that works for the given x

- It then *deterministically verifies* the truth of $R(x, y)$.

In other words, the machine works precisely as a mathematician does, when the latter approaches the task of proving (1) for a given y.

The run time of the task described by the two bullets equals the time it takes to write down y, plus *the time it takes for the verification.*

Pause. But surely deterministic and nondeterministic *steps* in a computation are *in general* intertwined? How can (1) be *the* "general" case?◄

It is. Every set $S \in \mathbb{N}$ that is acceptable by a *nondeterministic* URM (or TM) is c.e.

Cf. the discussion in the ⬦-passage on p. 329. Thus the relation $x \in S$ is equivalent to $(\exists y)Q(x, y)$ for some primitive recursive Q, by the projection theorem. Thus, χ_Q can be computed by a a loop program.

Consider two examples:

Suppose first that we want to (nondeterministically[144]) *prove* that some given x is a composite number.

If such a claim about x is true, then we would probably want to *guess* a factor y and then verify that it works, by computing $rem(x, y)$ and making sure that its value is 0.

---

[144]The reader no doubt has noted that most proofs *are* nondeterministic!

The next example addresses the question of how the "problem-template" (1) can represent a problem where multiple "guessed values" are needed: Suppose that we want to prove that a given Boolean formula $\mathscr{X}$—whose Boolean variables are (in metanotation) $v_1, \ldots, v_n$—is satisfiable. We can use as a *single* guessed value y an appropriate coding of n *appropriate* truth values, which—when assigned into the v_i—will make $\mathscr{X}$ true. Of course, if we hope to "solve" the problem in poly-time with respect to the input size, $|\mathscr{X}|$, we will need to use a coding/decoding scheme that can pack/unpack the truth-values into/from a single y, and do so within time that is a polynomial function of $|\mathscr{X}|$—the length of the formula. One such fast coding/decoding scheme is to form a string formed by concatenating all the guessed truth-values together. See also 5.1.2.13 below.

5.1.1.4 Remark. In 5.1.1.3 we have quoted run time *as a function of input length*. This is a natural choice in complexity theory that deals with a large variety of problems that involve non-numerical inputs. For example, problems about *graphs*—such as the *clique problem*: "given a graph G and a number k; does G have a clique of size k?"—and problems about formulae—such as the *SAT problem*: "given a Boolean formula $\mathscr{A}$; is it satisfiable?"

Of course, every set of strings over an alphabet Σ of k elements can be viewed as a set of numbers, written base-k, as we already have remarked (e.g., Section 2.11). However, *quoting run time in terms of input value is not*, in general, *the same as quoting time in terms of input length, unless $k = 1$*. Here is why:

Each language L discussed in this and the next subsection is assumed to be over some alphabet Σ_L of $k \geq 2$ elements. The length of a string x over Σ_L—viewed as a number base-k—essentially equals the logarithm base-k of x. Indeed, if

$$x = a_n k^n + a_{n-1} k^{n-1} + \cdots + a_0$$

then

$$k^n \leq x \leq (k-1)k^n + (k-1)k^{n-1} + \cdots + (k-1)$$

therefore,

$$k^n \leq x \leq (k-1)\frac{k^{n+1} - 1}{k - 1} = k^{n+1} - 1 < k^{n+1} \tag{1}$$

Now, $n + 1 = |x|$ and, by (1), $|x| - 1 \leq \log_k x < |x|$. That is, $|x| = \lceil \log_k x \rceil$.

Often we take our time bound functions $T(n)$ from some class of functions, $\mathcal{C}$, e.g., the class of polynomials, or the class $\mathcal{PR}$. Obviously, if a TM runs within time n^c for some constant c (a polynomial run time) it makes a world of difference if we are quoting run time as a function of the *input value* or of the *input size, when $k \geq 2$*: because x (the input value) is equal to about $k^{|x|}$ and thus a TM that runs within time x^c—a polynomial bound *with respect to input value*—runs within time $(k^{|x|})^c$, which is an exponential bound *with respect to input size (length)*.

Therefore, for time bounds from the class of polynomials we *must* be clear how we quote run time! *In this case we invariably do so with respect to input length.*

On the other hand, if $T(n)$ comes from a class that contains exponentials, e.g., T is in $\mathcal{PR}$ or in the class $\mathscr{L}_2$ of Section 5.2, then it is irrelevant how we quote run time, whether with respect to input value or input size, since $\lambda n.k^{T(n)} \in \mathcal{PR}$ as well. $\qquad\square$

5.1.1.5 Remark. Since every deterministic TM is also a nondeterministic TM, Definition 5.1.1.3 yields at once

$$\mathscr{NP} \supseteq \mathscr{P}$$

The converse inclusion, and therefore the question "$\mathscr{NP} = \mathscr{P}$?" of Cook (1971) is a major open problem of complexity theory. In words, it asks: *can we eliminate* guessing *from a program that accepts in polynomial time to obtain a new, equivalent, program that also runs in polynomial time?* We turn now to concepts on which this question hinges. $\qquad\square$

5.1.1.6 Definition. (Feasible Computations) That the computations of a TM are *feasible* means that they are poly-time bounded, *the bound being quoted with respect to input length.* $\qquad\square$

5.1.1.7 Remark.   (I) Thus, *unfeasible* or *intractable*—these are technical terms!— computations are those that require exponential run time or more. The distinction is fairly natural, but not totally. It leads to a nice theory, however. From a *practical* standpoint, a TM that runs within time n^{350000} has no computations that are any more "feasible" than one that runs within time 2^n. The former is as bad with an input of length 2 as the latter is with an input of length 350000.

Cobham (1964) has given a *machine-independent characterization* of the class of *feasibly computable* functions.

(II) *Machine-independent*? How can this be possible? Are *all* machines as "fast" (or as "slow")? Do the URM and the TM, for example, spend the same amount of time for every problem that one throws at them and for every input thereof?

Well, not *the same*, and "all machines" is a rather loose concept. However, it is a known fact that if we equip the URM with the proper set of *primitive* instructions and choose the concept of "step" carefully, so that it is mindful of the *length of the number* stored in a variable $\mathbf{x}$, then *any* URM that has computations running within time $t(n)$ (n being the length of input) can be simulated by some TM within time $p(t(n))$, for some polynomial p of very small degree. That the converse simulation, TM by URM, also entails negligible run time loss—a multiplicative constant overall, in fact—is rather immediate, once one notes that the TM is essentially a a 2-variable URM (Subsection 5.1.1) with the instructions $\mathbf{x} \leftarrow m \times \mathbf{x}$ and $\mathbf{x} \leftarrow \lfloor \mathbf{x}/m \rfloor$ built in as primitives, and executable in one step each.[145] Thus,

---

[145]Cf. also Papadimitriou (1994), where it is shown that a TM can simulate in polynomial time even an "augmented" URM that can access an unbounded number of registers via indirect addressing. Such a URM is called a *random access machine* in the literature, a *RAM*.

for polynomial time bounds it does not matter if we compute with URMs or with TMs

as long as the URM programming language that is called upon to compute functions $f : \Sigma^* \to \Sigma^*$, for some fixed Σ of $m > 1$ elements, is properly equipped—in the interest of efficiency of computations—as follows: We *add* to the basic model of 2.1.1, the instructions $\mathbf{x} \leftarrow \mathbf{x} * d$,[146] where $d \in \Sigma$, and $\mathbf{x} \leftarrow \lfloor \mathbf{x}/m \rfloor$ as *primitives* (cf. also the preamble of Section 5.1). Failing to do so would result in "charging" an inordinate amount of steps to just add or delete a *single digit* at the right end of the number (string!) stored in a variable (least significant location) despite the obvious fact that such a string operation ought not to depend on the size of the stored number.

Thus, for a *string-processing URM* one will charge $|\mathbf{x}|$ *steps* for *each* of

$\mathbf{x} \leftarrow \mathbf{x} + 1$

$\mathbf{x} \leftarrow \mathbf{x} \dot{-} 1$

$\mathbf{x} \leftarrow a$

and will charge *one* step for each of

if $\mathbf{x} = 0$ **goto** M **else goto** R

$\mathbf{x} \leftarrow \mathbf{x} * d$, where $d \in \Sigma$

$\mathbf{x} \leftarrow \lfloor \mathbf{x}/m \rfloor$

The wisdom for the assigned charge, for example, to the first instruction above stems from the fact that to add 1 to $2^n - 1$ will necessitate to flip n binary digits, from 1 to 0.

(III) Cobham's main theorem is that the class of functions, $\mathscr{C}_\Sigma$, that can be computed *feasibly* is the closure of the set below,

$$\{\lambda x.x * d \text{ (for all } d \in \Sigma), \lambda xy.x^{|y|}\} \tag{1}$$

under *substitutions* (2.1.2.6)—*including* the case of substituting *any constant* into a variable—and *bounded (right) recursion on m-adic notation*, the latter meaning the schema

$$
\begin{aligned}
f(0, \vec{y}_n) &= h(\vec{y}_n) \\
f(x * d, \vec{y}_n) &= g_d(x, \vec{y}_n, f(x, \vec{y}_n)), \text{ for all } d \in \Sigma \\
|f(x, \vec{y}_n)| &\leq |B(x, \vec{y}_n)|
\end{aligned}
$$

where the functions h, $(g_d)_{d \in \Sigma}$, and B are given. Cf. 1.7.0.30.

Cobham prefers m-adic (Exercise 1.8.47) [over the more common $(m+1)$-ary] notation for integers $n > 0$, which—unlike the latter that allows digits

---

[146]The symbol $*$ denotes concatenation.

$0, \ldots, m$—*only allows the digits* $1, \ldots, m$. *Wherever concatenation is used in his paper, it is concatenation of m-adic notations of* numbers.

The notation in (1) and in the bounded recursion schema above should be read *number-theoretically*! Thus, $x^{|y|}$ in (1) is ordinary (number-theoretic) exponentiation but the exponent is the m-adic *length* of the y-argument, while $x * d$—a *right successor* of x—has *value* the number $m \times x + d$ and length $|x| + 1$. The base m of the m-adic notation is the number of elements in Σ. Note that we identify ϵ with 0, as m-adic notation does not apply to 0.

Cobham's theorem is independent of the choice of alphabet Σ *as long as the latter has at least two elements—thus his class can be unambiguously named* $\mathscr{C}$.

As an example, notice the simple recursion on notation that defines $|x|$:

$$\begin{aligned} |0| &= 0 \\ |x * d| &= |x| + 1, \text{ for all } d \in \Sigma \\ \big||x|\big| &\leq |x| \end{aligned}$$

Pause. "Simple", yes; but *only* if we know that $\lambda x.x$ and $\lambda x.x + 1$ are in $\mathscr{C}$!◄

Well, here is how to obtain $\lambda x.x + 1$ by bounded recursion in terms of functions known to be in $\mathscr{C}$:

$$\begin{aligned} 0 + 1 &= 1 \\ x * d + 1 &= x * (d + 1), \text{ for } d = 1, 2, \ldots, m - 1 \\ x * m + 1 &= (x + 1) * 1 \\ |x + 1| &\leq |x * 1| \end{aligned}$$

For more on Cobham's feasibly computable functions (and relations) see Exercises 5.3.1–5.3.11, where you are asked to verify the membership in $\mathscr{C}$ of $\lambda x.1$, $\lambda x.0$, and $\lambda x.x$ as well as that of many other more complex functions. Cobham's result is retold (and proved[147]) in full detail in Tourlakis (1984).   □

5.1.2 $\mathcal{N}\mathscr{P}$-Completeness

There are some languages (sets)[148] in $\mathcal{N}\mathscr{P}$ that have maximal complexity, just as there are maximally complex c.e. sets, such as K.

5.1.2.1 Definition. ($\mathcal{N}\mathscr{P}$-**Hard and** $\mathcal{N}\mathscr{P}$-**Complete Sets**) A set L is $\mathcal{N}\mathscr{P}$-*hard* iff for every $L' \in \mathcal{N}\mathscr{P}$ we have $L' \leq_m L$, where the reduction is effected by a

---

[147]As far as I know, Cobham never published the complete proof.
[148]"Sets" is the habitual term used for sets of numbers. "Languages" is preferred for sets of strings over some alphabet. The two concepts, as we know, are equivalent, and the nomenclature only reflects (momentarily) a point of view, not a difference of any import.

function that is computable by some URM in a number of steps that is a polynomial function of input length.

> We say that we have a polynomial-time or poly-time reduction and write $L' \leq_p L$ (p for polynomial).

A language L is called $\mathcal{N}\mathcal{P}$-complete iff it is $\mathcal{N}\mathcal{P}$-hard and, moreover, is a member of $\mathcal{N}\mathcal{P}$.   □

5.1.2.2 Remark. The Turing formalism, as we have already noted, is well positioned to be used in the proof of Cook's theorem. Thus we have introduced it here as an acceptor device.

On the other hand, we have not developed the very primitive Turing machine formalism as a basis of computability. *Thus our reduction functions are computed by URMs.*

Another way to state this, in a machine-independent manner, is that these poly-time reductions are effected using *feasibly computable* functions from Cobham's class $\mathscr{C}$ (cf. discussion in 5.1.1.7).   □

5.1.2.3 Definition. (Big-O Notation) Given $g : \mathbb{N} \to \mathbb{N}$. $O(g(n))$ is the set of all $f : \mathbb{N} \to \mathbb{N}$ such that, for some constant C, we have $f(n) \leq Cg(n)$ a.e. (cf. 2.4.2.8). The notation $f(n) = O(g(n))$, introduced by the number-theorist E. Landau, means $f(n) \in O(g(n))$ and is called *big-O notation.*

> *Expressions in big-O notation are read from left to right.* In particular, $O(h(n)) = O(g(n))$ is abuse of notation for $O(h(n)) \subseteq O(g(n))$. Thus, it means that "for every f, if $f(n) = O(h(n))$, then it is also the case that $f(n) = O(g(n))$".

> *Some operations defined:* $f(n) \in O(h(n)) + O(g(n))$ iff $f(n) \leq f'(n) + f''(n)$ a.e., with $f'(n) \in O(h(n))$ and $f''(n) \in O(g(n))$.

> $f(n) \in O(h(n)) \times O(g(n))$ iff $f(n) \leq f'(n) \times f''(n)$ a.e., with $f'(n) \in O(h(n))$ and $f''(n) \in O(g(n))$.

> $f(n) \in O(h(n))^{O(g(n))}$ iff $f(n) \leq f'(n)^{f''(n)}$ a.e., with $f'(n) \in O(h(n))$ and $f''(n) \in O(g(n))$.   □

5.1.2.4 Example. It is readily verifiable that $O(n) = O(n^2)$. But it is not true that $O(n^2) = O(n)$. Recall, "*Expressions in big-O notation are read from left to right*". Also, $O(n) + O(n^2) = O(n^2)$ and $O(n)O(n^2) = O(n^3)$. Incidentally, writing $f(n) = O(1)$ means that $f(n)$ is bounded a.e. by a constant.   □

5.1.2.5 Exercise. It is immediate that $O(n^{n^2}) = O(n)^{O(n^2)}$. Is it also $O(n)^{O(n^2)} = O(n^{n^2})$?   □

5.1.2.6 Example. If $p(n)$ is a polynomial of *degree* k, then $p(n) = O(n^k)$. The reader will recall that the degree of a polynomial in the variable n is the highest exponent of n that occurs in the polynomial. Indeed, let

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0, , \text{ where } a_k > 0$$

Without loss of generality, all the integer coefficients a_i are natural numbers; if not, we replace them all by their absolute values and work instead with the so obtained $p'(n)$ that obviously satisfies $p(n) \leq p'(n)$ for all n. We next note that

$$p(n) = n^k \left(a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} \cdots + \frac{a_0}{n^k} \right)$$

Taking $n > \max\{a_i : i = 0, \ldots, k-1\}$ we get $a_i/n^{k-i} < 1$, for $i = 0, \ldots, k-1$. Thus, if we set $C = a_k + k$ we have $p(n) \leq Cn^k$, a.e. $\quad\square$

5.1.2.7 Remark. In view of 5.1.2.6, we replace every statement of the form "... within poly-time $T(n)$ ..." by "... within time $O(n^k)$, for some k ...", where "within time $O(n^k)$" means "within time $f(n)$ *in* $O(n^k)$". $\quad\square$

5.1.2.8 Remark. We note here that if $\lambda x.f(x)$ is computable by a URM—as this was modified in 5.1.1.7—within time $t(|x|)$, then $|f(x)| \leq t(|x|) + |x|$. For in the worst case, all instructions (besides **stop**) are $x \leftarrow x * d$. In one step we have the length of x increase by 1 in such an instruction. Thus, in $t(|x|)$ steps the length of any variable increases by $t(|x|)$, at most. $\quad\square$

The following is fundamental but easy. It is the counterpart of Proposition 2.7.2.3.

5.1.2.9 Proposition. *If $L' \leq_p L$ and $L \in \mathscr{P}$, then also $L' \in \mathscr{P}$.*

Proof. Let $x \in L'$ iff $f(x) \in L$, where f is polynomial-time computable—say, within time $O(n^m)$, for some m—where $n = |x|$. By assumption on L, $z \in L$ is verified or rejected by a deterministic TM within $O(n^k)$ steps, for some k, where $n = |z|$.

The polynomial-time algorithm for the verification or rejection of the claim $x \in L'$ then is:

(1) *we compute $f(x)$.* As this runs within time $O(n^m)$, we have that

$$|f(x)| = O(n^m), \text{ where } n = |x| \tag{$*$}$$

cf. 5.1.2.8 and note that $n + O(n^m) = O(n^m)$;

(2) *we run the algorithm for "$z \in L$",* using $z = f(x)$. This will conclude the verification/rejection of $x \in L'$ within time

$$\underbrace{O(n^m)}_{\text{for } f(x)} + \underbrace{O(n^{km})}_{\text{for } f(x) \in L} = O(n^{km}) \qquad\qquad \square$$

The proof outline above should suffice. The suppressed (tedious) details include "reprogramming" the computation of f in the TM "language", and then composing (combining) this TM with the one that deterministically solves "$z \in L$?"

5.1.2.10 Corollary. $\mathscr{P} = \mathscr{NP}$ *iff $\mathscr{P}$ contains some $\mathscr{NP}$-complete language L.*

Proof. We will see soon that $\mathcal{NP}$-complete languages exist.

The *only if* is trivial, since every $\mathcal{NP}$-complete language is in $\mathcal{NP}$. The *if* part is just as trivial: If $L \in \mathcal{P}$ is $\mathcal{NP}$-complete, then consider an arbitrary $L' \in \mathcal{NP}$. By definition, $L' \leq_p L$. Thus, by 5.1.2.9, $L' \in \mathcal{P}$, proving $\mathcal{NP} \subseteq \mathcal{P}$. $\square$

The following is the counterpart of 2.10.0.12.

5.1.2.11 Proposition. *A set L in $\mathcal{NP}$ is $\mathcal{NP}$-complete iff $S \leq_p L$, where S is known to be $\mathcal{NP}$-complete.*

Proof. For the *if*, let B be in $\mathcal{NP}$. Then $B \leq_p S$; say, via $f(x)$ that runs within time $O(n^k)$ $(n = |x|)$ on some URM M. Let also g effect $S \leq_p L$ within time $O(n^r)$ (computed by some URM N). Thus,

$$x \in B \text{ iff } f(x) \in S \text{ iff } g(f(x)) \in L$$

Note that $g(f(x))$ is computed by the concatenation of M and N (in that order) within time $O(n^k) + O(|f(x)|^r)$. By 5.1.2.8, $|f(x)| = O(n^k)$. Thus the concatenation of M and N runs within time $O(n^k) + O(|f(x)|^r) = O(n^{kr})$.

The *only if* is immediate since L is $\mathcal{NP}$-complete and S is in $\mathcal{NP}$. $\square$

At the beginning of Section 3.4 we (re)defined—via BNF—the set of *Boolean formulae* over the variables-set

$$p, p', p'', p''', \ldots$$

employing only two connectives, $\neg$ and $\vee$. Their semantics was given in Definition 1.1.1.26—a point where we circumvented an explicit definition of Boolean formulae, and defined instead the concept of tautology directly, for any *mathematical formula*. For the record,

5.1.2.12 Definition. (Satisfiable Boolean Formulae; SAT) A Boolean formula, $\mathscr{A}$, as defined in Section 3.4, is *satisfiable* iff there is *at least one* truth assignment to its variables that renders $\mathscr{A}$ true (**t**; cf. 1.1.1.14).

$\mathscr{A}$ is a *tautology* iff every truth assignment to its variables renders it true.

We denote by SAT the set of all Boolean formulae—as defined in Section 3.4— that are satisfiable. We denote by $TAUT$ the set of all Boolean formulae that are tautologies. $\square$

5.1.2.13 Proposition. $SAT \in \mathcal{NP}$.

Proof. [Outline; cf. the discussion surrounding (1) on p. 334.] A nondeterministic TM that will accept precisely all the $x \in SAT$ will operate as follows:

(1) It will "guess" and write down a string y (over $\{\mathbf{t}, \mathbf{f}\}$) of truth assignments to the variables

$$p^{(n_1)}, p^{(n_2)}, \ldots, p^{(n_r)} \, [149]$$

---

[149]For $n_i = 0$, $p^{(n_i)}$ means p.

of x—these truth assignments are guessed so that they *will* work! As we have no more variables than $|x|$, writing down y is doable within $O(|x|)$ steps.

(2) *Deterministically*, the TM will *verify* that the "guess" indeed works by evaluating x for said truth assignment. It is easy to see that this verification is doable in $O(n^c)$ steps, for a very small c ($n = |x|$). The timing of the overall computation, (1) plus (2), is thus $O(n^c)$. □

The reader will recall—from first year programming courses, for example—the nomenclature *infix notation* meaning that non-atomic formulae are written as $(\neg \mathscr{A})$ or $(\mathscr{A} \vee \mathscr{B})$. An important alternative notation is *postfix notation*, also named *reverse Polish* notation. An easy way to define it is to do so recursively:

Let us denote the postfix notation of a formula $\mathscr{A}$ by $post(\mathscr{A})$. Then, for atomic formulae $\mathscr{A}$ (these are the Boolean variables!), $post(\mathscr{A})$ is just $\mathscr{A}$. On the other hand,

$$post\Big((\neg \mathscr{A})\Big) \text{ is } post(\mathscr{A})\neg$$

and

$$post\Big((\mathscr{A} \vee \mathscr{B})\Big) \text{ is } post(\mathscr{A})post(\mathscr{B})\vee$$

For example, $post(p''')$ is p'''; $post(((\neg p''') \vee p))$ is $p'''\neg p\vee$.

Postfix notation neither utilizes (nor needs) brackets.

Another folklore item from a first year programming course is how to program algorithms, one designed to convert an infix formula to its postfix equivalent, the other to evaluate a postfix expression, given a set of values for all its variables. Both of these algorithms, which we will not describe here, use a stack and, in a URM-like language (like Pascal, Algol, C, etc.) they will run in $O(n)$ steps each, where n is the length of the given infix input. A TM then can do almost as well, that is, finish the task within time $O(n^c)$ for a very small (integer) $c > 0$ (cf. 5.1.1.7).

5.1.3 Cook's Theorem

Cook's theorem [cf. Cook (1971)] is that SAT is $\mathscr{NP}$-complete. He proved this by simulating nondeterministic poly-time bounded accepting TM computations by Boolean formulae, which were satisfiable iff the TM accepted its input. Before we embark on details, we will assume some restrictions—*without loss of generality*, to be sure!—for our TM model.

Restriction 1. The nondeterministic TMs of this subsection have a "*1-way infinite tape*", which is folkloric jargon for: "*the TM will never extend its string variable contents to the left by adding a B symbol*"—that is, the second bullet of (iii) in 5.1.1.2 will never apply. This can be accommodated simply by modifying the TM, if necessary, so that if it need to so add, it will rather first *shift* the contents of its string variable to the right, and add a B as a leftmost symbol. This is legitimate and only "adds" a polynomial number of steps, by Cobham's theorem and

the fact that $\lambda x.B * x$ is in Cobham's class $\mathscr{C}_\Gamma$ (cf. 5.3.1)—Γ being the TMs total tape alphabet.

Restriction 2. Once the TM reaches the *unique* accepting state, q_f, the TM computation *continues forever without changing the ID!* This is entirely analogous with the behavior of URM computations once the **stop** instruction has been reached.

Both the uniqueness and the trivial continuation of the computation upon acceptance can be easily incorporated in a given TM. For example, starting with an accepting state q_f that has no moves, we modify the TM by adding the moves $q_f aaq_f$, for all $a \in \Gamma$.

5.1.3.1 Theorem. (Cook's Theorem; Cook (1971)) *SAT is $\mathscr{N}\mathscr{P}$-complete.*

Proof. So, let $L \in \mathscr{N}\mathscr{P}$, accepted by a nondeterministic TM, M, which is restricted as above. The TM is accepting within time $p(n)$, where p is a polynomial and n is the length of input w. Our task is to construct the function f that effects the reduction

$$L \leq_p SAT$$

Thus, $f(w)$ must be a formula in SAT iff w is acceptable. We must make sure that $f(w)$ is constructed in $O(n^r)$ time, for some r—which will result in $|f(w)| = O(n^r)$.

Let $M = \langle Q, T, \Sigma, \Gamma, q_0, F, \mathcal{I} \rangle$, where $Q = \{q_0, q_1, \ldots, q_f\}, \Gamma = \{a_1, a_2, \ldots, a_m\}$ with $F = \{q_f\}$ and $B = a_1$. We proceed as follows:

(1) Consider the (accepting) computation

$$I_0 \vdash I_0 \vdash I_1 \vdash \cdots \vdash I_l \tag{1}$$

where $l \leq p(n)$, $n = |w|$, $I_0 = q_0 w$ and I_l contains q_f.
By the second restriction adopted for our TMs, we may assume $l = p(n)$.

(2) In $p(n)$ steps, and starting the computation with the ID $q_0 w$, no ID—of the form $tqau$ (cf. 5.1.1.2)—can be longer than $p(n) + 1$. Thus, we will let all IDs, I_j, have the same length, equal to $p(n) + 1$, since we can pad them with B-symbols as needed. This allows us to utilize an *ID-template*, that is, an array of $p(n) + 1$ *locations*—$0, 1, \ldots, p(n)$—and also several Boolean variables, *one for each location*, that will "say" something characteristic about these locations for every point in "time" as the latter is measured by i.

(3) We will construct, for each ID I_i, a Boolean formula $\mathscr{I}_i$ that will be satisfiable iff I_i *has the correct structure* as the i-th ID.

The formula $\mathscr{I}_i$ "knows" that it refers to time step i by virtue of the fact that all its Boolean variables have i as their first subscript—see below. This subscript denotes "time" as it is measured along the computation (1).

(4) We define several mnemonic names that will be the Boolean variables of $f(w)$:

  (a) $S_{i,s}^j$—for $j = 1, \ldots, m$ and $0 \le i, s \le p(n)$—is a variable that is *true* (it is assigned **t**) iff the s-th location of I_i holds a_j.

  We have $m(p(n) + 1)^2 = O(p(n)^2)$ S-variables.

  (b) $Q_{i,s}^j$—for $j = 0, \ldots, f, 0 \le i \le p(n)$ and $0 \le s \le p(n) - 1$—is a variable that is *true* (it is assigned **t**) iff the s-th location of I_i holds q_j.

  Note that a state cannot be in the last location $(p(n))$ since the ID must have the form $tqau$, with $a \in \Gamma$. We have $(f + 1)p(n)(p(n) + 1) = O(p(n)^2)$ Q-variables.

Thus, $\mathscr{I}_i$ that says "*I_i has the correct form at time i, namely, $tqau$*" is the formula that "says", specifically:

  (i) The ID-template has no empty locations.

  (ii) Only *one* location, s, contains a state q_j—i.e., $Q_{i,s}^j$ is **t**—all other locations, t, containing some a_k from Γ—i.e., $S_{i,t}^k$ is **t**.

  (iii) No location holds more than one Q-symbol or one Γ-symbol.

  (iv) The $p(n)$-th location contains a Γ-symbol.

Thus, the abbreviation $\mathscr{I}_i$ stands for the following Boolean formula that we divide into four parts that implement (i)–(iv), respectively.

Below we are using the notation

$$\bigvee_{s \le i \le t} \mathscr{A}_i \stackrel{Def}{=} \left(\mathscr{A}_s \vee \mathscr{A}_{s+1} \vee \cdots \vee \mathscr{A}_t \right)$$

and

$$\bigwedge_{s \le i \le t} \mathscr{A}_i \stackrel{Def}{=} \left(\mathscr{A}_s \wedge \mathscr{A}_{s+1} \wedge \cdots \wedge \mathscr{A}_t \right)$$

Note the brackets! We will also use more complicated ranges, such as

$$\bigvee_{\substack{j \in T \\ s \le i \le t}} \mathscr{B}_j^i$$

an expression that expands into an $\vee$-chain of formulae $\mathscr{B}_j^i$, for all $j \in T$ and all i such that $s \le i \le t$. The order of participating formulae in $\vee$- and $\wedge$-chains is, of course, immaterial. For exposition purposes, we use $\mathscr{X} \wedge \mathscr{Y}$ as an abbreviation for $\neg(\neg\mathscr{X} \vee \neg\mathscr{Y})$.

this is (i): $\bigwedge_{0\leq s\leq p(n)} \left(\bigvee_{1\leq j\leq m} S^j_{i,s} \vee \bigvee_{0\leq k\leq f} Q^k_{i,s} \right)$

$\wedge$

this is (ii): $\bigvee_{0\leq s\leq p(n)} \left(\bigvee_{0\leq k\leq f} Q^k_{i,s} \wedge \bigwedge_{t\neq s} \bigvee_{1\leq j\leq m} S^j_{i,t} \right)$

$\wedge$

this is (iii): $\bigwedge_{0\leq s\leq p(n)} \bigwedge_{\substack{0\leq k,t\leq f \\ k\neq t \\ 1\leq j\leq m}} \left(\neg Q^k_{i,s} \vee \neg(Q^t_{i,s} \vee S^j_{i,s}) \right)$

$\wedge \bigwedge_{0\leq s\leq p(n)} \bigwedge_{\substack{1\leq k,t\leq m \\ k\neq t \\ 0\leq j\leq f}} \left(\neg S^k_{i,s} \vee \neg(S^t_{i,s} \vee Q^j_{i,s}) \right)$

$\wedge$

this is (iv): $\bigvee_{1\leq j\leq m} S^j_{i,p(n)}$

We next define subformulae $\mathscr{Y}_i$ of $f(w)$. For each i, *satisfiability of $\mathscr{Y}_i$ will be tantamount to* $I_i \vdash I_{i+1}$. Thus, $\mathscr{Y}_i$ stands for the formula

$$\mathscr{I}_i \wedge \mathscr{I}_{i+1} \wedge \Bigg[$$

$$\bigvee_{q_j a_x a_y q_k \in \mathcal{I}} \bigvee_{0\leq s<p(n)} \left(Q^j_{i,s} \wedge S^x_{i,s+1} \wedge S^y_{i+1,s+1} \wedge Q^k_{i+1,s} \wedge \right.$$

$$\left. \bigwedge_{\substack{0\leq t\leq p(n) \\ s\neq t\neq s+1 \\ 1\leq k\leq m}} (S^k_{i,t} \equiv S^k_{i+1,t}) \right)$$

$$\vee$$

$$\bigvee_{q_j a_x a_y q_k \rightarrow \in \mathcal{I}} \bigvee_{0\leq s<p(n)-1}{}^{150} \left(Q^j_{i,s} \wedge S^x_{i,s+1} \wedge S^y_{i+1,s} \wedge Q^k_{i+1,s+1} \wedge \right.$$

$$\left. \bigwedge_{\substack{0\leq t\leq p(n) \\ s\neq t\neq s+1 \\ 1\leq k\leq m}} (S^k_{i,t} \equiv S^k_{i+1,t}) \right)$$

$$\vee$$

$$\bigvee_{q_j a_x a_y q_k \leftarrow \in \mathcal{I}} \bigvee_{1\leq s<p(n)}{}^{151} \left(Q^j_{i,s} \wedge S^x_{i,s+1} \wedge S^y_{i+1,s+1} \wedge Q^k_{i+1,s-1} \wedge \right.$$

$$\left. \bigwedge_{1\leq k\leq m}(S^k_{i,s-1} \equiv S^k_{i+1,s}) \wedge \bigwedge_{\substack{0\leq t\leq p(n) \\ t\notin\{s-1,s,s+1\} \\ 1\leq k\leq m}} (S^k_{i,t} \equiv S^k_{i+1,t}) \right) \Bigg]$$

In the above $\bigwedge(S^k_{i,t} \equiv S^k_{i+1,t})$ says that for all k and t, the two symbols in location t *before and after the move* are the same ($\equiv$ is "Boolean equality"). Of course, $\mathscr{X} \equiv \mathscr{Y}$ is short for $(\neg\mathscr{X} \vee \mathscr{Y}) \wedge (\neg\mathscr{Y} \vee \mathscr{X})$.

---

[150]q_k cannot occupy location $p(n)$.
[151]q_k cannot occupy a location to the left of 0.

Putting all this together, $f(w)$ is the formula below, where $w = a_{i_1} a_{i_2} \cdots a_{i_n}$.

$$\bigwedge_{0 \leq i < p(n)} \mathscr{Y}_i \wedge Q^0_{0,0} \wedge \overbrace{S^{i_1}_{0,1} \wedge \cdots \wedge S^{i_n}_{0,n}}^{\text{initial ID}} \wedge \overbrace{S^1_{0,n+1} \wedge S^1_{0,n+2} \wedge \cdots \wedge S^1_{0,p(n)}}^{\text{blanks: } a_1 = B}$$
$$\wedge \bigvee_{0 \leq s < p(n)} Q^f_{p(n),s}$$

The *structure* of $f(w)$ is straightforward and thus the formula $f(w)$ can be *constructed* ("written down") in time that is a polynomial function of its length. Note that $p(n)$ itself is an important parameter of the formula structure, as it determines the range of certain subscripts and the ID-template length. By 5.1.2.6, $p(n)$ is in $O(n^a)$, where a is its degree. Thus, we can assume without loss of generality that $p(n)$ *is* $Cn^a + C'$, for appropriate constants C and C'.

In the most unsophisticated manner $Cn^a + C'$ can be computed by $a - 1$ multi-plications each operating with operands of length $O(\log n)$.[152] This requires a total time of $O((\log n)^2) = O(n)$ steps, using the standard "school method", digit by digit, multiplication algorithm. On the other hand, $n = |w|$ can be obtained from w in polynomial time with respect to $|w|$, for example, due to the fact that $\lambda w.|w| \in \mathscr{C}$ (5.1.1.7).

Next, by inspection, the length of $f(w)$ is estimated as follows:

(a) A subformula $\mathscr{I}_i$ has length $O(p(n)^2)$—due to case (ii)

(b) A subformula $\mathscr{Y}_i$ has length $O(p(n)^2)$

(c) Thus $|f(w)| = O(p(n)^3)$

The above estimate is based on a simplifying first-order approximation that all the Boolean variables $S^j_{i,s}$ and $Q^r_{i,s}$ each have length equal to one. If the super/subscripts are displayed in base-2 notation, then each variable needs space $O(\log_2(p(n))) = O(\log_2 n) = O(n)$ to be written down.[153] Thus $|f(w)| = O(np(n)^3) = O(p(n)^4)$. It flows directly from the construction of $f(w)$ that $w \in L$ iff $f(w) \in SAT$.   □

5.1.3.2 Example. ($CSAT$) In preparation for identifying our next $\mathcal{NP}$-complete problem we review the concept of *conjunctive normal form* (*CNF*). A formula *is* in *conjunctive normal form* iff it has the form below.

$$\bigwedge_{i=1}^{t} \bigvee_{j=1}^{s_i} l_i^j, \text{ where each } l_i^j \text{ is a Boolean variable } p^{(n)} \text{ or a negation thereof}$$

We call each *conjunct* $\bigvee_{j=1}^{s_i} l_i^j$ a *clause* of the CNF. We call each l_i^j a *literal*.

---

[152]Since logarithms of different bases are equal within a constant multiplier, there is no point to indicate the base.

[153]On the other hand, if they are written in unary (k written as a string of k 1-symbols), then each variable needs space $O(p(n))$ to be written down.

It is easy to see that for every Boolean formula $\mathscr{A}$ there is another one, $\mathscr{B}$, that *is* in CNF and they are either both in SAT or neither is. We outline how to construct $\mathscr{B}$ given $\mathscr{A}$.

This can be seen by induction on the number of occurrences of $\vee$ and $\wedge$ connectives in the formula. We assume that Boolean formulae are defined as at the onset of Section 3.4, but this time we will allow all three of $\neg$, $\vee$, and $\wedge$ as primitives.

Prior to the induction, we *preprocess* $\mathscr{A}$ by "pushing" all negations as close to variables as possible. We then replace any maximal odd-length string of $\neg$ symbols by a single one; we eliminate any maximal even-length string of $\neg$ symbols. This is achieved by repeated use of the two de Morgan laws (1.1.1.18) and of the well-known provable identity $\neg\neg\mathscr{X} \equiv \mathscr{X}$. Once this is done, in every subformula of the (fully parenthesized) form $(\neg\mathscr{Y})$, $\mathscr{Y}$ is a variable.

For the basis, if $\mathscr{A}$ is a literal, then it is already in CNF.

For the induction step, let first $\mathscr{A}$ be $\mathscr{B} \wedge \mathscr{C}$. The I.H. yields that $\mathscr{B}$ and $\mathscr{C}$ have associated CNF formulae $\mathscr{B}'$ and $\mathscr{C}'$ as above. A CNF associated with $\mathscr{A}$ then is $\mathscr{B}' \wedge \mathscr{C}'$: If $\mathscr{A}$ is satisfiable, then so are $\mathscr{B}$ and $\mathscr{C}$ and hence $\mathscr{B}'$ and $\mathscr{C}'$ by the I.H. The same is true of $\mathscr{B}' \wedge \mathscr{C}'$. The converse is as easy.

The other induction step deals with the case where $\mathscr{A}$ is $\mathscr{B} \vee \mathscr{C}$. Again, by the I.H. we have associated CNF formulae $\mathscr{B}'$ and $\mathscr{C}'$. Let us invoke Exercise 1.8.2:

$$\mathscr{X} \to \mathscr{Y}, \mathscr{Z} \to \mathscr{W} \vdash \mathscr{X} \vee \mathscr{Z} \to \mathscr{Y} \vee \mathscr{W} \tag{1}$$

The special case where $\mathscr{X}$ and $\mathscr{Z}$ are literals is of import for our I.S. Let p be a *new variable*—i.e., *not occurring* in $\mathscr{B}'$ or $\mathscr{C}'$. Then we have, by (1),

$$\neg p \vee \mathscr{B}', p \vee \mathscr{C}' \vdash \mathscr{B}' \vee \mathscr{C}' \tag{2}$$

where "$p \vee \neg p \to$" is eliminated from the right hand side of $\vdash$ in (2) since $p \vee \neg p$ is a tautology. It follows that

$$(\neg p \vee \mathscr{B}') \wedge (p \vee \mathscr{C}') \vdash \mathscr{B}' \vee \mathscr{C}' \tag{3}$$

Note that, conversely, whenever we satisfy $\mathscr{B}' \vee \mathscr{C}'$ with some array of truth assignments s to its variables, then *there is a truth assignment to p* that when we *append* it to the array s yields a truth assignment that satisfies the left hand side of $\vdash$ in (3).

This appending can happen without conflicts because p is new!

Indeed, if $\mathscr{B}' \vee \mathscr{C}'$ is $\mathbf{t}$ due to $\mathscr{B}'$ being $\mathbf{t}$, then we extend s by letting p to be $\mathbf{t}$. If we know instead that $\mathscr{C}'$ is $\mathbf{t}$, then we extend s by letting p to be $\mathbf{f}$.

This can be summarized as

$$\mathscr{B}' \vee \mathscr{C}' \text{ is satisfiable iff } (\neg p \vee \mathscr{B}') \wedge (p \vee \mathscr{C}') \text{ is}$$

Distributing the two $\vee$ over the CNF forms $\mathscr{B}'$ and $\mathscr{C}'$ above we get the required CNF and we are done with the proof.

The reader can propose an algorithm that implements the above on a string-processing URM and verify that going from $\mathscr{A}$ to a CNF, tracking the above proof, can be done within time $O(|\mathscr{A}|^a)$ for some $a > 0$ (cf. Exercise 5.3.12).

With these preliminaries out of the way, we can now show that $CSAT$, that is, the set of all satisfiable formulae in CNF, is $\mathscr{N}\mathscr{P}$-complete. To this end, note that to prove that $CSAT \in \mathscr{N}\mathscr{P}$ we argue exactly as in the case of SAT (5.1.3.1). For the $\mathscr{N}\mathscr{P}$-hardness part we show that

$$SAT \leq_p CSAT$$

(cf. 5.1.2.11). But this is achieved by the algorithm of Exercise 5.3.12. $\square$

5.1.3.3 Example. Here is a problem, CP, that we can show $\mathscr{N}\mathscr{P}$-complete by effecting the reduction $CSAT \leq_p CP$.

The reader has encountered the concept of a *graph* in discrete math courses. For completeness, let us recall that a *directed graph*, or *digraph*, is, mathematically, a relation $E : V \to V$. We call the members of V *nodes* or *vertices*, and the members of E—the pairs $\langle a, b \rangle$ where $\{a, b\} \subseteq V$—*edges*. A *digraph is finite* iff V is finite. It is habitual to identify a *finite* digraph with a *physical object*, a drawing, where each node is depicted by a dot or a small circle and each edge $\langle a, b \rangle$ is depicted by an arrow that starts on the (physical depiction of) node a and ends (that is where the arrowhead touches) on the node b. Flow-diagrams of FA and PDA are (labeled) digraphs.

A digraph is *complete* iff all possible edges are present, except those of the type $\langle a, a \rangle$. That is, $E : V \to V$ is complete, iff $E = V \times V - 1_V$, where 1_V is the identity (or *diagonal*) relation on V, namely, $\{\langle a, a \rangle : a \in V\}$ (cf. 1.2.0.20).

If the relation E is *symmetric*, that is, whenever aEb it is also true that bEa, then the graph is called *undirected*. One usually says "graph" implying *undirected graph* and says "digraph" otherwise. In the physical depiction, we draw

rather than

A digraph $C : A \to A$ is a *subgraph* of $E : V \to V$ iff $A \subseteq V$ and $C = E \cap (V \times V)$—that is, C uses some of the nodes of E and *all* the edges that E uses to interconnect said nodes.

A graph (*undirected!*) C is a *clique* in E iff it is a complete subgraph of E. The clique C is a *k-clique* if it has k nodes.

The clique problem, *CP, is the question "given a graph (undirected!) E and a number k; does E have a k-clique?"*

Before we discuss the reduction, let us settle that $CP \in \mathcal{NP}$. We need to establish that *given* a graph $E : V \to V$ and a number k, every instance where there is a positive answer to the question "does E have a k-clique?" can be verified as follows:

(I) We "correctly" choose (*nondeterministically*) a set of k nodes, A, from V; this is accomplished in $O(k)$ steps, and hence in linear time with respect to input length—the input being $\langle k, E \rangle$.

We then proceed deterministically to *verify* that

(II) These nodes define a complete subgraph of E (one checks that $A \times A - \mathbf{1}_A \subseteq E$); to do so we check $k^2 - k$ pairs of (distinct) nodes.

Pause. How is a graph (and a number) *given*?◄

We can code $\langle k, E \rangle$ "linearly" as a sequence $k, e_1, \ldots, e_r$ where the e_i are the edges—$\langle a, b \rangle$—of E. Of course, the edges implicitly include the node information (set V). Without loss of generality, since V is finite, we assume that $V \subseteq \mathbb{N}$.

A particularly efficient coding that does not involve exponentiation and can be easily decoded using string operations is the *Quine-Smullyan coding* [cf. Smullyan (1961), Tourlakis (1984)]: Given the alphabet $\Sigma = \{1, 2, 3, \ldots, m\}$, where $m > 1$. A sequence of strings (which can be viewed as integers written in m-adic notation) in Σ^+,

$$a_1, a_2, \ldots, a_s$$

can be coded as follows:

First, locate the maximum length string over the alphabet $\{1\}$—*tally* of ones—that occurs as a substring in some of the a_i. Say it is

$$x = \underbrace{11 \ldots 1}_{r \text{ ones}}$$

Then form the "glue"-string $g = 2 * x * 1 * 2$. Finally, code the sequence by gluing the a_i in order:

$$c = g * a_1 * g * a_2 * g * a_3 * g \cdots g * a_s * g$$

Decoding is easy: Find the maximum length tally of ones in c—this will be $x * 1$. Now the glue $g = 2 * x * 1 * 2$ is known, and the a_i can easily be recovered.

Let us now establish the reduction $CSAT \leq_p CP$, which proves that CP is $\mathcal{NP}$-hard and along with the preceding discussion settles its $\mathcal{NP}$-completeness.

We need to find a poly-time computable function f that converts the question $w \in CSAT$ to $f(w) \in CP$. So, we are given a Boolean CNF formula w:

$$\bigwedge_{i=1}^{t} (l_1^i \vee \cdots \vee l_{s_i}^i)$$

We build an undirected graph, $f(w)$, that will have a k-clique (for appropriate k) iff $w \in CSAT$. The graph has nodes arranged in t rows. Row i has nodes *labeled*

$$\left\langle i, l_1^i \right\rangle, \left\langle i, l_2^i \right\rangle, \ldots, \left\langle i, l_{s_i}^i \right\rangle$$

There are no horizontal edges, that is, edges connecting nodes that have the same first component, i, in their labels. An edge connects nodes $\left\langle i, l_a^i \right\rangle$ and $\left\langle j, l_b^j \right\rangle$, where $i \neq j$, iff the literals l_a^i and l_b^j are compatible, that is, one is not the negation of the other.

It is immediate that the constructed undirected graph has a t-clique iff at least one literal in each row can be assigned $\mathbf{t}$, without conflicts; and this is so iff w is satisfiable. On the other hand, it is also easy to see that the graph, $f(w)$, is constructible within poly-time with respect to $|w|$.   □

There is a huge library of known $\mathcal{NP}$-complete problems. A good start point for the reader who wants to explore it is Papadimitriou (1994).

5.2   AXT, LOOP PROGRAM, AND GRZEGORCZYK HIERARCHIES

Computable functions can have some quite complex definitions. For example, a loop programmable function might be given via a loop program that has depth of nesting of the loop-end pair, say, equal to 200. Now this *is* complex! Or a function might be given via an arbitrarily complex sequence of primitive recursions, with the restriction that the computed function is *majorized* by some known function, for all values of the input (for the concept of majorization see Subsection 2.4.3).

But does such *definitional*—and therefore, "static"—complexity have any bearing on the *computational*—dynamic—complexity of the function? We will see that it does, and we will connect definitional and computational complexities quantitatively.

Our study will be restricted to the class $\mathcal{PR}$ that we will subdivide into an infinite sequence of increasingly more inclusive subclasses, S_i. A so-called *hierarchy* of classes of functions.

5.2.0.4 Definition. A sequence $(S_i)_{i \geq 0}$ of subsets of $\mathcal{PR}$ is a *primitive recursive hierarchy* provided all of the following hold:
   (1) $S_i \subseteq S_{i+1}$, for all $i \geq 0$
   (2) $\mathcal{PR} = \bigcup_{i \geq 0} S_i$.

The hierarchy is *proper* or *nontrivial* iff $S_i \neq S_{i+1}$, for all but finitely many i.

If $f \in S_i$ then we say that its *level in the hierarchy* is $\leq i$. If $f \in S_{i+1} - S_i$, then its level is equal to $i + 1$.   □

The first hierarchy that we will define is due to Axt and Heinermann [Heinermann (1961) and Axt (1965)].

5.2.0.5 Definition. (The Axt-Heinermann Hierarchy) We define the class $\mathcal{K}_n$ for each $n \geq 0$ by recursion on n. We let $\mathcal{K}_0$ stand for the closure of $\{\lambda x.x, \lambda x.x + 1\}$ under substitution (2.1.2.6).

For $n \geq 0$, $\mathcal{K}_{n+1}$ is the closure under substitution of $\mathcal{K}_n \cup \{prim(h, g) : h \in \mathcal{K}_n \wedge g \in \mathcal{K}_n\}$, where $prim(h, g)$ is the function defined by primitive recursion from the basis function h and the iterator function g (cf. 2.1.1.14). $\square$

Thus, primitive recursion is the "expensive" operation, an application of which takes us out of a given $\mathcal{K}_n$. On the other hand, as the classes are defined (the $n + 1$ case), it follows that any finite number of substitution operations keeps us in the same class; all $\mathcal{K}_n$, that is, are closed under substitution.

We list a number of straightforward properties.

5.2.0.6 Proposition. $(\mathcal{K}_n)_{n \geq 0}$ *is a hierarchy, that is,*

(1) $\mathcal{K}_n \subseteq \mathcal{K}_{n+1}$, *for* $n \geq 0$,

   and

(2) $\mathcal{PR} = \bigcup_{i \geq 0} \mathcal{K}_i$.

Proof.

(1) Immediate from the definition of $\mathcal{K}_{n+1}$ in 5.2.0.5.

(2) This is straightforward, from 5.2.0.5 and 2.1.2.3. The part $\supseteq$ is rather trivial, while the $\subseteq$ part can be done by induction on $\mathcal{PR}$. $\square$

5.2.0.7 Proposition. $\lambda x.A_n(x) \in \mathcal{K}_n$, *for all* $n \geq 0$, *where* $\lambda n x.A_n(x)$ *is the Ackermann function of Subsection 2.4.1.*

Proof. Induction on n. For $n = 0$, we note that $A_0 = \lambda x.x + 2 \in \mathcal{K}_0$. By 5.2.0.5 and 2.4.1.2, if $\lambda x.A_n(x) \in \mathcal{K}_n$, then $\lambda x.A_{n+1}(x) \in \mathcal{K}_{n+1}$—since $\lambda x.2 \in \mathcal{K}_0$ by substitution, and $\mathcal{K}_0 \subseteq \mathcal{K}_n$— and this concludes the induction. $\square$

5.2.0.8 Proposition. *For every* $f \in \mathcal{K}_n$ *there is a* $k \in \mathbb{N}$ *such that* $f(\vec{x}) \leq A_n^k(\max(\vec{x}))$, *for all* $\vec{x}$.

Proof. In 2.4.3.1 we proved that the Ackermann function majorizes every primitive recursive function. The induction proof over $\mathcal{PR}$ demonstrated that composing finitely many functions f_i—each majorized by $A_n^{k_i}$ using the same fixed n—produces a function that is majorized by $A_n^{\sum_i k_i}$. Thus, in the present context, and to settle the proposition by induction on n, we will only need to show that every initial function of $\mathcal{K}_0$ is majorized by some A_0^r and each initial function of $\mathcal{K}_{n+1}$, namely,

$$f \in \mathcal{K}_n \cup \{prim(h, g) : h \in \mathcal{K}_n \wedge g \in \mathcal{K}_n\} \tag{1}$$

is majorized by some appropriate A_n^r.

Well, each of x and $x+1$ are less than $x+2 = A_0(x)$ and this settles the basis. Assume the claim (I.H.) for $\mathcal{K}_n$—fixed $n \geq 0$—and tackle that for $\mathcal{K}_{n+1}$. By our plan, we need to show the initial function are majorized by some A_{n+1}^r. For those $f \in \mathcal{K}_n$ [cf. (1)] this is the result of Lemmata 2.4.2.7 and 2.4.2.10. If $f = prim(h, g)$, then, by the I.H. on n, we have, for all x, z and $\vec{y}$,

$$h(\vec{y}) \leq A_n^{r_1}\big(\max(\vec{y})\big)$$

and

$$g(x, \vec{y}, z) \leq A_n^{r_2}\big(\max(x, \vec{y}, z)\big)$$

By 2.4.3.1 we have some r such that $f(x, \vec{y}) \leq A_{n+1}^r\big(\max(x, \vec{y})\big)$, for all x and $\vec{y}$. □

5.2.0.9 Corollary. *The Axt-Heinermann hierarchy is proper.*

Proof. Indeed, $\lambda x.A_{n+1} \in \mathcal{K}_{n+1} - \mathcal{K}_n$, for all $n \geq 0$. By 5.2.0.7, we only need to see that $\lambda x.A_{n+1} \notin \mathcal{K}_n$. Indeed, otherwise, we would have, for all x, and some r, $A_{n+1}(x) \leq A_n^r(x)$ (cf. 2.4.2.10). □

We can also base the definition of classes similar to $\mathcal{K}_n$ on simultaneous recursion:

5.2.0.10 Definition. We define the class $\mathcal{K}_n^{sim}$ for each $n \geq 0$ by recursion on n. We let $\mathcal{K}_0^{sim} = \mathcal{K}_0$.

For $n \geq 0$, $\mathcal{K}_{n+1}^{sim}$ is the closure under substitution of $\mathcal{K}_n^{sim} \cup \{f : f \text{ is obtained}$ by simultaneous primitive recursion from functions in $\mathcal{K}_n^{sim}\}$. □

The following are straightforward (see Exercises 5.3.15, 5.3.16).

5.2.0.11 Proposition. *For $n \geq 0$, we have $\mathcal{K}_n \subseteq \mathcal{K}_n^{sim}$.*

Thus, $\mathcal{PR} = \bigcup_{n \geq 0} \mathcal{K}_n \subseteq \bigcup_{n \geq 0} \mathcal{K}_n^{sim} \subseteq \mathcal{PR}$.

Thus, by 5.2.0.7,

5.2.0.12 Corollary. *For $n \geq 0$, we have $\lambda x.A_n(x) \in \mathcal{K}_n^{sim}$.*

5.2.0.13 Proposition. *For every $f \in \mathcal{K}_n^{sim}$ there is a $k \in \mathbb{N}$ such that $f(\vec{x}) \leq A_n^k\big(\max(\vec{x})\big)$, for all $\vec{x}$.*

Proof. A straightforward modification of the proof of 5.2.0.8. □

5.2.0.14 Corollary. *The $(\mathcal{K}_n^{sim})_{n \geq 0}$ hierarchy is proper.*

Proof. Exactly as in the proof of 5.2.0.9. □

A closely related hierarchy—that is once again defined in terms of how complex a function's definition is—is based on loop programs [Ritchie (1965)].

5.2.0.15 Definition. (A Hierarchy of Loop Programs) Cf. Section 2.2. We denote by L_0 the class of all loop programs that do not employ the **Loop-end** instruction pair.

Assuming that L_n has been defined, then L_{n+1} is the set of programs that is *the closure under program concatenation* of this initial set:

$$L_n \cup \Big\{ \mathbf{Loop}\, X; P; \mathbf{end} : \text{for any variable } X \text{ and } P \in L_n \Big\} \qquad \square$$

Trivially, $L_n \subseteq L_{n+1}$ and the maximum nesting depth of the **Loop-end** pair increases by one as we pass from L_n to L_{n+1}. Of course, by virtue of $L_n \subseteq L_{n+1}$, not every $P \in L_{n+1}$ nests the **Loop-end** pair as deep as $n + 1$. Thus, $R \in L_n$ iff the depth of nesting of the **Loop-end** instruction pair is at most n. Nesting depth equal to 0 means the absence of a **Loop-end** instruction pair.

The following is immediate.

5.2.0.16 Proposition. *Cf. 2.2.0.8.* $(L_n)_{n \geq 0}$ *is a proper L-hierarchy. That is,*

(1) $L_n \subset L_{n+1}$, *for* $n \geq 0$

    and

(2) $L = \bigcup_{n \geq 0} L_n$

We are more interested in the induced (by the L_n sets) hierarchy of primitive recursive classes:

5.2.0.17 Definition. Cf. 2.2.1.2. We denote by $\mathscr{L}_n$, for $n \geq 0$, the class

$$\{ P^{\vec{x}_r}_{x_k} : P \in L_n \wedge \text{ the } \vec{x}_r \text{ and } x_k \text{ occur in } P \} \qquad \square$$

5.2.0.18 Proposition. *For* $n \geq 0$, *we have that* $\mathcal{K}_n^{sim} = \mathscr{L}_n$.

Proof. In outline, the instruction pair **Loop-end** implements one simultaneous recursion. On the other hand, by the definition of $\mathcal{K}_n^{sim}$, this class contains functions obtained from those of $\mathcal{K}_0^{sim} = \mathcal{K}_0$ by n nested simultaneous recursions (and possibly some substitutions).

In detail, one can do induction on n and imitate the proofs of 2.2.1.4 and 2.2.1.3. See Exercise 5.3.17. $\qquad \square$

Thus, everything we said about the $(\mathcal{K}_n^{sim})_{n \geq 0}$ hierarchy carries over to the $(\mathscr{L}_n)_{n \geq 0}$ hierarchy—after all, it is the same hierarchy under two different definitions. In particular, by 5.2.0.14,

5.2.0.19 Proposition. *The* $\mathcal{PR}$- *(or* $\mathscr{L}$-*)hierarchy,* $(\mathscr{L}_n)_{n \geq 0}$, *is proper.*

5.2.0.20 Example. Here are some functions and predicates in the "lower" (small n) classes of the $(\mathcal{K}_n^{sim})_{n \geq 0}$ hierarchy.

The following are in $\mathcal{K}_1$ and hence in $\mathcal{K}_1^{sim} = \mathscr{L}_1$.

(1) $\lambda xy.x + y$. Indeed,

$$0 + y = y$$
$$(x + 1) + y = (x + y) + 1$$

and $\lambda y.y$ and $\lambda z.z + 1$ are in $\mathcal{K}_0 = \mathcal{K}_0^{sim}$.

(2) $\lambda xy.x(1 \dot{-} y)$. Indeed,

$$x(1 \dot{-} 0) = x$$
$$x(1 \dot{-} (y + 1)) = 0$$

and $\lambda y.y$ and $\lambda z.0$ are in $\mathcal{K}_0 = \mathcal{K}_0^{sim}$.

(3) $\lambda x.1 \dot{-} x$. By substitution operations from the previous function.

(4) $\lambda x.x \dot{-} 1$. Indeed,

$$0 \dot{-} 1 = 0$$
$$(x + 1) \dot{-} 1 = x$$

and $\lambda y.y$ and $\lambda z.0$ are in $\mathcal{K}_0 = \mathcal{K}_0^{sim}$.

(5) $\lambda x. \lfloor x/2 \rfloor \in \mathcal{K}_1^{sim}$. Indeed, see 2.1.3.3.

This example shows that $\mathcal{K}_1 \neq \mathcal{K}_1^{sim}$, since $\lambda x. \lfloor x/2 \rfloor \notin \mathcal{K}_1$ as follows from results of Ritchie (1965) and Tsichritzis (1970) that were retold in Tourlakis (1984).

(6) $switch = \lambda xyz.$if $x = 0$ then y else z. Indeed, we have the recursion

$$switch(0, y, z) = y$$
$$switch(x + 1, y, z) = z$$

where $\lambda y.y$ is in $\mathcal{K}_0 = \mathcal{K}_0^{sim}$.

The following are in $\mathcal{K}_2$ and hence in $\mathcal{K}_2^{sim} = \mathscr{L}_2$.

(a) $\lambda xy.x \dot{-} y$. Indeed,

$$x \dot{-} 0 = x$$
$$x \dot{-} (y + 1) = (x \dot{-} y) \dot{-} 1$$

and $\lambda y.y$ and $\lambda z.z \dot{-} 1$ are in $\mathcal{K}_1 \subseteq \mathcal{K}_1^{sim}$.

(b) $\lambda xy.xy$. Indeed,

$$x0 = 0$$
$$x(y + 1) = xy + x$$

and $\lambda y.0$ and $\lambda wz.w + z$ are in $\mathcal{K}_1 \subseteq \mathcal{K}_1^{sim}$.

(c) $\lambda x.2^x$. Indeed,

$$2^0 = 1$$
$$2^{y+1} = 2^y + 2^y$$

and $\lambda y.1$ and $\lambda wz.w + z$ are in $\mathcal{K}_1 \subseteq \mathcal{K}_1^{sim}$. $\qquad\square$

5.2.0.21 Definition. As is usual, the predicate classes $\mathcal{K}_{n,*}$ and $\mathcal{K}_{n,*}^{sim}$—the latter being the same as $\mathscr{L}_{n,*}$—are defined for all $n \geq 0$ as $\{f(\vec{x}) = 0 : f \in \mathcal{K}_n\}$ and $\{f(\vec{x}) = 0 : f \in \mathcal{K}_n^{sim}\}$, respectively. $\qquad\square$

5.2.0.22 Proposition. *For $n \geq 1$, we have that $\mathcal{K}_{n,*}$ and $\mathcal{K}_{n,*}^{sim}$ are closed under $\neg$ and $\vee$—and hence under $\wedge$, $\rightarrow$, and $\equiv$ as well.*

Proof. Let $Q(\vec{x}) \in \mathcal{K}_{n,*}$. Then, for some $q \in \mathcal{K}_n$, $Q(\vec{x}) \equiv q(\vec{x}) = 0$. Since $r = \lambda \vec{x}.1 \doteq q(\vec{x}) \in \mathcal{K}_n$ if $n \geq 1$ by 5.2.0.20, we are done, noting $\neg Q(\vec{x}) \equiv r(\vec{x}) = 0$. Next, let also $S(\vec{y}) \equiv s(\vec{y}) = 0$ with $s \in \mathcal{K}_n$. Then $Q(\vec{x}) \vee S(\vec{y}) \equiv switch(q(\vec{x}), 0, r(\vec{y})) = 0$; but $switch \in \mathcal{K}_n$, for $n \geq 1$ (cf. 5.2.0.20).

The cases for $\mathcal{K}_{n,*}^{sim}$ are argued identically with the preceding two. $\qquad\square$

5.2.0.23 Corollary. *The relations $\lambda x.x \leq a$, $\lambda x.x < a$ and $\lambda x.x = a$ are in $\mathcal{K}_{1,*}$ and hence in $\mathcal{K}_{1,*}^{sim}$.*

Proof. By 5.2.0.20(4) and substitution, we have that $\lambda x.x \doteq a \in \mathcal{K}_1$. But $x \leq a \equiv x \doteq a = 0$. On the other hand, $x < a \equiv x + 1 \doteq a = 0$. Thus the claim about $\lambda x.x < a$ is true. Noting that $\lambda x.a \leq x$ is in $\mathcal{K}_{1,*}$ due to

$$a \leq x \equiv \neg x < a$$

and 5.2.0.22, we have that $\lambda x.x = a$ is in $\mathcal{K}_{1,*}$ by 5.2.0.22 and the observation $x = a \equiv x \leq a \wedge a \leq x$. $\qquad\square$

5.2.0.24 Proposition. *For $n \geq 1$, we have that $\mathcal{K}_n$ and $\mathcal{K}_n^{sim}$ are closed under definition by cases.*

Proof. This is immediate from either of the suggested proofs for 2.1.2.37, noting 5.2.0.20, (1), (2) and (6). $\qquad\square$

The three hierarchies that we introduced include increasingly complex classes, using as a yardstick of complexity the nesting depth of primitive recursion. The next hierarchy, due to Grzegorczyk (1953), gauges *complexity of definition* by the (numerical) size of the function it produces—and, correspondingly, the class complexity at level n by the size of the functions it contains. As the definition does *not necessarily* force a function such as $prim(h, g)$ to exit from a given level, the Grzegorczyk hierarchy is much more amenable to mathematical analysis.

5.2.0.25 Definition. (The Grzegorczyk Hierarchy) We are given a fixed sequence of functions, $(g_n)_{n \geq 0}$ by

$$g_0 = \lambda x . x + 1$$
$$g_1 = \lambda x y . x + y$$
$$g_2 = \lambda x y . x y$$

and, for $n \geq 2$,

$$g_{n+1} = \lambda x y . A_n \big(\max(x, y) \big)$$

where $\lambda n y . A_n(x)$ is the Ackermann function of Subsection 2.4.1.

The hierarchy $(\mathcal{E}^n)_{n \geq 0}$ is defined as follows: $\mathcal{E}^n$ is the closure of

$$\{ \lambda x . x + 1, \lambda x . x, g_n \}$$

under *substitution* and *bounded primitive recursion*, the latter being the schema below

$$f(0, \vec{y}) = h(\vec{y})$$
$$f(x + 1, \vec{y}) = q \big(x, \vec{y}, f(x, \vec{y}) \big)$$
$$f(x, \vec{y}) \leq B(x, \vec{y})$$

where h, q and B are given functions.   □

A class $\mathcal{C}$ is closed under bounded primitive recursion iff whenever h, q, and B are in $\mathcal{C}$, then so is the f produced as above.

We note that the bounded recursion is *not on notation*. Rather, it is an ordinary number-theoretic primitive recursion along with a condition that the function f has actually been "produced" *only if* its values are bounded everywhere by those of the *given B*.

The g_n-function included among the initial functions at each level, which gauges the (numerical) size of functions included in each $\mathcal{E}^n$ is (a version of) the Ackermann function. Grzegorczyk used a different version than we do here. Our choice to use the function of Subsection 2.4.1 was partly dictated by ease-of-use considerations, but mostly because we know quite a bit about the A_n already. The reader may consult Tourlakis (1984) to read a proof that the version we use here produces the same $\mathcal{E}^n$ classes as in Grzegorczyk (1953).

The class of relations at level n of the Grzegorczyk hierarchy is defined as usual.

5.2.0.26 Definition. $\mathcal{E}^n_*$, for $n \geq 0$, denotes the class of relations $\{ f(\vec{x}) = 0 : f \in \mathcal{E}^n \}$.   □

5.2.0.27 Example. Here are some examples of functions and relations in $\mathcal{E}^0$ and $\mathcal{E}^0_*$:

(1) $\lambda x y . x (1 \mathbin{\dot{-}} y)$.

$$\begin{cases} x(1 \mathbin{\dot{-}} 0) = x \\ x(1 \mathbin{\dot{-}} (y + 1)) = 0 \\ x(1 \mathbin{\dot{-}} y) \leq x \end{cases}$$

(2) $\lambda x.1 \dotminus x$. By (1) and substitution.

(3) $\lambda x.x \dotminus 1$.

$$\begin{cases} 0 \dotminus 1 = 0 \\ (x+1) \dotminus 1 = x \\ x \dotminus 1 \leq x \end{cases}$$

(4) $\lambda xy.x \dotminus y$.

$$\begin{cases} x \dotminus 0 = x \\ x \dotminus (y+1) = (x \dotminus y) \dotminus 1 \\ x \dotminus y \leq x \end{cases}$$

(5) $\lambda xy.x \leq y$ and $\lambda xy.x < y$ are in $\mathcal{E}_*^0$. Indeed, $x \leq y \equiv x \dotminus y = 0$ and $x < y \equiv (x+1) \dotminus y = 0$. $\square$

5.2.0.28 Lemma. *For all $n \geq 0$, $\mathcal{E}^0 \subseteq \mathcal{E}^n$.*

Proof. $\mathcal{E}^n$ contains the initial functions of $\mathcal{E}^0$ and is closed under the same operations.

$\square$

5.2.0.29 Theorem. *For $n \geq 0$, $\mathcal{E}_*^n$ is closed under Boolean operations and also under bounded quantification, namely, $(\exists y)_{<z}$, $(\exists y)_{\leq z}$, $(\forall y)_{<z}$, $(\forall y)_{\leq z}$.*

Proof. We implicitly use 5.2.0.28. For Boolean operations it suffices to consider $\neg$ and $\vee$ only. So, let $R(\vec{x}) \equiv r(\vec{x}) = 0$ and $Q(\vec{y}) \equiv q(\vec{y}) = 0$, where r and q are in $\mathcal{E}^n$. Now, $\neg R(\vec{x}) \equiv 1 \dotminus r(\vec{x}) = 0$ and we are done by 5.2.0.27(2). On the other hand, $R(\vec{x}) \vee Q(\vec{y}) \equiv r(\vec{x})\big(1 \dotminus (1 \dotminus q(\vec{y}))\big) = 0$ and we are done by 5.2.0.27(1).

For closure under bounded quantification, let $P(y, \vec{x}) \equiv p(y, \vec{x}) = 0$, where $p \in \mathcal{E}^n$. Let $\chi_\exists$ be the characteristic function (cf. 2.1.2.15) of $(\exists y)_{<z} P(y, \vec{x})$. Noting that

$$(\exists y)_{<0} P(y, \vec{x}) \text{ is false, and } (\exists y)_{<z+1} P(y, \vec{x}) \equiv P(z, \vec{x}) \vee (\exists y)_{<z} P(y, \vec{x})$$

we have that $\chi_\exists$ satisfies the bounded recursion below:

$$\begin{cases} \chi_\exists(0, \vec{x}) = 1 \\ \chi_\exists(z+1, \vec{x}) = \chi_\exists(z, \vec{x})\Big(1 \dotminus \big(1 \dotminus p(z, \vec{x})\big)\Big) \\ \chi_\exists(z, \vec{x}) \leq 1 \end{cases}$$

and we are done. The "1" in the inequality above is the output of $\lambda x.1$ which is in $\mathcal{E}^0$. Clearly $\chi_\exists$ belongs where p does, and $(\exists y)_{<z} P(y, \vec{x}) \equiv \chi_\exists(z, \vec{x}) = 0$.

To conclude the proof for the remaining cases of quantification, note that $(\exists y)_{\leq z} R \equiv R \vee (\exists y)_{<z} R$; moreover, the universal quantifier cases follow from the closure of $\mathcal{E}_*^n$ under negation. $\square$

Note that if a class $\mathcal{C}$ contains $\lambda x.1 \doteq x$ and is closed under substitution, then 2.1.2.16 applies to its class of predicates $\mathcal{C}_*$.

The following result is, modulo choice of Ackermann function, from Grzegorczyk (1953).

5.2.0.30 Lemma. (Bounding Lemma) (1) *For each $f \in \mathcal{E}^0$, there are i and k such that $f(\vec{x}) \leq x_i + k$ everywhere.*

(2) *For each $f \in \mathcal{E}^1$, there are C and k such that $f(\vec{x}) \leq C \max(\vec{x}) + k$ everywhere.*

(3) *For each $f \in \mathcal{E}^2$, there are C, n, and k such that $f(\vec{x}) \leq C \max(\vec{x})^n + k$ everywhere.*

(4) *For each $f \in \mathcal{E}^{n+1}$, $n \geq 2$, there is a k such that $f(\vec{x}) \leq A_n^k\big(\max(\vec{x})\big)$ everywhere.*

Proof.

All proofs are by induction over the appropriate $\mathcal{E}^n$.

(1) The claim trivially holds for the initial functions and propagates with bounded recursion since the I.H. applies to whichever bounding function B was employed. Let then f be obtained by substitution,

$$f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n, \vec{y}) = g(x_1, \ldots, \underset{\underset{h(\vec{y})}{\uparrow}}{x_i}, \ldots, x_n)$$

The I.H. applies to g. Now, if $g(\vec{x}) \leq x_j + k$ everywhere, for $j \neq i$, then also

$$f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n, \vec{y}) \leq x_j + k, \text{ everywhere.}$$

If, on the other hand, $j = i$, then $f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n, \vec{y}) \leq h(\vec{y}) + k$, everywhere. By the I.H. on h, $f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n, \vec{y}) \leq y_m + k + k'$, everywhere, for some m and k'.

The other cases of substitution are as easy.

(2) The basis and the propagation of the claim with bounded recursion are as above [note, incidentally, that $x + y \leq 2\max(x, y)$]. Let us now look at a substitution $h(\vec{y}, g(\vec{x}), \vec{z})$. We have, by the I.H. applied to h,

$$h(\vec{y}, g(\vec{x}), \vec{z}) \leq C \max(\vec{y}, g(\vec{x}), \vec{z}) + k$$
$$\overset{\text{I.H. for } g}{\leq} C \max(\vec{y}, C' \max(\vec{x}) + k', \vec{z}) + k$$
$$\leq CC' \max(\vec{y}, \vec{x}, \vec{z}) + Ck' + k$$

(3) Left as an exercise (see Exercise 5.3.20).

(4) The claim is true for the initial functions and propagates with bounded recursion for the reason named earlier. As for substitution, we know that the subscript n will not change (cf. 2.4.3.1) and thus if $A_n^{k_i}$ majorize the component-functions of the substitution, then $A_n^{\sum k_i}$ majorizes the result.   □

We can now prove that $\mathcal{E}^n \subset \mathcal{E}^{n+1}$ for all n.

5.2.0.31 Theorem. *$(\mathcal{E}^n)_{n \geq 0}$ is a proper primitive recursive hierarchy.*

Proof. First, $\mathcal{E}^n \subseteq \mathcal{E}^{n+1}$, for all n, since every bounded recursion in $\mathcal{E}^n$ can use as bounding functions the bounds $x_i + k$, $C \max(\vec{x}) + k$, and $C \max(\vec{x})^n + k$ if $n = 0, 1, 2$, respectively—cf. (1)–(3) in 5.2.0.30—or A_{n-1}^k if $n \geq 3$.

I am implying an induction over $\mathcal{E}^n$ in the above remark. But is $A_{n-1} \in \mathcal{E}^{n+1}$? Yes, if we assume that A_{n-2} is. See Exercise 5.3.21.

Reverting to the unified notation "g_n" and noting that $g_{n+1} \in \mathcal{E}^{n+1} - \mathcal{E}^n$ by 5.2.0.30, we promote $\subseteq$ above to $\subset$:

$$\mathcal{E}^n \subset \mathcal{E}^{n+1}, \text{ for all } n.$$

Now, trivially, $\mathcal{E}^n \subseteq \mathcal{PR}$, for all n. On the other hand, by 2.4.3.1, every primitive recursion is a bounded recursion with bounding function A_n^k for some k, so $\mathcal{PR} \subseteq \bigcup_{n \geq 0} \mathcal{E}^n$ as well.   □

5.2.0.32 Exercise. In view of 5.2.0.30, prove that *switch* (the "full" if-then-else) and max are *not* in $\mathcal{E}^0$.   □

We defined bounded summation and multiplication in 2.1.2.30 and saw that, as operations, they do not take us out of $\mathcal{PR}$. More interesting is this:

5.2.0.33 Proposition. *For $n \geq 2$, $\mathcal{E}^n$ is closed under bounded summation.*

Proof. By reference to 2.1.2.30, we only need a bounding function for $\sum_{i < z} f(i, \vec{x})$ in $\mathcal{E}^n$.

For $n = 2$, $f(i, \vec{x}) = O(\max(i, \vec{x})^r)$, for some r, due to 5.2.0.30. But then,

$$\sum_{i < z} f(i, \vec{x}) = \sum_{i < z} O(\max(i, \vec{x})^r) = O(z \max(z, \vec{x})^r)$$

Since, for any constants C and D, $\lambda z \vec{x}.Cz \max(z, \vec{x})^r + D$ is in $\mathcal{E}^2$, our bounding function is obtained by choosing the right C and D.

For $n > 2$, let, by 5.2.0.30, r be such that $f(i, \vec{x}) \leq A_{n-1}^r(\max(i, \vec{x}))$, for all $i, \vec{x}$. Then

$$\sum_{i < z} f(i, \vec{x}) \leq \sum_{i < z} A_{n-1}^r \big(\max(i, \vec{x}) \big) \leq z A_{n-1}^r \big(\max(z, \vec{x}) \big) \tag{1}$$

But $\lambda xy.xy$ and $\lambda z\vec{x}.A_{n-1}^k \big(\max(z, \vec{x}) \big)$ are in $\mathcal{E}^n$ for $n > 2$. We have obtained the required bounding function in (1).   □

5.2.0.34 Proposition. *For $n \geq 3$, $\mathcal{E}^n$ is closed under bounded multiplication.*

Proof. We proceed as in the proof of 5.2.0.33 above.

Let, by 5.2.0.30, r be such that $f(i, \vec{x}) \leq A_{n-1}^r(\max(i, \vec{x}))$, for all $i, \vec{x}$. Then

$$\prod_{i < z} f(i, \vec{x}) \leq \prod_{i < z} A_{n-1}^r\big(\max(i, \vec{x})\big) \leq \Big(A_{n-1}^r\big(\max(z, \vec{x})\big)\Big)^z \qquad (2)$$

But $\lambda xy.x^y$ and $\lambda z\vec{x}.A_{n-1}^k\big(\max(z, \vec{x})\big)$ are in $\mathcal{E}^n$, for $n \geq 3$. We obtained the required bounding function in (2). $\qquad \square$

A definition of *bounded search* that is used in Grzegorczyk (1953) [cf. also Péter (1967)] is the following:

5.2.0.35 Definition. (Alternative Bounded Search) For any total number-theoretic function $\lambda y\vec{x}.f(y, \vec{x})$ we define

$$(\overset{\circ}{\mu}y)_{<z} f(y, \vec{x}) \overset{Def}{=} \begin{cases} \min\{y : y < z \wedge f(y, \vec{x}) = 0\} & \text{if } (\exists y)_{<z} f(y, \vec{x}) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$(\overset{\circ}{\mu}y)_{\leq z} f(y, \vec{x})$ means $(\overset{\circ}{\mu}y)_{<z+1} f(y, \vec{x})$, and $(\overset{\circ}{\mu}y)_{<z} R(y, \vec{x})$ means $(\overset{\circ}{\mu}y)_{<z}\chi_R(y, \vec{x})$, where χ_R is the characteristic function of R. $\qquad \square$

5.2.0.36 Theorem. *For $n \geq 0$, $\mathcal{E}^n$ is closed under $(\overset{\circ}{\mu}y)_{<z}$.*

Proof. Let $f \in \mathcal{E}^n$. We set $g(z, \vec{x}) = (\overset{\circ}{\mu}y)_{<z} f(y, \vec{x})$. Notice that

$$\begin{cases} g(0, \vec{x}) & = 0 \\ g(z + 1, \vec{x}) & = \text{if } (\forall y)_{<z} f(y, \vec{x}) \neq 0 \wedge f(z, \vec{x}) = 0 \text{ then } z \\ & \quad \text{else } g(z, \vec{x}) \\ g(z, \vec{x}) & \leq z \end{cases}$$

The above bounded recursion works for $n \geq 1$, but will not work for $n = 0$ due to 5.2.0.32; some acrobatics will be necessary:

We note that the right hand side of the second equation is obtained by substituting $g(z, \vec{x})$ into w and $\chi(z, \vec{x})$—the value at $\langle z, \vec{x}\rangle$ of the characteristic function of $(\forall y)_{<z} f(y, \vec{x}) \neq 0 \wedge f(z, \vec{x}) = 0$—into x in

$$\text{if } x = 0 \text{ then } z \text{ else } w$$

Noting that $g(z, \vec{x}) \leq z$, we can replace the troublesome full if-then-else by the expression $sw(x, z, w)$ given by

$$\text{if } x = 0 \text{ then } z$$
$$\text{else if } w \leq z \text{ then } w \text{ else } 0$$

Note that $sw \in \mathcal{E}^0$ because of the bounded recursion below.

$$\begin{cases} sw(0, z, w) = z \\ sw(x + 1, z, w) = \text{if } w \dot{-} z = 0 \text{ then } w \text{ else } 0 \\ sw(x, z, w) \leq z \end{cases}$$

Moreover, note that $g(z + 1, \vec{x}) = sw(\chi(z, \vec{x}), z, g(z, \vec{x}))$. $\square$

The absence of the full switch from $\mathcal{E}^0$ qualifies the result about closure under definition by cases:

5.2.0.37 Corollary. *For $n \geq 1$, $\mathcal{E}^n$ is closed under definition by cases 2.1.2.37.*

$\mathcal{E}^0$ is closed under definition by cases provided the produced function f satisfies $f(\vec{x}) \leq x_i + k$ everywhere, for some i and k.

Proof. For $n \geq 1$ the proof of 2.1.2.37 works. For $\mathcal{E}^0$, if f is given as in 2.1.2.37, where the f_i are in $\mathcal{E}^0$ and the R_i in $\mathcal{E}^0_*$, then

$$f(\vec{x}) = (\overset{\circ}{\mu} y)_{\leq x_i + k} \Big(y = f_1(\vec{x}) \wedge R_1(\vec{x}) \vee \ldots \vee y = f_{n+1}(\vec{x}) \wedge R_{n+1}(\vec{x}) \Big) \quad (1)$$

where we wrote R_{n+1} for the "otherwise" relation. The reader should carefully identify all the results that we proved so far about the Grzegorczyk classes that make (1) work. $\square$

5.2.0.38 Theorem. *$\mathcal{E}^2$ is closed under* simultaneous bounded recursion, *that is, under the schema of Subsection 2.1.3, where, additionally, k bounding functions B_i, for $i = 1, \ldots, k$, are given, and the functions f_i resulting from the schema* must *satisfy $f_i(x, \vec{y}) \leq B_i(x, \vec{y})$ everywhere.*

Proof. Consider the schema below, where the h_i, g_i and B_i are in $\mathcal{E}^2$.

$$\begin{cases} f_1(0, \vec{y}_n) & = h_1(\vec{y}_n) \\ \vdots \\ f_k(0, \vec{y}_n) & = h_k(\vec{y}_n) \\ f_1(x + 1, \vec{y}_n) & = g_1(x, \vec{y}_n, f_1(x, \vec{y}_n), \ldots, f_k(x, \vec{y}_n)) \\ \vdots \\ f_k(x + 1, \vec{y}_n) & = g_k(x, \vec{y}_n, f_1(x, \vec{y}_n), \ldots, f_k(x, \vec{y}_n)) \\ f_1(x, \vec{y}_n) & \leq B_1(x, \vec{y}_n) \\ \vdots \\ f_k(x, \vec{y}_n) & \leq B_k(x, \vec{y}_n) \end{cases} \quad (1)$$

The pairing function $J = \lambda xy.(x + y)^2 + x$ (cf. 2.1.4.8) is in $\mathcal{E}^2$, and so are its projections $K = \lambda z.(\overset{\circ}{\mu} x)_{\leq z} (\exists y)_{\leq z} J(x, y) = z$ and $L = \lambda z.(\overset{\circ}{\mu} y)_{\leq z} (\exists x)_{\leq z} J(x, y) = z$.

Thus, we have the coding-decoding scheme—$\lambda \vec{z}_k. [\![z_1, \ldots, z_k]\!]^{(k)}$ and Π_i^k—of 2.1.4.9 in $\mathcal{E}^2$.

The proof of 2.1.3.1—that shows how to simulate a simultaneous recursion by a single recursion—goes through unchanged if we replace the used there prime power coding/decoding by the alternative $[\![\ldots]\!]/\Pi_i^k$ adopted here. Noting that

$$[\![f_1(x, \vec{y}_n), \ldots, f_k(x, \vec{y}_n)]\!]^{(k)} \leq [\![B_1(x, \vec{y}_n), \ldots, B_k(x, \vec{y}_n)]\!]^{(k)}$$

and that the right hand side of the above $\leq$ is in $\mathcal{E}^2$ (as a function of $x, \vec{y}_n$) by substitution, we obtain that

$$\lambda x \vec{y}_n. [\![f_1(x, \vec{y}_n), \ldots, f_k(x, \vec{y}_n)]\!]^{(k)} \in \mathcal{E}^2$$

and therefore, for $i = 1, \ldots, k$, $f_i = \lambda x \vec{y}_n. \Pi_i^k \big([\![f_1(x, \vec{y}_n), \ldots, f_k(x, \vec{y}_n)]\!]^{(k)} \big)$ is in $\mathcal{E}^2$. $\qquad\square$

5.2.0.39 Corollary. $\mathcal{E}^n$, *for* $n \geq 2$, *is closed under simultaneous bounded recursion.*

We have introduced four primitive recursive hierarchies—of Axt-Hienermann, Dennis Ritchie, and Grzegorczyk—the yardstick of "complexity" of a class at each level n being that of its *definition*, whether the measure was *numerical size* of produced functions (Grzegorczyk) or *nesting depth* of primitive recursion (in all the others).

We conclude this subsection by showing that this *definitional complexity* tracks very accurately the *computational complexity* of the primitive recursive functions. *The URM formalism will be the computing model to which the computational complexity will related.*

The "main lemma" toward connecting the four hierarchies to each other on one hand, and with the computational complexity of their functions on the other, will be the *Ritchie*[154]*-Cobham property* of the Grzegorczyk classes, that

for $n \geq 0$, $f \in \mathcal{E}^n$ iff f is computable by some URM within time $t \in \mathcal{E}^n$ (RC)

We will need a *simulation tool*, namely, we will show that the *computation* of a URM can be simulated by a very simple simultaneous primitive recursion. The reader is referred to 2.3.0.5, which defines the computation-related concepts that we need.

Important! Unlike our practice in Section 5.1, where run time was expressed as a function of input *length*, in the present section we will gauge run time as function of input (numerical) value.

Thus, for the record:

5.2.0.40 Definition. Consider the function $f = M_{\vec{y}}^{\vec{x}_n}$, where M is a URM—whether M is normalized or not is immaterial for the purpose of this definition. A function

---

[154]Dennis Ritchie.

$\lambda \vec{x}_n.t(\vec{x}_n)$ *majorizes* the run time complexity of $M_{\mathbf{y}}^{\vec{x}_n}$ iff, for all $\vec{a}_n$, if $f(\vec{a}_n) \downarrow$ with an M-computation of length l, then $l \le t(\vec{a}_n)$; else if $f(\vec{a}_n) \uparrow$, then also $t(\vec{a}_n) \uparrow$.

We say that $\lambda \vec{x}_n.f(\vec{x}_n)$ is *computable within time $\lambda \vec{x}_n.t(\vec{x}_n)$*. $\square$

5.2.0.41 Simulation lemma. *Let M be a normalized URM (2.3.0.4) with variables $V_1, V_2, \ldots V_{n+1}, V_{n+2}, \ldots, V_m$, of which V_1 is the output variable and the $V_i, for\ i=2, \ldots, n+1$, are input variables. With reference to 2.3.0.5 we define $m+1$ simulating functions—for all $y, \vec{a}_n$—as follows:*

$v_i(y, \vec{a}_n) =$ *the value of variable V_i in the y-th ID of a (possibly non terminating) computation with input $\vec{a}_n$*

$I(y, \vec{a}_n) \ \ =$ *instruction number in the y-th ID of a (possibly non terminating) computation with input $\vec{a}_n$*

All the simulating functions are in $\mathcal{K}_1^{sim}$.

In view of (v) in 2.3.0.5, all the simulating functions are total, since once the instruction **stop** is reached the computation continues forever "trivially", that is, without changing either the V_i or the instruction number.

Proof. We have the following simultaneous recursion that defines the simulating functions:

$$
\begin{aligned}
v_1(0, \vec{a}_n) &= 0 \\
v_i(0, \vec{a}_n) &= a_{i-1}, \text{ for } i = 2, \ldots, n+1 \\
v_i(0, \vec{a}_n) &= 0, \text{ for } i = n+2, \ldots, m \\
I(0, \vec{a}_n) &= 1
\end{aligned}
$$

For $y \ge 0$ and $i = 1, \ldots, m$, we have

$$
v_i(y+1, \vec{a}_n) = \begin{cases}
c & \text{if } I(y, \vec{a}_n) = k \text{ where "}k : V_i \leftarrow c\text{" is in } M \\
v_i(y, \vec{a}_n) + 1 & \text{if } I(y, \vec{a}_n) = k \text{ where "}k : V_i \leftarrow V_i + 1\text{" is in } M \\
v_i(y, \vec{a}_n) \dotminus 1 & \text{if } I(y, \vec{a}_n) = k \text{ where "}k : V_i \leftarrow V_i \dotminus 1\text{" is in } M \\
v_i(y, \vec{a}_n) & \text{otherwise}
\end{cases}
$$

$$
I(y+1, \vec{a}_n) = \begin{cases}
l_1 & \text{if } I(y, \vec{a}_n) = k \text{ where "}k : \textbf{if } V_i = 0 \textbf{ goto } l_1 \textbf{ else} \\
& \textbf{goto } l_2\text{" is in } M \text{ and } v_i(y, \vec{a}_n) = 0 \\
l_2 & \text{if } I(y, \vec{a}_n) = k \text{ where "}k : \textbf{if } V_i = 0 \textbf{ goto } l_1 \textbf{ else} \\
& \textbf{goto } l_2\text{" is in } M \text{ and } v_i(y, \vec{a}_n) > 0 \\
k & \text{if } I(y, \vec{a}_n) = k \text{ where "}k : \textbf{stop}\text{" is in } M \\
I(y, \vec{a}_n) + 1 & \text{otherwise}
\end{cases}
$$

Since the iterator functions only utilize the functions $\lambda x.a$, $\lambda x.x + 1$, $\lambda x.x \dotminus 1$, $\lambda x.x$, and predicates $\lambda x.x = a$, and $\lambda x.x > a$—all in $\mathcal{K}_1^{sim}$ and $\mathcal{K}_{1,*}^{sim}$—it follows that all the simulating functions are in $\mathcal{K}_2^{sim}$, as claimed. $\square$

5.2.0.42 Example. Let M be the program below

$$1 : V_1 \leftarrow V_1 + 1$$
$$2 : V_2 \leftarrow V_2 \mathbin{\dot-} 1$$
$$3 : \textbf{if } V_2 = 0 \textbf{ goto } 4 \textbf{ else goto } 1$$
$$4 : \textbf{stop}$$

Let us assume that V_2 is the input variable and V_1 is the output variable. The simulating equations take the concrete form below, where a denotes the input value:

$$v_1(0, a) = 0$$
$$v_2(0, a) = a$$

For $y \geq 0$ we have

$$v_1(y+1, a) = \begin{cases} v_1(y, a) + 1 & \text{if } I(y, a) = 1 \\ v_1(y, a) & \text{otherwise} \end{cases}$$

$$v_2(y+1, a) = \begin{cases} v_2(y, a) \mathbin{\dot-} 1 & \text{if } I(y, a) = 2 \\ v_2(y, a) & \text{otherwise} \end{cases}$$

$$I(y+1, a) = \begin{cases} 4 & \text{if } I(y, a) = 3 \wedge v_2(y, a) = 0 \\ 1 & \text{if } I(y, a) = 3 \wedge v_2(y, a) > 0 \\ 4 & \text{if } I(y, a) = 4 \\ I(y, a) + 1 & \text{otherwise} \end{cases}$$

$\square$

5.2.0.43 Corollary. *The simulating functions are in $\mathcal{K}_4$.*

Proof. The above mentioned predicates and functions that are part of the iterator are in $\mathcal{K}_1$ and $\mathcal{K}_{1,*}$. Moreover, K_1 is closed under definition by cases (5.2.0.24). To convert the simultaneous recursion to a single recursion and back, we need pairing functions and their projections.

The quadratic pairing function $J = \lambda xy.(x+y)^2 + x$ is appropriate. Immediately, $J \in \mathcal{K}_2$ by 5.2.0.20. Now, let us place its projections, K and L, in the Axt hierarchy. We know that (2.1.4.8) $Kz = z \mathbin{\dot-} \lfloor \sqrt{z} \rfloor^2$ and $Lz = \lfloor \sqrt{z} \rfloor \mathbin{\dot-} Kz$. By the results of 5.2.0.20 we need only locate $\lambda z.\lfloor \sqrt{z} \rfloor$ in the hierarchy.

We start by noting that if $z + 1$ is a perfect square, that is, $z + 1 = (k+1)^2$ for some k, then $z + 1 = k^2 + 2k + 1$ hence $z = k^2 + 2k$, thus

$$k^2 \leq z < (k+1)^2$$

hence $k = \lfloor \sqrt{z} \rfloor$. This yields

$$\lfloor \sqrt{z+1} \rfloor = k + 1 = \lfloor \sqrt{z} \rfloor + 1 \tag{1}$$

Suppose next that $z + 1$ is *not* a perfect square. That is,

$$m^2 < z + 1 < (m + 1)^2 \tag{2}$$

for some m, and hence $m^2 \le z < m^2 + 2m < (m + 1)^2$. This entails $m \le \sqrt{z} < m + 1$, thus $m = \lfloor \sqrt{z} \rfloor$. But $m = \lfloor \sqrt{z + 1} \rfloor$ as well, by (2).

At the end of all this we obtain the following recursion:

$$\begin{cases} \lfloor \sqrt{0} \rfloor & = 0 \\ \lfloor \sqrt{z + 1} \rfloor & = \begin{cases} \lfloor \sqrt{z} \rfloor + 1 & \text{if } z + 1 = (\lfloor \sqrt{z} \rfloor + 1)^2 \\ \lfloor \sqrt{z} \rfloor & \text{otherwise} \end{cases} \end{cases}$$

By reference to 5.2.0.20—and noting that $x = y \equiv (x \dotminus y) + (y \dotminus x) = 0$, thus $\lambda xy.x = y \in \mathcal{K}_{2,*}$—we see that $\lambda z. \lfloor \sqrt{z} \rfloor \in \mathcal{K}_3$, and thus so are K and L. But then, the coding/decoding scheme that is based on this J, K, L (2.1.4.9) is in $\mathcal{K}_3$. Referring to 2.1.3.1, we see that, due to the presence of the Π_i^{m+1} in the iterator part, the single recursion that simulates the simultaneous recursion of the simulation lemma yields the function

$$\lambda y \vec{a}_n . [\![\, I(y, \vec{a}_n), v_1(y, \vec{a}_n), \ldots, v_m(y, \vec{a}_n) \,]\!]^{(m+1)}$$

in $\mathcal{K}_4$. This guarantees that

$$\lambda y \vec{a}_n . \Pi_i^{m+1} \Big([\![\, I(y, \vec{a}_n), v_1(y, \vec{a}_n), \ldots, v_m(y, \vec{a}_n) \,]\!]^{(m+1)} \Big)$$

are in $\mathcal{K}_4$, for $i = 1, \ldots, m + 1$. $\qquad \square$

5.2.0.44 Corollary. *The simulating functions are in $\mathcal{E}^2$.*

Proof. Given that the iterators in the simultaneous recursion employed in 5.2.0.41 are trivially in $\mathcal{E}^2$, we only need to provide $\mathcal{E}^2$-bounds for all the produced functions (5.2.0.38). Well, $I(y, \vec{a}_n) \le k$, where k is the label of the stop instruction of M. On the other hand, since all we do with the iterators can at most add 1 in each step, we also have the bounds $v(y, \vec{a}_n) \le \max \vec{a}_n + y + C$, a bound which is in $\mathcal{E}^2$ as a function of y and $\vec{a}_n$, seeing that $\max(x, y) = x \dotminus y + y$. The "$+C$" accounts for all the constants that may be assigned to a variable during the computation (instructions of type $V_i \leftarrow a$). $\qquad \square$

We can now prove (the nontrivial) half of the Ritchie-Cobham property:

5.2.0.45 Lemma. *If $f = M_{\mathbf{z}}^{\vec{x}_n}$ runs on M within time $t \in \mathcal{E}^n$, for some $n \ge 2$, then $f \in \mathcal{E}^n$.*

Proof. Let the simulating functions of M be as in 5.2.0.41, where $\mathbf{z}$ is "V_1", the output variable. Then, for all $\vec{a}_n$, we have $f(\vec{a}_n) = v_1(t(\vec{a}_n), \vec{a}_n)$, and this settles the claim by 5.2.0.44. $\qquad \square$

The "easy" half of the Ritchie-Cobham property is proved by doing a bit of programming.

5.2.0.46 Lemma. *For $n \geq 2$, any $\lambda \vec{x}.f(\vec{x}) \in \mathcal{E}^n$ is URM-computable within time $\lambda \vec{x}.t(\vec{x}) \in \mathcal{E}^n$.*

Proof. Induction over $\mathcal{E}^n$.

We settle the case of the initial functions first (cf. 5.2.0.25). $\lambda x.x$ is computable, as $M_{V_1}^{V_2}$, within $O(x)$ steps by the normalized URM M below

$$
\begin{aligned}
&1 : V_1 \leftarrow V_1 + 1 \\
&2 : V_2 \leftarrow V_2 \div 1 \\
&3 : \textbf{if } V_2 = 0 \textbf{ goto } 4 \textbf{ else goto } 1 \\
&4 : \textbf{stop}
\end{aligned}
$$

while $\lambda x.x + 1$ is computable, as $N_{V_1}^{V_2}$, also within $O(x)$ steps by the normalized URM N below:

$$
\begin{aligned}
&1 : V_1 \leftarrow V_1 + 1 \\
&2 : V_2 \leftarrow V_2 \div 1 \\
&3 : \textbf{if } V_2 = 0 \textbf{ goto } 4 \textbf{ else goto } 1 \\
&4 : V_1 \leftarrow V_1 + 1 \\
&5 : \textbf{stop}
\end{aligned}
$$

The non normalized URM P below

$$
\begin{aligned}
&1 : V_1 \leftarrow V_1 + 1 \\
&2 : \textbf{stop}
\end{aligned}
$$

computes $\lambda x.x + 1$ as $P_{V_1}^{V_1}$ in $O(1)$ steps.

$\lambda xy.xy$ is computable by the following loop-program, R, within time $O(xy)$, as R_Z^{XY}:

$$
\begin{aligned}
&\textbf{Loop } X \\
&\quad \textbf{Loop } Y \\
&\qquad Z \leftarrow Z + 1 \\
&\quad \textbf{end} \\
&\textbf{end}
\end{aligned}
$$

A straightforward URM simulation of the above is

$$
\begin{aligned}
&1 : \textbf{goto } 7 \\
&2 : \textbf{goto } 5 \\
&3 : \quad Z \leftarrow Z + 1 \\
&4 : \quad Y \leftarrow Y \div 1 \\
&5 : \textbf{if } Y = 0 \textbf{ goto } 6 \textbf{ else goto } 3 \\
&6 : \quad X \leftarrow X \div 1 \\
&7 : \textbf{if } X = 0 \textbf{ goto } 8 \textbf{ else goto } 2 \\
&8 : \textbf{stop}
\end{aligned}
$$

This still runs within $O(xy)$ time. With the case of $n = 2$ done, we now turn to the initial functions of $\mathcal{E}^{n+1}$ for $n \geq 2$.

The only new case is A_n. We show that it is computable by some URM M within time A_n^k, for some k.

We know that $A_n \in \mathcal{L}_n$. So let $A_n = P_z^x$, where the program $P \in L_n$ terminates within $O(A_n^k(x))$ steps (cf. Exercise 5.3.22).

But how about computing P_z^x on a URM? We can efficiently translate any loop program into a URM program!

To this end, note that loop program instructions, other than those of type $X \leftarrow Y$ and the **Loop-end** pair, occur also in URM programs and thus can be the translated as themselves. On the other hand, $X \leftarrow Y$ can be simulated by a URM (cf. 2.1.1.10).

Recursively, assume that we know how to translate R into a URM $\widetilde{R}$ and consider Q:

$$Q : \begin{cases} \textbf{Loop } X \\ R \\ \textbf{end} \end{cases}$$

This is simulated by the URM

$$
M : \quad
\begin{array}{lll}
& B \leftarrow X & \text{\{A \textit{new} B is associated with \textit{each} instruction ``\textbf{Loop} X''}^{155}\} \\
& \textbf{goto } L & \text{\{ L labels the ``\textbf{end}'' that matches the simulated ``\textbf{Loop} X''\}} \\
& \widetilde{R} & \\
& B \leftarrow B \dotminus 1 & \\
L : & \textbf{if } B = 0 & \textbf{goto } L+1 \textbf{ else goto } M \\
L+1 : &&
\end{array}
$$

Let next the run time of a loop program be $O(t)$. If an instruction of type "$B \leftarrow X$" were to take 1 step in a URM, then the above described simulating URM would also run within time $O(t)$. But this is not a primitive instruction of a URM! It takes time $O(X)$ to perform it (cf. 2.1.1.10).

For the P above in particular, and since $t = O(A_n^k(x))$, it follows that for any variable X of P, we have $O(X) = O(A_n^k(x))$,[156] and thus the URM runs within time $O\big((A_n^k(x))^2\big) = O(A_n^{k+1}(x))$ due to $x^2 = O(A_2(x)) = O(A_n(x))$.

We have concluded the basis case for all $n \geq 2$. To conclude the induction over $\mathcal{E}^n$ ($n \geq 2$) we show that the property *propagates* with substitution and bounded recursion.

Let then f and g from $\mathcal{E}^n$, $n \geq 2$, be URM-computable (by programs M_f and M_g) with run times bounded by t_f and t_g—both in $\mathcal{E}^n$. Consider

$$\lambda \vec{x} \vec{y}.f(\vec{x}, g(\vec{y})) \tag{$*$}$$

We can (essentially) concatenate M_g and M_f in that order (cf. 2.1.1.11) to compute $(*)$. The run time of this program is bounded by $\lambda \vec{x} \vec{y}.t_g(\vec{y}) + t_f(\vec{x}, g(\vec{y}))$, which is in $\mathcal{E}^n$, just as $\lambda \vec{x} \vec{y}.f(\vec{x}, g(\vec{y}))$ is. The other cases of substitution are trivial and are omitted.

---

[155] For a given X the instruction "**Loop** X" may appear several times. Each occurrence is associated with a new "B".

[156] To see this think of X as the output variable!

Finally, let $\lambda x \vec{y}. f(x, \vec{y})$ be obtained by a bounded recursion from basis h, iterator g and bound B, all in $\mathcal{E}^n$, and all programmable in respective URMs within time bounds t_h, t_g and t_B, all in $\mathcal{E}^n$. A URM program for f, in "pseudo code", is

$$
\begin{aligned}
& z \leftarrow h(\vec{y}) \\
& i \leftarrow 0 \\
R: & \textbf{if } x = 0 \textbf{ goto } L \textbf{ else goto } L' \\
& z \leftarrow g(i, \vec{y}, z) \\
& i \leftarrow i + 1 \\
& x \leftarrow x \dot{-} 1 \\
& \textbf{goto } R \\
L: & \textbf{stop}
\end{aligned}
$$

Its run time is

$$
t_h(\vec{y}) + O\Big(\sum_{i<x} t_g(i, \vec{y}, f(i, \vec{y})) \Big) \; ^{157} \tag{1}
$$

Since t_h, t_g and f are all in $\mathcal{E}^n$, then so is the function given by expression (1), due to 5.2.0.33. $\qquad\square$

The simulation of a loop program by a URM given on p. 367 represents the general-purpose, "faithful" simulation that, in particular, is true to the fact that the number of iterations of a loop, **Loop** X, *depend only on the value of X upon entry in the loop.* That is the purpose of the new variable B.

The simulation on p. 366 is expedient but acceptable since neither X nor Y are present inside the "scope" of either loop.

By virtue of Lemmata 5.2.0.45 and 5.2.0.46 we have now proved:

5.2.0.47 Theorem. (The Ritchie-Cobham Property of $\mathcal{E}^n$) *For $n \geq 2$, a function f is in $\mathcal{E}^n$ iff it can be computed on some URM within time $t_f \in \mathcal{E}^n$.*

The Ritchie-Cobham property shows the extremely close relationship between static and computational complexity of primitive recursive functions: The *run time* complexity of a function f in $\mathcal{E}^{n+1}$—as it is measured by the amount of time it takes to compute it, namely, A_n^k—is exactly predicted by the *definitional* complexity of the function: its level in the hierarchy. And conversely! The run time predicts the definitional complexity. *Very accurately.*

We can now compare all the hierarchies that we introduced:

5.2.0.48 Corollary. *For $n \geq 2$, we have $\mathcal{K}_n^{sim} = \mathcal{E}^{n+1}$.*

Proof. The $\supseteq$ is immediate by 5.2.0.47: Let $f \in \mathcal{E}^{n+1}$ and let it run on some M within time $t_f \in \mathcal{E}^{n+1}$. Now $t_f(\vec{x}) \leq A_n^r(\max \vec{x})$, everywhere, by 5.2.0.30. If v_1 is, as before (5.2.0.41), the simulating function for the output variable of M, then

$$
f = \lambda \vec{x}. v_1(A_n^r(\max \vec{x}), \vec{x})
$$

---

[157] Of course, this denotes, for some C and D, the expression $t_h(\vec{y}) + C \sum_{i<x} t_g(i, \vec{y}, f(i, \vec{y})) + D$.

But $A_n^r \in \mathcal{K}_n^{sim}$ (5.2.0.12), thus, $f \in \mathcal{K}_n^{sim}$.

For the $\subseteq$ we do induction on $n \geq 2$. For $n = 2$ note that, trivially, $\mathcal{K}_0^{sim} \subseteq \mathcal{E}^3$. Now—by varying r— we can make A_1^r majorize every function of $\mathcal{K}_1^{sim}$ (5.2.0.13), thus every simultaneous recursion that produces functions in $\mathcal{K}_1^{sim}$ (from functions in $\mathcal{K}_0^{sim}$) is a bounded recursion within $\mathcal{E}^3$ ($A_1 = \lambda x.2x + 2 \in \mathcal{E}^3$). Therefore, $\mathcal{K}_1^{sim} \subseteq \mathcal{E}^3$. Repeating this argument we have that

every simultaneous recursion that produces functions in $\mathcal{K}_2^{sim}$ (from functions in $\mathcal{K}_1^{sim}$) is a bounded recursion within $\mathcal{E}^3$ (since $A_2 \in \mathcal{E}^3$).

thus, $\mathcal{K}_2^{sim} \subseteq \mathcal{E}^3$.

Taking as an I.H. the validity of the claim for some fixed $n \geq 2$, the case for $n + 1$ is repeating the idea we employed in the basis: recursions taking us from $\mathcal{K}_n^{sim}$ to $\mathcal{K}_{n+1}^{sim}$ are bounded recursions performed *within* $\mathcal{E}^{n+2}$ ($\supseteq \mathcal{E}^{n+1} \supseteq$, by I.H., $\mathcal{K}_n^{sim}$), with bounding function some A_{n+1}^r—since $A_{n+1}^r \in \mathcal{K}_{n+1}^{sim} \cap \mathcal{E}^{n+2}$. $\qquad\square$

By 5.2.0.18 we have at once

5.2.0.49 Corollary. *For $n \geq 2$, we have $\mathcal{L}_n = \mathcal{E}^{n+1}$.*

5.2.0.50 Corollary. *For $n \geq 4$, we have $\mathcal{K}_n = \mathcal{E}^{n+1}$.*

Proof. The proof follows very closely that of 5.2.0.48. The $\subseteq$ goes through unchanged, but the $\supseteq$ "starts" later, $n \geq 4$, due to the fact that the simulating function v_1 is in K_4; cf. 5.2.0.43. $\qquad\square$

Schwichtenberg has improved 5.2.0.50 by proving the case for $n = 3$ [H. Schwichtenberg (1969)]. This is retold in Tourlakis (1984). H. Müller (1973) gives a proof for the case $n = 2$.

5.2.0.51 Remark. (A Very Hard Problem—Revisited) Corollary 5.2.0.49 adversely impacts a problem of practical significance: That of *program correctness*. The problem "program correctness" is an instance of the *equivalence problem* of programs, since it tasks us to determine whether a program follows faithfully a *specification*, the latter being, of course, given by a *finite description*, just as the program is.

We strengthen here the observation we made in 2.5.0.22 about the *equivalence problem* of primitive recursive functions, that is, the equivalence problem of loop programs:

Given loop programs P and Q, is it the case that $P_Y^{\vec{X}} = Q_Y^{\vec{X}}$?

We saw that the equivalence problem for $\mathcal{PR}$ is unsolvable—indeed, worse: not even c.e.—as a consequence of the fact $\lambda x.1$ and $\lambda y.\chi_T(x, x, y)$ are in $\mathcal{PR}$.

As these functions are also in $\mathcal{E}^3$—a fact that can be readily verified by looking at the proof of the normal form theorem (Corollary 2.3.0.8)—it follows that the equivalence problem for $\mathcal{E}^3$ functions is not c.e. either. By virtue of 5.2.0.49, this yields the rather disappointing alternative formulation:

The equivalence problem for programs in L_2—i.e., those that have loop depth equal to two—is not c.e.

Thus the various techniques employed to tackle *loop correctness* can be *successful in all instances of the problem* only when we have un-nested loops—L_1-programs. This holds true even though the loops are "FOTRAN-like", that is, they always terminate and the number of iterations of any such loop is known at the time the loop is entered. It should be noted that Tsichritzis (cf. Tsichritzis (1970) and Tourlakis (1984)) has shown that programs in L_1 have a solvable equivalence problem, but, on the other hand, the corresponding set of functions, $\mathscr{L}_1$ is rather trivial: it is the closure under substitution of $\{\lambda xy.x + y, \lambda x.x \div 1, \lambda xyz.$ if $x = 0$ then y else $z, \lambda x, \lfloor x/k \rfloor, \lambda x.rem(x, k)\}$. That is, all "looping" can be eliminated if we adopt this enlarged set of initial functions. □

5.3 ADDITIONAL EXERCISES

1. Fix an alphabet Σ with $m > 1$ elements. Show that the following functions are in Cobham's class $\mathscr{C}_\Sigma$ (cf. 5.1.1.7):

    (i) $\lambda x.x$

    (ii) $\lambda x.0$

    (iii) The *left successors*: $\lambda x.d * x$, for all $d \in \Sigma$

    (iv) $\lambda x.x^R$, where x^R is the number whose m-adic notation is the reverse of that of x.

2. Fix an alphabet Σ with $m > 1$ elements. Show that Cobham's class $\mathscr{C}_\Sigma$ is closed under *bounded* left *recursion on notation*, that is, under the schema below, where the h, $(g_d)_{d \in \Sigma}$, and B are in $\mathscr{C}_\Sigma$.

$$\begin{aligned}
f(0, \vec{y}_n) &= h(\vec{y}_n) \\
f(d * x, \vec{y}_n) &= g_d(x, \vec{y}_n, f(x, \vec{y}_n)), \text{ for all } d \in \Sigma \\
\left| f(x, \vec{y}_n) \right| &\leq \left| B(x, \vec{y}_n) \right|
\end{aligned}$$

Hint. (iv) of the preceding exercise helps.

3. Fix an alphabet Σ with $m > 1$ elements. Show that the following functions are in Cobham's class $\mathscr{C}_\Sigma$:

    (i) $init = \lambda x.y$, if $x = y * d$, for some $d \in \Sigma$. We define $init(0) = 0$.

    (ii) $last = \lambda x.d$, if $x = y * d$, for some $d \in \Sigma$. We define $last(0) = 0$.

    (iii) $first = \lambda x.d$, if $x = d * y$, for some $d \in \Sigma$. We define $first(0) = 0$.

    (iv) $tail = \lambda x.y$, if $x = d * y$, for some $d \in \Sigma$. We define $tail(0) = 0$.

    (v) $\lambda xyz.$if $x = 0$ then y else z.

    (vi) $\lambda x.1 \div x$.

(vii) $\lambda x.x \dot{-} 1$.

(viii) $ones = \lambda x.$if $x = 0$ then 0 else the numerical value of $\overbrace{11\cdots 1}^{|x|\ \text{ones}}$.

(ix) $sub = \lambda xy.$if $|x| \leq |y|$ then 0 else the numerical value of $\overbrace{11\cdots 1}^{|x|-|y|\ \text{ones}}$.

(x) $\lambda xy.x * y$ (that is, the numerical value of the string $x * y$).

4. For any Σ we define $\mathscr{C}_*$ to be the class of relations $\{f(\vec{x}) = 0 : f \in \mathscr{C}\}$.

Now, fix an alphabet Σ with $m > 1$ elements. We define

5.3.0.52 Definition. We define the predicates xBy ("x begins y"), xEy ("x ends y") and xPy ("x is part of y") by $(\exists z)y = x*z, (\exists z)y = z*x, (\exists z, w)y = w*x*z$ respectively.

If R is a relation, then the notations $(\exists y)_{Bz}R, (\exists y)_{Ez}R, (\exists y)_{Pz}R$, mean, respectively, $(\exists y)(yBz \wedge R), (\exists y)(yEz \wedge R), (\exists y)(yPz \wedge R)$. They are read in the obvious way, e.g., "there is a y that begins z such that R holds". The quantifiers $(\forall y)_{Bz}R, (\forall y)_{Ez}R, (\forall y)_{Pz}R$ are defined similarly. $\qquad \square$

With these concepts in mind, prove

(i) $\mathscr{C}_*$ is closed under Boolean operations.

(ii) $\mathscr{C}_*$ is closed under $(\exists y)_{Bz}R$.

(iii) $\mathscr{C}_*$ is closed under $(\exists y)_{Ez}R$.

(iv) $\mathscr{C}_*$ is closed under $(\forall y)_{Bz}R$.

(v) $\mathscr{C}_*$ is closed under $(\forall y)_{Ez}R$.

Hint. We do not know yet whether either of the relations xBy or $x = y$ are in $\mathscr{C}_*$. For (ii) you should offer a proof by direct bounded recursion on notation to define a function $f \in \mathscr{C}$ such that $f(z, \vec{x}) = 0 \equiv (\exists y)_{Bz}R(y, \vec{x})$. Your *assumption* will be that a function $r \in \mathscr{C}$ exists such that $r(y, \vec{x}) = 0 \equiv R(y, \vec{x})$.

5. Fix an alphabet Σ with $m > 1$ elements. Show that the following relations are in $\mathscr{C}_*$.

(i) $\lambda x.x = d$, for all $d \in \Sigma$.

(ii) $\lambda x.tally_d(x)$, for all $d \in \Sigma$, where $tally_d(x)$ is true iff $x = \overbrace{d\ldots d}^{n}$, for $n > 0$.

(iii) $\lambda xy.|x| \leq |y|$.

(iv) $\lambda xy.x = y$.

Hint. This is true iff x and y have the same length, and every prefix of x is also a prefix of y.

(v) $\lambda xy.xBy$, $\lambda xy.xEy$.

(vi) $\lambda xyz.z = x * y$.

(vii) $\lambda xy.xPy$.

6. Fix an alphabet Σ with $m > 1$ elements. Show that $\mathscr{C}_*$ is closed under $(\exists y)_{P_z} R$ and $(\forall y)_{P_z} R$.

7. **5.3.0.53 Definition.** Let Q be a relation. We define the notation $(\max y)_{Bz} Q(y, \vec{x})$ and $(\max y)_{Ez} Q(y, \vec{x})$ by

$$(\max y)_{Bz} Q(y, \vec{x}) = \begin{cases} \max\{y : yBz \wedge Q(y, \vec{x})\} \\ 0 & \text{if the max does not exist} \end{cases}$$

and

$$(\max y)_{Ez} Q(y, \vec{x}) = \begin{cases} \max\{y : yEz \wedge Q(y, \vec{x})\} \\ 0 & \text{if the max does not exist} \end{cases}$$

$\square$

Prove that if Q is in $\mathscr{C}_*$, then $\lambda z\vec{x}.(\max y)_{Bz} Q(y, \vec{x})$ and $\lambda z\vec{x}.(\max y)_{Ez} Q(y, \vec{x})$ are in $\mathscr{C}$.

8. Fix an alphabet Σ with $m > 1$ elements. Let the expression $maxtal_d(x)$ denote *the* maximum length tally of d-symbols that is a substring of x. We let $maxtal_d(0) = 0$.

Prove that $\lambda xy.y = maxtal_d(x)$ is in $\mathscr{C}_*$.

9. Fix an alphabet Σ with $m > 1$ elements.

Prove that the function $\lambda x.maxtal_d(x)$ is in $\mathscr{C}$.

10. Fix an alphabet Σ with $m > 1$ elements.

Define a pairing function $J(x, y)$ as

$$J(x, y) \stackrel{Def}{=} x * 2 * 1 * maxtal_1(x * 2 * y) * 2 * y$$

Prove that we have projections K and L in $\mathscr{C}$. For all x and y, these satisfy $K(J(x, y)) = x$ and $L(J(x, y)) = y$.

11. Use 5.3.9 to devise, for any fixed $n > 1$, a coding/decoding scheme $[\![\ldots]\!]^{(n)}$ and Π_i^n in $\mathscr{C}$. Cf. 2.1.4.9 and 2.1.4.10. Conclude that $\mathscr{C}$ is closed under simultaneous bounded (left and right) recursion on notation.

12. Devise, in (string-processing) URM pseudo code, an algorithm that will construct a CNF for any given formula $\mathscr{A}$ in poly-time with respect to $|\mathscr{A}|$. Your algorithm will do the preprocessing and will then track the steps of the induction proof.

13. $3SAT$ is the set of all satisfiable Boolean formulae written in CNF with clauses that each have length equal to 3 literals. Show that $3SAT$ is $\mathcal{N}\mathcal{P}$-complete by showing it in $\mathcal{N}\mathcal{P}$ and also showing that $CSAT \leq_p 3SAT$.

Hint. Explore the last case of the I.S. of the induction presented in 5.1.3.2, and see how it can be employed to transform any clause of $n > 3$ literals, $l_1 \vee l_2 \vee \cdots \vee l_n$, in CNF with clause-length equal to 3, without affecting satsfiability.

14. Complete the proof in 5.2.0.6.

15. Prove 5.2.0.11.

16. Prove 5.2.0.13.

17. Carry out the suggested induction proof in 5.2.0.18.

18. Prove that the relations $\lambda xy.x \leq y$, $\lambda xy.x < y$ and $\lambda xy.x = y$ are in $\mathcal{K}_{2,*}$ and hence in $\mathcal{K}_{2,*}^{sim}$.

19. Prove that $\lambda xy.x = y$ and $\lambda xy.x \neq y$ are in $\mathcal{E}_*^0$.

20. Prove the case (3) of 5.2.0.30.

21. Prove that all of $g_0, g_1, \ldots, g_n$ are in $\mathcal{E}^{n+1}$, for $n \geq 0$.

22. Prove that every loop program in L_n with input variables $\vec{X}$, has run time $O(A_n^k(\vec{X}))$ for some k that depends on the program.

Hint. There is a trivial "trick" to measure the run time of a loop program, using a *new* variable. Then refer to 5.2.0.13 and note 5.2.0.18.

23. Prove that the Kleene predicate $T^{(n)}$ (for any n) is in $\mathcal{E}_*^3$, and that the decoding function d is also in $\mathcal{E}^3$.

Hint. Systematically scan the proof that $T^{(n)} \in \mathcal{PR}_*$ and $d \in \mathcal{PR}$ given in 2.3.0.7 and modify it to obtain this sharper result.

In fact,

24. Prove that $T^{(n)} \in \mathcal{E}_*^0$ and $d \in \mathcal{E}^0$.

25. The *restricted bounded summation* of a function f is defined [Grzegorczyk (1953)] to be

$$\lambda z\vec{x}. \sum_{i<z}(1 \dotdiv f(i,\vec{x}))$$

Prove that $\mathcal{E}^0$ is closed under restricted bounded summation.

26. [Grzegorczyk (1953)] Prove that every c.e. set can be enumerated by an $\mathcal{E}^0$-function.

27. [L. Kálmar (1943)] The class of *elementary functions*, $\mathcal{E}$, was defined by Kálmar as the closure of $\{\lambda xy.x + y, \lambda xy.x \mathop{\dot{-}} y\}$ under substitution, bounded summation and bounded product (2.1.2.30).

Prove that $\mathcal{E} = \mathcal{E}^3$.

Hint. The $\subseteq$ direction is practically trivial. For $\supseteq$ you need a few tools:

- $\mathcal{E}$ contains $\lambda xy.xy$ (use sum) and $\lambda xy.x^y$ (use product).

- $\mathcal{E}$ contains $\lambda xy.x(1 \mathop{\dot{-}} y)$.

- $\mathcal{E}_*$ is closed under Boolean operations.

- $\mathcal{E}$ is closed under $(\overset{\circ}{\mu}y)_{<z}$. Use the observation that $(\forall i)_{<z} f(i, \vec{x}) \neq 0$ iff $\prod_{i<z} f(i, \vec{x}) \neq 0$. Using $\sum_{<z}$ you can now compute the smallest y that makes $f(y, \vec{x})$ zero.

- $\mathcal{E}_*$ is closed under bounded quantification (use product for the existential quantifier).

- $\mathcal{E}$ is closed under definition by cases.

- Simulate primitive recursion (bounded!) using the techniques from 2.11.1.

28. Prove that

$$\lambda x.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \left.\right\} x \ 2\text{'s}$$

is not in $\mathcal{E}$.

29. Prove that $\lambda xy.x + y$, $\lambda xy.x \mathop{\dot{-}} y$ and $\lambda xy.xy$ are in Cobham's $\mathscr{C}$.

Hint. Use the digit-by-digit "school method".

30. Prove that an alternative definition of $\mathcal{E}$ is this: the closure of $\{\lambda xy.x+y, \lambda xy.x \mathop{\dot{-}} y, \lambda xy.x^y\}$ under substitution and bounded summation.

31. Prove that $f \in \mathcal{PR}$ iff f is programmable on some URM that runs within time $t \in \mathcal{PR}$.

32. Prove that $\mathcal{PR}_* \subset \mathcal{R}_*$.

374

Bibliography

P. Axt. Iteration of Primitive Recursion. *Zeitschrift für math. Logik*, 11:253–255, 1965.

J. Bennett. *On Spectra*. PhD thesis, Princeton University, 1962.

E. Blum. A machine-independent theory of the complexity of recursive functions. *ACM*, 14:322–336, 1967.

N. Bourbaki. *Éléments de Mathématique; Théorie des Ensembles*. Hermann, Paris, 1966.

A. Church. A note on the Entscheidungsproblem. *J. Symbolic Logic*, 1:40–41, 101–102, 1936a.

A. Church. An unsolvable problem of elementary number theory. *Amer. Journal of Math.*, 58:345–363, 1936b. [Also in Davis (1965), 89–107].

A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *International Congress for Logic, Methodology and Philosophy of Science*, pp. 24–30. North-Holland, Amsterdam, 1964.

S. Cook. The complexity of theorem-proving procedures. In *Proceedings, 3rd ACM Symposium on Theory of Computing*, pp. 151–158, 1971.

M. Davis. *Computability and Unsolvability*. McGraw-Hill, New York, 1958.

M. Davis. *The Undecidable*. Raven Press, Hewlett, NY, 1965.

R. Dedekind. *Was sind und was sollen die Zahlen?* Vieweg, Braunschweig, 1888. [In English translation by W.W. Beman; cf. Dedekind (1963)].

R. Dedekind. *Essays on the Theory of Numbers*. Dover Publications, New York, 1963. [First English edition translated by W.W. Beman and published by Open Court Publishing, 1901].

H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, 1972.

K. Gödel. Die Vollständigkeit der Axiome des logischen Funktionen-kalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930.

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Math. und Physik*, 38:173–198, 1931. [Also in English in Davis (1965), 5–38].

D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, New York, 1994.

A. Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, 4: 1–45, 1953.

H. Müller. Characterization of the Elementary Functions in Terms of Nesting of Primitive Recursions. *Recursive Function Theory: Newsletter*, (5):14–15, April 1973.

H. Schwichtenberg. Rekursionszahlen und die Grzegorczyk-Hierarchie. *Arch. math. Logik*, 12:85–97, 1969.

P. Halmos. *Naive Set Theory*. Van Nostrand, New York, 1960.

W. Heinermann. *Untersuchungen über die Rekursionszahlen rekursiven Funktionen*. PhD thesis, Münster, 1961.

D. Hilbert and P. Bernays. *Grundlagen der Mathematik I and II*. Springer-Verlag, New York, 1968.

P. G. Hinman. *Recursion-Theoretic Hierarchies*. Springer-Verlag, New York, 1978.

J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Boston, 3rd edition, 2007.

L. Kalmár. An argument against the plausibility of Church's thesis. In *Constructivity in Mathematics*, pp. 72–80. *Proc. of the Colloquium*, Amsterdam, 1957.

E. Kamke. *Theory of Sets*. Dover Publications, Inc., New York, 1950. [Translated from the 2nd German edition by F. Bagemihl].

S. Kleene. General recursive functions of natural numbers. *Math. Annalen*, 112: 727–742, 1936.

S. Kleene. Recursive predicates and quantifiers. *Transactions of the Amer. Math. Soc.*, 53:41–73, 1943. [Also in Davis (1965), 255–287].

S. Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.

K. Kunen. Combinatorics. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter B.3, pp. 371–401. North-Holland, Amsterdam, 1978.

L. Kálmar. A Simple Example of an Undecidable Arithmetical Problem. *Matematikai és Fizikai Lapok*, 50:1–23, 1943.

W. J. LeVeque. *Topics in Number Theory, Volumes I and II*. Addison-Wesley, Reading, MA, 1956.

H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, Englewood Cliffs, NJ, 1998.

A. A. Markov. Theory of algorithms. *Transl. Amer. Math. Soc.*, 2(15), 1960.

A. R. Meyer and D. M. Ritchie. Computational complexity and program structure. Technical Report RC-1817, IBM, 1967.

M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1967.

C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

R. Péter. *Recursive Functions*. Academic Press, New York, 1967.

E. L. Post. Finite combinatory processes. *J. Symbolic Logic*, 1:103–105, 1936.

E. L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bull. Amer. Math. Soc.*, 50:284–316, 1944.

D. Ritchie. Complexity Classification of Primitive Recursive Functions by their Machine Programs. Term paper for Applied Mathematics 230, Harvard University, 1965.

M. Robinson, Raphael. Primitive recursive functions. *Bull. Amer. Math. Soc.*, 53: 925–942, 1947.

H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.

J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967.

M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, Boston, 1997.

R. M. Smullyan. *Theory of Formal Systems*. Number 47 in Annals of Math. Studies. Princeton University Press, Princeton, 1961.

R. M. Smullyan. *Gödel's Incompleteness Theorems*. Oxford University Press, Oxford, 1992.

G. Tourlakis. *Computability*. Reston Publishing, Reston, VA, 1984.

G. Tourlakis. *Lectures in Logic and Set Theory, Volume 1: Mathematical Logic*. Cambridge University Press, Cambridge, 2003a.

G. Tourlakis. *Lectures in Logic and Set Theory, Volume 2: Set Theory*. Cambridge University Press, Cambridge, 2003b.

G. Tourlakis. *Mathematical Logic*. John Wiley & Sons, Hoboken, NJ, 2008.

D. Tsichritzis. The Equivalence Problem of Simple Programs. *JACM*, 17:729–738, 1970.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math Soc.*, 2(42, 43):230–265, 544–546, 1936, 1937. [Also in Davis (1965), 115–154].

INDEX