Computation and

Discrete Mathematics

Klaus Sutner

Current version: June 2024 `v2.6`

Contents

Motivation

In other words: why? Why should a student of computer science or mathematics study logic, set theory and computability? How is any of this material relevant to "actual" topics in either one of these areas? It is often a good idea to hide behind the shoulders of giants, so let me start with a few quotations.

> Computers are mathematical machines.
> Computer programs are mathematical expressions.
> A programming language is a mathematical theory.
> Programming is a mathematical activity.
>
> Tony Hoare

> The standard of correctness and completeness necessary to get a computer program to work at all is a couple of orders of magnitude higher than the mathematical community's standard of valid proofs.
>
> Bill Thurston

> Generally, computer science, that no-nonsense child of logic, will exert growing influence on our thinking about the languages by which we express our vision of mathematics.
>
> Yuri Manin

> The Algorithm's coming-of-age as the new language of science promises to be the most disruptive scientific development since quantum mechanics.
>
> Bernard Chazelle

These comments are rather wide-ranging, as are the contents of these notes, but bear with me. Hoare[1] and Chazelle are computer scientists, Thurston and Manin are mathematicians. Loosely put, the suggestion here is that computer science, by necessity, leans heavily on mathematics, and some aspects of computer science are at least as hard as mathematics. On the other hand, mathematics can and will benefit from ideas emanating from computer science, the relationship is not a one-way street. This situation is not all that different from the relationship between mathematics and physics in the 18th and 19th centuries and should not be particularly surprising. Lastly, algorithms are a new paradigm that will profoundly change our understanding of all of science. This transformation is in its early stages, but one can already see the effect: for just about any field of human inquiry, there is now a *computational* version; look no further than computational anthropology.

From a computer science perspective, the question then becomes: how should one go about studying the sort of mathematics that is most relevant to computer science? It is important to realize that the traditional areas, arithmetic, algebra, geometry and calculus, while clearly helpful, are not quite the right framework. Alas, those are exactly the areas that you are already familiar with and feel most comfortable with. Instead, one needs to take a close look at mathematical logic and discrete mathematics. The rigorous and somewhat pedantic language of logic is quite similar to what one finds in programming languages, and the machinery of logic seems inevitable when it comes to proving programs correct.

[1] Incidentally, Hoare invented Quicksort, at age 22.

Turning to the mathematics point of view, it is often interesting to try to understand the computational content of mathematical concepts. For example, the extended Riemann Hypothesis[2] can be exploited to show that a particular primality testing algorithm runs in polynomial time—this test is deterministic, unlike the standard randomized tests in common use. It is now known that primality testing can be handled in polynomial time, without reference to open conjectures; sadly, the corresponding algorithm is irrelevant in practice. Questions relating to the complexity of computations seem rather difficult in general, the infamous $\mathbb{P} = \mathbb{NP}$ problem is just one particularly striking example. It is directly related to the general problem of verifying a proof as opposed to the problem of discovering one in the first place, obviously a topic to great interest to logicians and mathematicians. Incidentally, the problem was first mentioned by Gödel in a letter to John von Neumann in the 1950s, a full decade before the official beginnings of complexity theory. The question of whether $\mathbb{P} = \mathbb{NP}$ appears to be exceedingly difficult, entirely new mathematical methods may well be needed to make any serious inroads. Another interesting project is the formalization of mathematics, so proofs can be systematically verified by proof checkers. Transcribing a mathematical proof into a format so it can be checked is currently a time consuming task, but much progress has been made in the last decade.

Of all the quotations above, Thurston's is clearly the most militant. He won the Fields medal in 1982 and has used computers extensively in his research in low-dimensional topology. There is no question that he deeply understands the confluence of mathematics and computing; take a look at his wonderful book *Wordprocessing in Groups.* Discrete systems, and in particular digital computers, are very difficult to understand, even for the lofty standards of mathematics. This may sound a bit counterintuitive; after all, the classical, continuous world of mathematics where just about every object of interest is wildly infinite appears much more intimidating than just a bunch of bits, simple yes-or-no choices. Also note the point of reference that Thurston chose to calibrate the difficulties of writing correct programs against: mathematical proof, the hallmark of precision and rigor in mathematics. So what exactly is a proof in mathematics? Most mathematicians would agree to a statement along the following lines: a proof is a step-by-step argument that establishes the truth of an assertion in an absolutely compelling manner, beyond any reasonable objection. Unfortunately, this "definition" of a proof, while intuitively satisfying, does not really hold much water. In fact, the standard of acceptable proofs has changed considerably over time, and there really isn't a good, concise answer. In some sense, proofs have turned out to be extremely durable. For example, Euclid's proof of the infinitude of primes holds up fine and his work in geometry is still the prototype of axiomatization, a formalism that is nowadays accepted by almost everybody as the proper way of conducting mathematics. And yet, there has been a rather dramatic shift over the last two centuries, a shift in the direction of more and more precision, leading all the way up to attempts to mechanize proofs entirely, thus opening up the possibility of have proofs constructed or at least verified by machines, rather than humans. More on these efforts later.

The gentle reader might get the impression that we are trying to dismiss intuition as useless in the context of mathematics. Nothing could be further from the truth, in fact, intuition is the single most critical ingredient in all reasoning about mathematical concepts, absolutely nothing moves without some reasonable intuitive understanding. Unfortunately, intuition alone is simply not sufficient. Here is a comment by Mac Lane, one of the inventors of category theory.

[2] The Riemann Hypothesis is generally considered to be one of the most difficult open problems in mathematics; it has defied all attempts to find a solution for over 150 years.

> The sequence for the understanding of mathematics may be: intuition, trial, error, speculation, conjecture, proof. The mixture and the sequence of these events differ widely in different domains, but there is general agreement that the end product is rigorous proof–which we know and can recognize, without the formal advice of the logicians.
>
> Saunders Mac Lane

We fully agree with the first part, but the snide remark about logicians is rather uncalled for. Mac Lane ignores the fact that mathematics is becoming increasingly abstract, and, correspondingly, the trustworthiness of intuition or, more generally, the established machinery of mathematical argument, is decreasing. As painful as it may be for mathematicians to acknowledge, the use of proof checkers provides a level of precision that goes far beyond accepted "standard of valid proofs." V. Voevodsky, who won the Fields Medal in 2002 for his work in algebraic geometry, has this to say about computer verified proofs:

> I can't see how else it will go. I think the process will be first accepted by some small subset, then it will grow, and eventually it will become a really standard thing. The next step is when it will start to be taught at math grad schools, and then the next step is when it will be taught at the undergraduate level. That may take tens of years, I don't know, but I don't see what else could happen.
>
> Vladimir Voevodsky

Here is to hoping that it won't take "tens of years," though there are formidable obstacles to overcame. At any rate, in discrete mathematics, it is even more urgent to rethink standards of proof. The difficulties involved in "combinatorial" reasoning were already clearly articulated by John von Neumann.

> Everybody who has worked in formal logic will confirm that it is one of the technically most refractory parts of mathematics. The reason for this is that it deals with rigid, all-or-none concepts, and has very little contact with the continuous concept of the real or of the complex number, that is, with mathematical analysis. Yet analysis is the technically most successful and best-elaborated part of mathematics. Thus formal logic is, by the nature of its approach, cut off from the best cultivated portions of mathematics, and forced onto the most difficult part of the mathematical terrain, into combinatorics.
>
> . . . The theory of automata, the digital, all-or-none type as discussed up to now, is certainly a chapter in formal logic. It would, therefore, seem that it will have to share this unattractive property of formal logic. It will have to be, from the mathematical point of view, combinatorial rather than analytical.
>
> John von Neumann

We can trust von Neumann, if he says something is "refractory," it truly is. Unfortunately, classical continuous mathematics is provides little help when it comes to reasoning about a computer program, either in terms of resource requirements or in terms of correctness. To put this bluntly, without supporting analysis, computer programs, protocols and even hardware are in danger of being basically useless. Here is a famous example from 1994, concerning a problem with floating point arithmetic in the Pentium chip.

The Pentium chip integrated millions of transistors and was a major step forward in the development of digital computing. Sadly, for certain inputs, the arithmetic unit simply returned wrong answers.

$$\begin{array}{ll} 4195835.0/3145727.0 = 1.33382044913624100 25 & \text{correct} \\ 4195835.0/3145727.0 = 1.33373906890203758 94 & \text{pentium} \end{array}$$

Ironically, at the time of the discovery of the bug, there were already verification methods available (developed by the late Ed Clarke) that would have found the problem—if only they had been applied. On the upside, these algorithms were powerful enough to prove that the patch Intel concocted was indeed correct.

For those still unconvinced that a close inspection of proofs is a good idea for the aspiring computer scientist, one final quote.

> The utterly pure theory of mathematical proof and the utterly technological theory of machine computation are at bottom one, and the basic insights of each are insights of the other.
>
> W. V. O. Quine

This is from from Quine's book *On The Application of Modern Logic*. The link between computation and proof presents an obvious opportunity: we can use one area to illuminate the other, and conversely. In particular, we can use computation to make mathematical concepts more tangible and more accessible.

The Role of Intuition

Our repeated references to the need for formal, rigorous arguments might create the wrong impression that intuition is a strictly secondary issue here. It may seem tempting to focus on technical details early on, rather than trying first and foremost to get an intuitive grasp of the material, details be damned. This formalize-early approach is absolutely counterproductive and only leads to major frustration. The best way to handle a new concept is based on the following pattern:

Intuition	Understand the concept's meaning, its purpose, its intent.
Intuition	Understand the concept's meaning, its purpose, its intent.
Intuition	Understand the concept's meaning, its purpose, its intent.
Examples	Find some objects that the definition applies to.
Counterexamples	Find some objects where it nearly applies, but not quit.
Formalization	Pin things down in a semi-formal way, a definition.
Computation	Understand the computational aspects (if any).
Results	Be aware of basic theorems associated with the concept.
Connections	Understand how the concept fits into a larger framework.

The three-fold repetition of intuition is a stylistic atrocity, but it is worth hammering home this point: developing good, strong intuition is arguably far more important than anything else. In a nutshell, and with abject apologies to Don Knuth[3]:

> Premature formalization is the root of all evil.

It stands to reason that the ultimate reason for the difficulties humans encounter with formal, strictly precise methods is plain neuro-anatomy: our human cognitive system is actually quite bad at strictly formal manipulations, but we can lean heavily on intuition, informal insights and analogous thinking. For digital computers, the situation is more or less the opposite; to wit, they have no intuition whatsoever, but are excellent at plowing through long, insufferably boring sequences of strictly mechanical steps; they are *persistent plodders* according to the logician Hao Wang. One should think of this stark difference as a great opportunity: we can use computers to produce visualizations, carry out tedious calculations, find examples and counterexamples, check proofs and occasionally even find some. On a more mundane level, the internet has made it much easier to find and access related information, to communicate with colleagues, to cooperate on difficult projects and so on. At present, usability is admittedly a major problem, there is often a steep learning curve associated with assorted tools that can be quite useful once one has figured out how to deploy them properly. Still, at least in limited areas the payoff is substantially higher than the investment. The examples in this text will hopefully convince the reader that this is not just an empty promise.

History of these Notes

The goal of these notes is to present a few basic topics from set theory, logic (in the sense of proof theory), computability theory, automata theory, complexity theory and a little algebra and combinatorics. They came into existence by accretion, as a side-effect of teaching a number of courses in the theory curriculum[4]. Consequently, they are not as uniform as they ought to be, some parts are a bit more advanced than others (and can safely be skipped on first reading), some are too long, some are too short. On the upside, the current text represents a concerted effort to bridge the annoying gap between mathematics and computer science courses that plagues most, if not all, current textbooks in the area. While there are dozens of mathematics textbooks that have some enhancement like "for computer science" in their title, that seems to be mostly a public relations effort, there is typically very little in the way of an actual reorganization of topics. Looking in the opposite direction, one encounters a remarkable asymmetry: apparently there is no need to present computer science topics "for mathematicians." One splendid exception that proves this rule is Don Knuth's seminal book series *The Art of Computer Programming*, easily the

[3] D. Knuth pointed out that in the context of coding, "Premature optimization is the source of all evil."

[4] Discrete Math Primer (15-051), Mathematical Foundations (15-151), Great Ideas in TCS (15-251), Algorithms (15-451), Complexity Theory (15-455), FLAC (15-453) and CDM (15-354).

most important single publication in computer science. These books contain a staggering amount of mathematics, much of it developed for the explicit purpose of supporting Knuth's *analysis of algorithms*. Think of TAOCP as an attempt to present relevant mathematical material to computer science students, in a way that is also of keen interest to mathematics students. Unfortunately, the material is quite advanced and more suitable as a general reference and less so as a textbook.

Our quiet claim is that there really is not much of a difference between the two perspectives, computer science obviously cannot exist without mathematics, but, less obviously, mathematics can benefit greatly from taking a closer look at ideas and concepts that originated in computer science.

Speaking about Knuth, he has laid down the ultimate teaching challenge:

> It has often been said that a person does not really understand something until he teaches it to someone else. Actually, a person does not really understand something until he can teach it to a computer, i.e., express it as an algorithm.
>
> Don Knuth

This piece of advice is, of course, an exaggeration. For example, I have studied martial arts for half a century, and I cannot begin to teach a computer a front kick, though, on occasion, I was sorely tempted to kick my computer. However, for the intersection between mathematics and computer science, Knuth's challenge he is pertinent; writing a program to perform some task that one believes to have completely understood will often uncover gaps in one's knowledge, force a closer look and ultimately result in new insights.

Nota Bene: The sections on finite state machines, combinatorics and algebra are currently too much of a mess to post. Hopefully later this summer.

Part I

Foundations

One

Sets

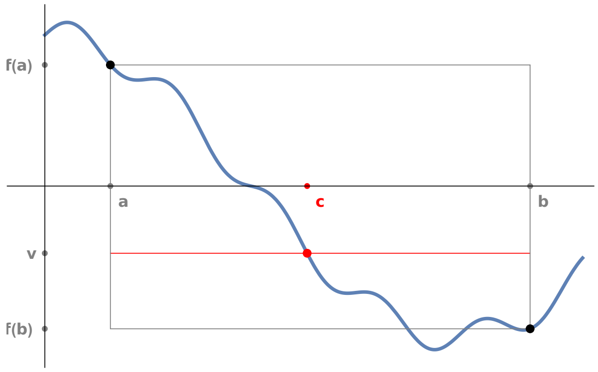
1.1 Intuitive Set Theory

1.1.1 Why Sets?

The ability to form collections of objects, such as a flock of sheep, a basket of apples or the planets of the solar system, is arguably one of the most basic human cognitive functions[1]. Collections of objects also play a critical rôle in mathematics and computer science: we speak of the collection of all prime numbers, of all differentiable functions, of all graphs on 5 points, of all terminating C-programs, of all computable functions, and so on. Somewhat surprisingly, one can interpret all mathematical objects as just collections, and this interpretation helps to clarify their basic properties. Before we start to develop an intuitive understanding of set theory, first an example taken from calculus, that shows how such a theory might come in handy. Calculus is not central to our enterprise, but it has the huge advantage that our intuition works extremely well in this area. For the most part, that is.

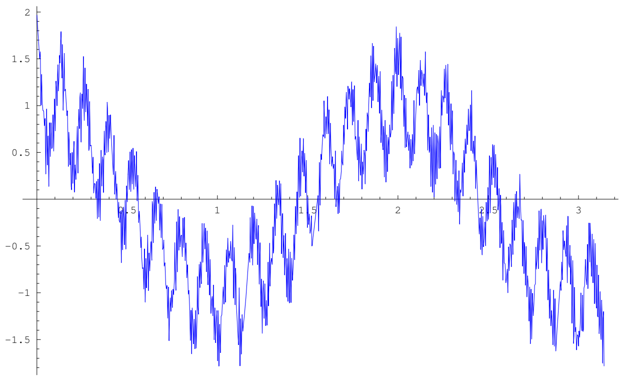
Intermediate Value Theorem

Let us take a look at a concrete example that you are already familiar with: the intermediate value theorem (IVT), one of the most basic results in calculus. Suppose f is some continuous function defined on an interval $[a, b]$ of the reals. Then, for any value v between $f(a)$ and $f(b)$, there is a point c in the interval such that $f(c) = v$. We can visualize the situation nicely:



[1] Though, amazingly, there is some amount of controversy about the last one. Poor Pluto.

This image is quite compelling: if we move the target value v on the y-axis up or down, the corresponding point c on the x-axis will follow. Since f never jumps, this will work over the whole interval from $f(a)$ to $f(b)$. Of course, even if we were to accept this argument for the given picture, it covers only one particular function. How about all the other continuous functions? Looking at a number of other examples, one senses that all the corresponding pictures will somehow look similar, and the same conclusion will apply. Alas, that intuitive feeling is terribly vague, one needs to somehow extract what precisely makes the pictures similar, and exploit that to come up with a weight-bearing argument. To be sure, prior to the early 1800s, the IVT had been considered basic and obvious: anyone who understands real numbers (the quantities in question here) and the concept of continuity can immediately "see" that this proposition holds. As we now know, geometric intuition can be quite deceptive: it took some two-thousand years for mathematicians to recognize the possibility of non-Euclidean geometries; the parallel axiom may hold, or it may not hold. Indeed, things can go off the rails quite easily. For example, geometric intuition might suggest that any continuous function is mostly differentiable: the absolute value function shows that there may be a few kinks, places where differentiability fails; with a bit more effort one may construct examples where there are infinitely many points of failure. But nothing prepares us for the following, rather disconcerting example due to K. Weierstrass:



Weierstrass realized that the functions $f(x) = \sum_n b^n \cos(a^n \pi x)$ are continuous but nowhere differentiable as long as $0 < b < 1$ and $ab > 1 + 3/2\pi$. Nowhere, there is not a single point where the tangent to this curve is defined, it oscillates too much. To make things worse, these are just trigonometric series, uncomfortably close to applied areas of mathematics. It even turns out that pathological functions like this one are in a sense more numerous than the well-behaved ones that our intuition works with best.

How do we deal with Weierstrass-type monsters that overwhelm our intuition? By exercising enormous care and attention when it comes to proofs, always making sure that intuition is supported by clear, logical arguments. In the case of IVT, the first major attempt to concoct a proof can be found in a classical 1821 textbook by A. Cauchy. Here is Cauchy's formulation, slightly edited for clarity:

Theorem 1.1.1 (Intermediate Value Theorem, Cauchy 1821) *If the function $f(x)$ is continuous with respect to the variable x between the limits $x = a$ and $x = b$ and if v designates a quantity between $f(a)$ and $f(b)$, one can always satisfy the equation $f(x) = v$ for one or several real values of x between a and b.*

And here is the argument proposed by Cauchy that is supposed to demonstrate this claim.

> To establish the preceding proposition, it suffices to see that the curve whose equation is $y = f(x)$ meets one or more times the straight line whose equation is $y = v$ in the interval between the ordinates that correspond to the abscissas a and b. Yet it is clear that this will take place under our hypotheses. Indeed, as the function $f(x)$ is continuous between the limits $x = a$ and $x = b$, the curve whose equation is $y = f(x)$ passing first through the point with coordinates a, $f(a)$ and second through the point with coordinates b, $f(b)$ will be continuous between these two points; and, as the constant ordinate v of the line whose equation is $y = v$ is found between the ordinates $f(a)$ and $f(b)$ of the two points under consideration, the line will necessarily pass between these two points so that it cannot avoid crossing the above mentioned curve corresponding to $y = f(x)$ in the interval.

From a modern perspective, Cauchy's proof is rather disappointing: it is really little more than a direct appeal to geometric intuition, spelled out in slightly more elaborate terms. One is hard pressed to find the reasoning any more convincing than a direct reference to a picture like the one above. Somewhat surprisingly, Cauchy offered a second proof in the appendix to his book that is essentially the modern argument, a form of binary search for the right number. First, it is easy to see that it suffices to show that, whenever $f(a) < 0 < f(b)$, there is some real number c, $a < c < b$, such that $f(c) = 0$. In slightly edited form, Cauchy then argues as follows. The proof is based on a standard algorithm, known as binary search, that can be used to determine a root of f.

> Define sequences (a_n) and (b_n) of reals in stages $k \geq 0$ as follows. Initially, at stage 0, set $a_0 = a$ and $b_0 = b$.
>
> *Stage $k > 0$:* Let $c = \frac{a_{k-1}+b_{k-1}}{2}$ be the midpoint between a_{k-1} and b_{k-1}.
>
> If we happen to hit a root, $f(c) = 0$, we are done, the process stops. Otherwise we distinguish two cases depending on the sign of $f(c)$. If $f(c) < 0$, we set $a_k = c$, $b_k = b_{k-1}$, else $f(c) > 0$ and we set $a_k = a_{k-1}$, $b_k = c$. We may safely assume the process never stops. An easy induction shows that
>
> - $a_k \leq a_{k+1} < v < b_{k+1} \leq b_k$
> - $|a_k - b_k| = |a - b|/2^k$
>
> But then both sequences converge to a uniquely determined real number c, and $f(c) = 0$ by continuity. □

This is a much better argument. The algorithm typically will take infinitely many steps, but that need not concern us at this point—we can stop after finitely many steps and at least obtain an approximately correct solution. The key step in the argument is the last one: the existence of $c = \lim a_k = \lim b_k$ is guaranteed by the fact that the reals are complete: every bounded set of reals has a least upper bound.

It is this completeness property that is critical for the IVT, not just in the intuitive sense but in a strict technical one: given a reasonable description of the reals without completeness we have:

IVT holds if, and only if, completeness holds.

So the question is: how do we produce a definition of the reals that allows us to conclude that they are in fact complete? How do we define precisely what a function is, and in particular a function from the reals to the reals? In a similar vein, how do we explain continuity in a way that supports careful reasoning about it, not just some possibly fallible intuition? If we wanted to push even further, can we organize the argument in a way that it could be checked for correctness by a computer, using a proof checker? Maybe a proof could even be found by a theorem prover? In the case of the IVT, this effort may seem like one bridge too far, but there are other results where a little assistance from a computer seems quite essential.

To be clear, most working mathematicians would approach this particular foundational question by simply stating the reals form a "complete Archimedean field," see the chapter on algebra for definitions. In other words, one filters out the critical properties of the reals, states them clearly as axioms and then postulates that there is in fact a collection of numbers that satisfy all the axioms. Still, it is worth thinking about a suitable justification at least once, to convince oneself that the axioms really make sense and can be used without having to worry about inconsistencies.

Sets to the Rescue

For the time being, we consider a set informally to be any collection of objects, an aggregate of sorts. The objects are referred to as the elements or members of the set. The key idea here is to think of the set itself as another single object, a new entity that is comprised of its members. Think of the elements of a set as being atomic objects whose existence we simply take for granted (we will introduce sets containing other sets as elements shortly). We can express small sets by listing their elements and enclosing the list in curly braces:

$$A = \{1, 2, \triangle, \square\}$$

Symbolically, we write the element-of relation as $\in$, so that $x \in A$ means that object x is an element of set A, it is contained in A or it belongs to A. Thus, $\triangle \in A$ but $3 \notin A$. The special set without any elements, the empty set, will be written $\emptyset$. Thus $x \notin \emptyset$, no matter what x might be.

With a little bit of exaggeration, one could say that this is all you need to know about sets. Of course, this is a white lie. Still, the simplicity of basic set theory makes it very tempting to use sets as a *foundational system*, a framework in which we can represent all objects in mathematics and computer science: natural numbers, reals, functions, vector spaces, computable functions, theorems, proofs and so on.

As a concrete example, suppose we want to formalize the notion of a Turing machine, one of the central ideas in computability theory. Since we can use sets freely, a closer look shows that we can come up with a decent description of Turing machines as long as we can manage to express two additional concepts: natural numbers and lists. One could argue that natural numbers are not in need of any further explanation, we all have a perfect intuitive grasp of counting numbers and we also have a simple notation system, say, $0, 1, 2, 3, \ldots, 125, \ldots$ By a list we mean an ordered sequence of objects and one can see that all lists can be expressed in terms of just pairs of objects, usually written (a, b): a is the first element of the pair, and b, the second. At this point we could decide to think of the natural numbers as atomic objects in our universe, and to add the concept of pairs as some sort of constructor. Our universe would then be populated by three distinct sorts of objects: natural numbers, sets and pairs. And these things mix and match, for example we can form a set of pairs of natural numbers.

Similar constructions can easily be handled in most programming languages and will seem rather quaint to those with some experience in writing computer programs. Remember, though, that we are looking for simplicity and clarity, we really want to avoid any unnecessary clutter. Another problem is that our basic constituents are connected: the set A from above has four elements, but there is no link between it and the natural number "four" in our universe. The question arises whether we can think of natural numbers and of pairs as sets themselves. The answer is a resounding yes, though it takes a bit effort to come up with representations that are useful in the long run, see section 1.1.3 and section 1.1.4 for details. For the moment, suffice it to say that the set $\{\emptyset, \{\emptyset\}\}$ is a perfectly elegant way to represent the natural number "two." We are not trying to claim that $\{\emptyset, \{\emptyset\}\}$ corresponds to the number "two" at some deep ontological level, or even, more pragmatically, in a cognitively appealing way. Rather, it is a convenient representation of a counting number as a set, no more and no less. This is really no different from some floating point number in memory representing a particular rational number or some byte representing the letter 'A'. Thus, we have a way to represent a Turing machine as a somewhat convoluted set, but a set that has a very simple and clear structure. If we understand how to reason about sets, we can also reason about Turing machines and computability.

move?

The idea of set theory as a foundation often meets with considerable skepticism. There are alternatives, in particular type theory and category theory are strong contenders and are critically important in the theory of programming languages. As one might suspect, there is some friction between the different camps; for instance, the category theorist Saunders Mac Lane famously complained:

> Does anyone seriously think that $2 = \{\emptyset, \{\emptyset\}\}$?

Of course not, no sane person would. The set on the right is just a representation of a particular counting number that might appear to be in no particular need of elaboration. Mac Lane picks a fairly cheap target here, natural numbers are arguably the most obvious and intuitive concept in mathematics. Arguably the same is true for integers and rationals, set representations for these concepts are just a warmup exercise and sanity check. Things change drastically when one gets to the real numbers, continuous functions and so on. Does anyone seriously think that $\sqrt{2}$ is just a point on the numberline, or maybe a string of digits? Or perhaps some algebraic expression, subject to particular rules of manipulation? Going one step further, when it comes to the reals, a solid set-theoretic definition seems unavoidable if one tries to understand these numbers. Rigorous proofs in analysis without set theory are hard to conceive, just think about the intermediate value theorem from above.

It is an interesting question how a working mathematician deals with a simple expression like x^2 in actual practice, clearly not by translating it internally into the monstrosity $x^{\{\emptyset,\{\emptyset\}\}}$. We do not unfold definitions down to the axiomatic level unless there is some urgent need to do so. Instead, we always group mathematical objects into informal *types*, collections of objects of the same kind, such as natural numbers, reals, lists, Turing machines or even proofs. Looking at x^2, we recognize 2 to be of type natural number and leave it at that. By the same token, x is presumably of type variable, and the superscript notation indicates exponentiation. All these concepts can be precisely defined in terms of sets, but that fact remains in the background. This type of abstraction and hiding of details is standard operating procedure in both mathematics and computer science. When one write code in a high-level language, one does emphatically not think about how exactly that code would translate into assembly, much less how it would execute in a particular runtime on some specific hardware—unless very special circumstances force us to dig down to that level.

To my mind, the most compelling reason to choose set theory for introductory purposes is the simplicity of the framework, a consideration that is irrelevant for experts. Another pragmatic reason to begin with sets is that, thanks to the tremendous influence of Bourbaki over the last century, the presentation of mathematics at all levels has been organized around set theory. This holds true even at the pre-university level and one should not change horses in the middle of the stream.

StringWorld

Since we are particularly interested in computation, it is important to point out another trap: in the context of algorithm analysis theory and complexity theory, objects are often represented by strings, finite sequences of symbols taken from some fixed alphabet such as ASCII characters. Specifically in complexity theory, it is customary to think of an object as being represented by a string over the alphabet $\{0,1\}$, a finite sequence of bits. All strictly finite structures can naturally be represented this way in an entirely natural way. For example, a finite directed graph can be thought of as an $n \times n$ matrix of bits, which can be flattened out, say, in row-major order into a single binary word of length n^2. From a more applied perspective, in computer memory, any data structure is just a sequence of zeros and ones, so the binary string model is clearly adequate in a way. Larger alphabets such as ASCII are just a notational convenience, in the end everything comes down to bits.

All true, but this mode of thinking cedes a lot of territory to digital computers, Wang's "persistent plodders." We can think of any finite structure as a binary string, but, for humans, this framework is rather too austere. We need higher order concepts, real meaning, not interminable strings of bits. Binary strings are not a good starting point to develop intuition and insight.

In a similar vein, whenever you see a definition of the form "such-and-such is an expression of the form ...," be exceedingly careful, a great many important details are probably being swept under the rug. For instance, you may see an apparently simple definition of a polynomial as an expression

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

where x is the "unknown" and the coefficients a_i are some sort of number, say, integers. This description is straightforward, but it raises quite a few problems. On this view, is $y+1$ a polynomial? How about $(x+1)^{10}$? The author most likely meant to include these objects, but it requires quite a bit of work to adjust the definition to handle them. In particular, we need to be able to rename variables, and we need to understand how to expand certain algebraic expressions into polynomials. This may seem rather trivial to an algebraist, but algorithmically it is not so straightforward. In particular expanding an implicit polynomial like $(x+1)^{10}$ is computationally hard in a technical sense, and one should be aware of such problems. In fact, if one could cheaply expand certain implicitly given polynomials one could easily solve computationally hard problems such as 3-coloring. In the end, it is better to give a more abstract definition of polynomials, say, in terms of a coproduct, and then think of various "expressions" as representations of polynomials, albeit eminently useful ones.

If this attack on StringWorld sounds a bit opaque right now, do not worry, we will come back to this issue on several occasions.

Bourbakism

Sets became on object of mathematical inquiry the late 1800s when Georg Cantor realized that, in order to truly understand Fourier analysis, one needs to first understand sets of reals, at a very detailed level. Over the following decades, set theory was embraced by some mathematicians as an excellent framework; others reacted with indifference or outright hostility[2]. Still, sets prevailed. The project to use sets as a foundation for math received a major boost starting in the 1930s, when a group of mostly French mathematicians decided to write a definitive exposition of (more or less) all of mathematics, in a strictly rigorous and semi-formal manner. They called themselves Nicolas Bourbaki, after a 19th century French general by that name:

The very first book in the series is titled "Theory of Sets" (or rather "Théory des Ensembles") and lays the foundation for all the other volumes. It is just about impossible to overstate the importance of Bourbaki for the development of mathematics, and, by implication, for computer science, over the last century. The Bourbaki approach to all things mathematics has become de facto gold-standard, at present most mathematics literature is written in a superficially set-theoretic style, along the lines of "a group is a set together with a binary operation." Surprisingly, this is true not just for research math but also for math education. Much of your prior training in math has been greatly influenced by Bourbaki (the dreaded "new math"). There are good reasons to be skeptical about several aspects of this development, in particular their logical framework is peculiar at best. Still, at present, there are no compelling alternatives. This may change in the not-so-distant future, in part due to the increasing influence of computer science on mathematics. For the time being, though, we live in a Bourbaki world.

1.1.2 Extensionality and Comprehension

Our treatment of sets in the next sections will be informal and intuitive. Later, in chapter 6, we will take a closer look at what needs to be done to formalize set theory and justify our intuitive approach by developing a set of axioms. At present, we will not be concerned with these technical details, we just want to pin down a few basic ideas that explain the nature of sets and help to make our intuitive constructions somewhat precise. It is then fairly easy to see that our axioms match up nicely with the informal approach.

We have been rather vague about what kinds of objects could populate a set. As mentioned already, one is often interested in a particular type of element such as natural numbers in computability or the reals in analysis. We could assume the reals $\mathbb{R}$ are simply given to us, and we can start forming set of reals right away. For instance, we can build the set of positive reals $\mathbb{R}_+$, or intervals such as $[0, 2\pi]$. Once we have constructed a few sets of

[2] It seems that a great many students would gladly identify with the hostile camp.

reals, we can in turn collect them into more complicated sets. To illustrate, we may want to consider the set

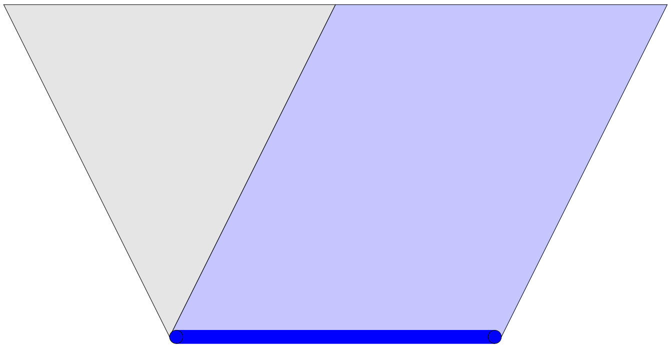
$$I = \text{all intervals } [a, a+1] \text{ of reals, } a \text{ an integer}$$

of closed unit-length intervals between two consecutive integers. Similarly we could form I_k, the collection of all intervals starting at an integer, but this time with length k, some positive natural number (so that $I = I_1$). At the next level, we can then collect all of these interval sets:

$$\mathcal{I} = \text{ all } I_k \text{ where } k \text{ is a positive integer}$$

And so on and so forth, we can pile up sets upon sets of increasing complexity. In typical applications, a handful of rounds is enough to obtain the sets we need. Still, at least in principle, we want to be able to continue the process forever; sets may be nested at arbitrary depth. Most humans find this process a bit daunting, but we will see that it is quite manageable, given the right kind of abstractions and definitions to hide gorey details.

In this scenario, where the reals are given ab initio, they are called urelements or atoms, basic objects that are somehow available right from the start, there is no need to worry about where they come from. This is quite similar to the situation in programming languages: certain basic types like `int` are built-in and we do not have to bother to implement them on our own, we can directly use them in data structures of our own design. Here is a picture that depicts the universe of sets over some collection of urelements.



The dark-blue bar at the bottom represents the reals and the expanding cone stands for all the sets we can build from there, level after level, as in the preceding example. The cone encompasses the universe of all sets, which is often written $\mathbb{V}$, indicating the V-shape in the picture. Note the gray part on the left: it is supposed to represent pure sets that contain no urelements whatsoever, at any level of the construction. For example, in the gray cone we could find $\emptyset$, $\{\emptyset\}$, $\{\{\emptyset\}\}$, $\{\emptyset, \{\emptyset\}\}$ and the like. As we have already seen, adjoining the natural numbers as urelements is unnecessary, we can reconstruct them fairly easily in the world of pure sets. They same is true for the reals, or really any standard mathematical object: we can always find a way to build (a representation of) the object from pure sets.

Extensionality Principle

What does it mean for two sets to be equal? Since sets are collections of their respective members, equality can be interpreted simply as having the same elements. This is known as the Extensionality Principle:

$$A = B \iff \text{for all } z\text{: } z \in A \text{ iff } z \in B$$

We will often abbreviate expressions like the right hand side somewhat more cryptically as $\forall\, z\, (z \in A \Leftrightarrow z \in B)$, using a universal quantifier rather than plain English; see the chapter 7 on logic below for details. Think of conducting a sequence of tests, trying to differentiate between A and B. The only questions we can ask is whether $z \in A$ or $z \in B$. If the answer to all these queries is the same, then the two sets are indistinguishable and hence identical. As a direct consequence of Extensionality, we can now justify our intuitive understanding that order and multiplicity should not matter for sets. It is only the elements that make a difference, not the way they are enumerated. Hence, $\{1, 2, 3\}$ is the same set as $\{3, 2, 1, 2, 1, 3\}$, no matter that our list notation suggests otherwise. Incidentally, from a computer science perspective, the lack of order is actually an impediment: it is much easier to deal with plain lists or trees.

Here is a slightly different take on extensionality. A set A is said to be a subset of a set B if every element of A is also an element of B: $\forall\, z\, (z \in A \Rightarrow z \in B)$. A subset A of B is a proper subset if $A \neq B$. In symbols: $A \subseteq B$ and $A \subseteq B$. We can also explain Extensionality in terms of subsets:

$$A = B \iff A \subseteq B \wedge B \subseteq A$$

This version suggests a proof method to establish equality of two sets: it suffices to prove the two inclusions separately, a method sometimes called double inclusion.

Extensionality has another important side-effect: it can be very difficult to establish equality even when one of the sets in question is trivial. For example, let $A = \{1, 2\}$ and B the set of all exponents n for which the Fermat equation $x^n + y^n = z^n$ has a solution over the integers. Then $A = B$, but the proof of this fact required centuries of development and can only be fully understood by a few number theory experts. A more recent example: ChatGPT answered the question "is $\mathbb{P}$ the same as PSPACE?" with a bold 'No', pointing out that $\mathbb{P}$ is the class of all problems that can be solved with a polynomial time algorithm, whereas PSPACE is based on polynomial space algorithms. In other words, intensionally the two classes are different. A bit later in the response, a hedge appeared, pointing out that it is currently unknown whether the two classes are the same—the bot had suddenly switched to the extensional form of the question.

The question of when two objects are identical may sound utterly banal, but, as we will see, it can be surprisingly problematic. In algebra, for example, one often quietly identifies structures that are isomorphic without being identical in the strict set-theoretic sense. For example, it seems silly to distinguish between the following two algebraic structures, one over the carrier set $\{0, 1\}$ and the other over $\{a, b\}$, given by their multiplication tables:

	0	1
0	0	1
1	1	0

	a	b
a	a	b
b	b	a

The reason we would want to identify the two structures is that there is an isomorphism between them: $0 \mapsto a$, $1 \mapsto b$. Algebraically, they are "the same," but their set-theoretic representations are different. In fact, we can build arbitrarily many copies of the same underlying structure by replacing 0 and 1 by arbitrary, distinct sets a and b. Another important example of questions involving identity arises in the context of computable functions, see chapter 8. For the time being, just think of a function as being computable whenever there is a program in some language like C that implements the function. If we think of a computable function strictly in set-theoretic terms as defined in chapter 3, in a purely extensional fashion, we ignore entirely the way in which the function is computed, the intensional view of the function. In computer science, the latter perspective is hugely important and the theory of algorithms has numerous examples of clever and efficient ways to compute particular functions. Yet, as far as the corresponding sets are concerned, there is no difference between a function computed by an exponential time algorithm versus a linear time algorithm. Perhaps surprisingly, the fact that every computable function can be implemented by infinitely many programs, is quite useful in various arguments in complexity theory.

The philosopher and logician W. V. O. Quine[3] captured the importance of understanding precisely what equality means in the slogan "*No entity without identity.*" For any class of mathematical object, we need to have a clear understanding of what it means that two of these objects are identical. Almost 3 centuries earlier, Gottfried Leibniz[4] already recognized the importance of equality and formulated the *principium identitatis indiscernibilium*: objects that are indistinguishable from each other are identical.

In the case of sets, the only method we have to distinguish between two sets is to perform membership queries: does such-and-such object belong to one set or the other. If all these queries produce the same answer, the two sets are already identical, even though they may have been defined in a completely different manner. For sets, we have a clear definition of equality.

Comprehension Principle

So far, the only way we can construct a set is to write an explicit list of the elements as in $\{1, 2, \triangle, \square\}$, or to use verbal descriptions as in the interval example above. We can stretch the list description a little bit and omit some of the elements if they are obvious from context. So we write

$$A = \{a_1, a_2, \ldots, a_{n-1}, a_n\}$$

where it is up to the reader to figure out what the missing elements are. This is perfectly fine in simple situations such as $A = \{1, 2, \ldots, 99, 100\}$ but it has its limitations: what should $A = \{3, 5, 7, \ldots, 104729\}$ mean? One can guess that A is supposed to be set of odd primes up to 104729, but that is a bit uncertain. We can even push our list approach to handle simple infinite sets along the lines of

$$E = \{0, 2, 4, \ldots, 2n, \ldots\}$$

Presumably E is intended to be the set of even natural numbers[5]. Clearly, the ellipsis method is too limited, we need a better way to form sets.

The idea behind set comprehension is that we use some predicate, some general property of potential elements, to filter out the ones that we want to be members of the new set. We write $P(x)$ for this predicate, where x is supposed to range over potential elements. If you prefer to think about $P(x)$ more syntactically, interpret it as a formula with free variable x. If we substitute a particular object a for x, we obtain an assertion $P(a)$ that is either true or false. For the time being, think of the range of the variable as being all of $\mathbb{V}$, the

[3] Quine also coined the slogan "To be is to be the value of a variable."

[4] Along with Isaac Newton, Leibniz is the co-inventor of calculus. Newton was a vastly superior mathematician, but Leibniz is arguably a much broader thinker, the last "universal genius."

[5] The ellipsis is easily the most abused symbol in math, always make sure to understand what is hidden there. Here there are two different kinds in one expression.

whole universe of sets, possibly with urelements such as the reals. This is usually written

$$A = \{\, x \mid P(x) \,\}$$

or, if all the x come from an already constructed set B,

$$A = \{\, x \in B \mid P(x) \,\}$$

We could now build $E = \{\, n \mid n \text{ even natural number} \,\}$ or $E = \{\, n \in \mathbb{N} \mid n \text{ even} \,\}$, assuming we already have built the set $\mathbb{N}$ of natural numbers. If $P(x)$ is always false, as in $x \neq x$, we get the empty set. If $P(x)$ expresses the notion that x is a natural number, we obtain the set of all naturals. And, if $P(x)$ is always true, as in $x = x$, we get the whole universe $\mathbb{V}$.

Returning to our interval example, we can give a fairly concise definition of the various sets involved:

$$\begin{aligned} [a,b] &= \{\, x \in \mathbb{R} \mid a \leq x \leq b \,\} \\ I_k &= \{\, [a, a{+}k] \subseteq \mathbb{R} \mid a \in \mathbb{Z} \,\} \\ \mathcal{I} &= \{\, I_k \mid k \in \mathbb{Z}, k \geq 1 \,\} \end{aligned}$$

We have written $\mathbb{Z}$ for the integers. Novices may find this sort of formal description harder to decipher than verbal renditions, but that is mostly a question of getting accustomed to the notation. After a while, things are often clearer and far less ambiguous in the formal description.

There is one piece of bad news: there are some technical problems with the unrestricted form of comprehension that we will address later. In a nutshell, we need to make sure that the collections do not become too large. For example, $\mathbb{V} = \{\, x \mid x = x \,\}$ seems like a perfectly reasonable collection of objects, and it is in many ways, but it cannot be treated like a set. These overly large collections are called (proper) classes and we will have more to say about them in chapter 6. On the plus side, the restricted form of comprehension, where only elements from some set B are chosen, is perfectly fine. For the time being, we will just occasionally mention issues with unrestricted comprehension.

That's All!

The ideas behind Extensionality and Comprehension are intuitively easy to grasp, but they provide enough power to construct all the sets we need and to reason about their properties. As already mentioned, we don't need urelements at all in our development. We can reconstruct all objects in mathematics as pure sets: integers, reals, continuous functions, vector spaces, computable functions, complexity classes, proofs and theorems; whatever, it all comes down to sets.

One last comment: we are not interested in exploring set theory as an independent and rich subject of mathematical logic. Our approach is purely mercenary: we use set theory to explain a few basic objects in mathematics and in particular in discrete mathematics, no more. In fact, we are much more interested in algorithmic aspects and will comment on those frequently. We will occasionally pretend that we can compute directly with sets, as described in the next section. Take this with a grain of salt, for us this is just a way to motivate some of the material.

Set Programming

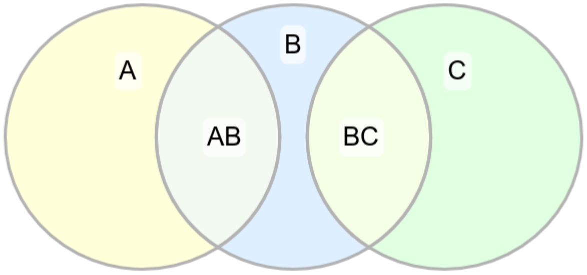
Sets may seem a bit overly cold and abstract, as opposed to warm and fuzzy notions like "triangle" or "$\sqrt{2}$". To make them a little more tangible, we will occasionally press an entirely fictitious programming language SETTRAN[6] into service. The language is very simple in that it supports only one datatype, namely set. SETTRAN has all sorts of built-in operations and control structures, but it can only operate on sets. We will be very casual about specifying the details of the language, just use your intuition and knowledge of far more pedestrian programming languages like C++ or Haskell. As an example, here is a simple SETTRAN program.

```
$x = \emptyset$
do 5 times
    $x = \mathsf{pow}(x)$
od
return $\mathsf{card}(x)$
```

Here pow is a built-in operation that returns the set of all subsets of its argument. Likewise, card is a built-in that returns the size of its argument (this is a much dicier subject, see section 5.3). At any rate, the program should return 65536. More precisely, it should return the set that represents that natural number, see section 1.1.4.

Visualization

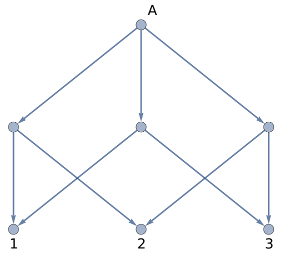
Sets and in particular pure sets are somewhat abstract, it may be helpful on occasion to use visualization to get a better feeling for sets and their interactions. Probably the most popular technique is Venn diagrams: one represents individual sets as regions in the plane in a way that overlapping regions correspond exactly to overlap between the corresponding sets. For example, for three sets A, B and C such that A and B overlap, B and C overlap, but A and C are disjoint (they share no elements whatsoever), we can use the following diagram:



This type of visualization may seem a bit frivolous, but it exploits geometric intuition and can occasionally be helpful in organizing rigorous arguments. Note that it is a moderately interesting exercise to construct a Venn diagram that displays all possible forms of overlap between 4 or 5 sets.

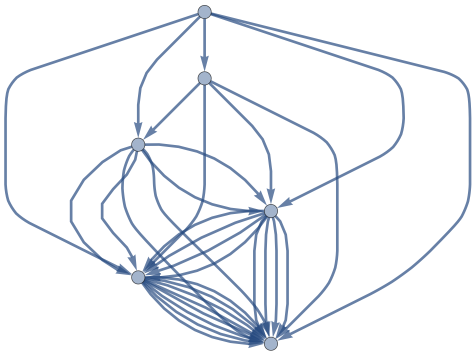
Venn diagrams express overlap between several sets, but is there a way to visualize just a single set? We can translate a set into a directed graph: the nodes are the set and its elements, and the element-of relation turns into directed edges: $x \in A$ corresponds to $A \to x$. If x is $\emptyset$ or an urelement, we are done; otherwise we apply the construction recursively. As an example, the set $A = \{\{1,2\},\{1,3\},\{2,3\}\}$ translates into

[6] Think IBM's Fortran or Conway's Fractran.

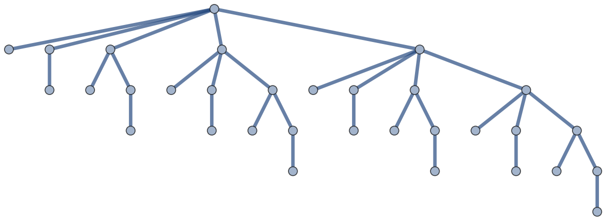


Unfortunately, for pure sets, there is only one possible leaf, the empty set. This results in many multiple edges and gets a bit unwieldy. Here is a slightly strange pure set (as we will later see, it is a very elegant representation of the natural number 5), together with its graph.

$$S = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}\}$$



If we count the edges ending at the leaf, we can see that there are 16 occurrences of $\emptyset$ in the set S, exactly as in the algebraic description above the image. A better representation can be obtained by unfolding the graph into a tree, without sharing subtrees as in our original attempt.



The tree picture brings out the recursive structure of the set S much better. Still, it should be clear that pictures are only mildly useful when dealing with sets, in particular when the sets in question are infinite.

1.1.3 Set Operations

Set comprehension is an all-purpose power tool to build sets, but in practice one usually can make do with much more concrete and intuitive operations.

Boolean Operations

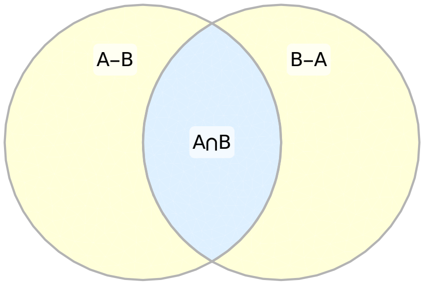
The most elementary set operations are based on selecting elements from two given sets.

$$\begin{aligned} A \cup B &= \{\, x \mid x \in A \text{ or } x \in B \,\} && \text{union} \\ A \cap B &= \{\, x \mid x \in A \text{ and } x \in B \,\} && \text{intersection} \\ A - B &= \{\, x \mid x \in A \text{ and } x \notin B \,\} && \text{difference} \end{aligned}$$

Thus, $A \cup B$ is the collection of all elements belonging to A or B, in the inclusive sense. We can see directly from the definition that $A \cup B = B \cup A$, $A \cup A = A$ and $A \cup \emptyset = A$. Similar laws exist for the other operations, a topic we will return to below. Occasionally, the symmetric difference also is useful:

$$A \oplus B = (A - B) \cup (B - A)$$

The inevitable Venn diagram for symmetric difference:



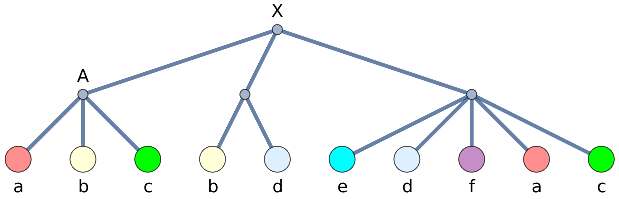
Note that we did not attempt to define "the complement of set A" in general. In order to have a general complementation operation, we need to fix some set $\mathcal{U}$ as the universe of discourse, and only consider sets A with elements from $\mathcal{U}$: we can then set $A^- = \mathcal{U} - A$. Complements of the form $\{\, x \mid x \notin A \,\}$ are too large and produce classes rather than sets.

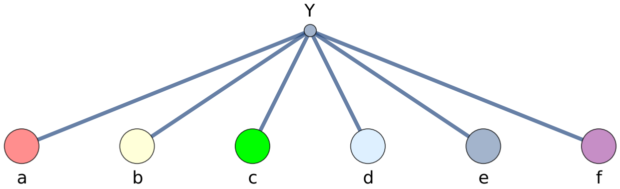
There are unary versions of union and intersection that work even when the number of arguments is infinite.

Definition 1.1.1 (Union and Intersection) *For any set X, define unary union and intersection operations as follows.*

$$\begin{aligned} \bigcup X &= \{\, z \mid \exists A \in X (z \in A) \,\} \\ \bigcap X &= \{\, z \mid \forall A \in X \, (z \in A) \,\} \end{aligned}$$

For union, the elements of the elements of X are collected and form the new set. For intersection, we collect only those elements of elements that appear in all the elements. As a sanity check we can verify that $A \cup B = \bigcup\{A, B\}$ and $A \cap B = \bigcap\{A, B\}$. From the definitions, $\bigcup \emptyset = \emptyset$. But $\bigcap \emptyset$ causes a problem: $\bigcap \emptyset = \{\, z \mid \forall x \in \emptyset \, (z \in x) \,\} = \mathbb{V}$. To avoid this kind of blow-up, we have to make sure that $X \neq \emptyset$ before forming $\bigcap X$. Visualizing sets as trees, we can see how level 2 elements are promoted to level 1. For $X = \{\{a, b, c\}, \{b, d\}, \{e, d, f, a, c\}\}$, $Y = \bigcup X$ looks like so.





We could implement unary union in SetTran with two nested loops:

```
$Y = \emptyset$
**forall** $A \in X$ **do**
    **forall** $z \in A$ **do**
        add $z$ to $Y$
    **od**
**od**
**return** $Y$
```

A typical example of the use of the unary intersection operator is to describe the smallest subset of a given set that has some particular property. For example, suppose we are interested in sets X of integers with the following property $P(X)$: they contain 0 and, whenever $z \in X$, then $z+3$ and $z+5$ are also in X. Now set

$$\mathcal{X} = \{\, X \subseteq \mathbb{Z} \mid P(X) \,\}$$

The collection $\mathcal{X}$ is not empty since certainly $\mathbb{Z} \in \mathcal{X}$. But our property is closed under intersection, in the sense that $P(x)$ and $P(y)$ implies $P(x \cap y)$. Hence $A = \bigcap \mathcal{X}$ must be the subset we are after. Note that from a certain perspective this argument is a bit circular: $A \in \mathcal{X}$, so we are using an object to define itself. This is called an impredicative definition, and nowadays is generally accepted practice in mathematics. Another issue with this argument is that it tells us nothing about the nature of A, but a little fumbling shows that

$$A = \{0,3,5,6,8,9,10,11,\ldots\} = \{0,3,5,6\} \cup \{\, z \in \mathbb{Z} \mid z \geq 8 \,\}$$

In this case, we have a constructive way of describing A, arguably a much better answer. Still, for mere existence our intersection operator is good enough.

One often writes the set X in a more suggestive form as a family of its elements:

$$X = (A_i)_{i \in I}$$

where I is an arbitrary index set and the A_i are themselves sets. Popular choices for index sets are $[n] = \{1, 2, \ldots, n\}$, $\mathbb{N}$, $\mathbb{Z}$ or even $\mathbb{R}$, but any set is fine as a matter of principle. In this setting, we can then write

$$\bigcup X = \bigcup_{i \in I} A_i$$

This sort of notation is quite standard in analysis. For example, we can construct a half-open interval of reals as a union of closed ones:

$$[0, 1) = \bigcup_{n \geq 1} [0, 1 - 1/n]$$

Here n is supposed to range over the positive integers.

Union, intersection and complement are directly based on propositional logic. It is not too surprising that these operations exhibit similar properties.

Lemma 1.1.1 *The following identities hold for all sets x, y, z. For the last four, we assume $x, y, z \subseteq \mathcal{U}$.*

- *Associativity*
  $x \cup (y \cup z) = (x \cup y) \cup z$ *and* $x \cap (y \cap z) = (x \cap y) \cap z$.
- *Commutativity*
  $x \cup y = y \cup x$ and $x \cap y = y \cap x$.
- *Distributivity*
  $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$ *and* $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$.
- *Identity*
  $x \cup \emptyset = x$ and $x \cap \mathcal{U} = x$.
- *Idempotence*
  $x \cup x = x$ and $x \cap x = x$.
- *Absorption*
  $x \cup (x \cap y) = x$ *and* $x \cap (x \cup y) = x$.
- *Domination*
  $x \cup \mathcal{U} = \mathcal{U}$ *and* $x \cap \emptyset = \emptyset$.
- *Complements*
  $x \cup x^- = \mathcal{U}$ and $x \cap x^- = \emptyset$.
- *Involution:*
  $x^{--} = x$.
- *De Morgan's Laws:*
  $(x \cup y)^- = x^- \cap y^-$ *and* $(x \cap y)^- = x^- \cup y^-$.

The Powerset

A set A is said to be a subset of a set B if every element of A is also an element of B: $\forall\, z\, (z \in A \Rightarrow z \in B)$. A subset A of B is a proper subset if $A \neq B$. In symbols: $A \subseteq B$ and $A \subset B$, or, for emphasis, $A \subsetneq B$.

Note that by Extensionality $A = B \iff A \subseteq B \wedge B \subseteq A$, so in order to establish equality of two sets it suffices to prove the two inclusions separately. From the definition, $\emptyset \subseteq A$ for any set A. We can now collect all subsets of a given set.

Definition 1.1.2 (Powersets) *The powerset of a set is the set of all its subsets. We write $\mathfrak{P}(A)$ for the powerset of A.*

This is justified by Comprehension, $\mathfrak{P}(A) = \{\, x \mid x \subseteq A \,\}$. For example, $\mathfrak{P}(\emptyset) = \{\emptyset\}$ and $\mathfrak{P}(\{a,b\}) = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$. We have not yet introduced any machinery dealing with ideas of size or measure of a set; for the next lemma we rely on the intuitive understanding of size for finite sets (incidentally, we have not yet defined what that means, either).

Lemma 1.1.2 *If A is a finite set with n elements, then $\mathfrak{P}(A)$ has 2^n elements.*

Proof. By induction on n. The base case $n = 0$ clearly is correct. For the induction step, suppose $A = \{a_1, a_2, \ldots, a_{n-1}, a_n\}$ where the tacit assumption is that the a_i are all distinct. Now let $x \subseteq A$. If $a_n \notin x$, then x is a subset of $\{a_1, a_2, \ldots, a_{n-1}\}$. There are 2^{n-1} such subsets by induction hypothesis. On the other hand, if $a_n \in x$, then $x = \{a_n\} \cup y$ where y is a subset of $\{a_1, a_2, \ldots, a_{n-1}\}$. Again, there are 2^{n-1} such subsets by IH, and these two collections do not overlap. The total number of elements is thus $2^{n-1} + 2^{n-1} = 2^n$, as required. □

Note that the proof provides a recursive algorithm to construct the powerset of a finite set. We can implement this algorithm in SetTran, assuming that we have some sort of recursions available in the language.

```
pow($A$):
    if $A = \emptyset$
    then return $\{\emptyset\}$
    pick $a \in A$
    $X$ = pow($A - \{a\}$)
    $Y = \emptyset$
    forall $x \in X$ do
        add $x \cup \{a\}$ to $Y$ od
    return $X \cup Y$
```

A much more interesting and much more difficult problem is to make sense out of the size of $\mathfrak{P}(A)$ when A is infinite. We postpone this discussion to section 5.3.

Pairs and Cartesian Products

One of our primary concerns is to find a clean way to define discrete objects such as lists of integers, matrices of rationals, words over an alphabet, directed graphs, and so on. A good starting point for this project is to find a way to implement lists[7]. Since sets are unordered and do not allow for repetitions, we cannot simply use the set of the list elements, not even for pairs, lists of length 2. We need to come up with a more complicated representation. As it happens, this is not entirely true: suppose A and B are two disjoint sets and we want to have lists where the first element is in A and the second in B. We could use the collection of all subsets of $A \cup B$ containing exactly two elements, one from A and the other from B. Alas, this method fails miserably when $A = B$, a rather important special case: just think about the two-dimensional plane in geometry.

We are looking for a pairing operation on sets: given two sets x_1 and x_2, we need to construct a compound set $\mathsf{pair}(x_1, x_2)$ that allows us to retrieve the components: there have to be un-pairing operations $\mathsf{unpair}_i(\mathsf{pair}(x_1, x_2)) = x_i$. The challenge is to implement these operations purely in terms of sets. Nowadays, the standard construction used for this purpose is due to K. Kuratowski.

[7] We will use the terms 'list', 'tuple', 'sequence' and 'vector' more or less interchangeable.

Definition 1.1.3 (Kuratowski 1921) *The (Kuratowski) pair of two sets x and y is the set*

$$\mathsf{pair}(x, y) = \{\{x\}, \{x, y\}\}$$

Just to be clear, there is nothing sacred about the Kuratowski pair, other possibilities exist. Here is a suggestion due to N. Wiener: $\mathsf{pair}(x, y) = \{\{\{x\}, \emptyset\}, \{\{y\}\}\}$. It works fine, it is just less elegant than Kuratowski's solution. To make sure that Kuratowski's operation works as intended, we have to prove the following lemma.

Lemma 1.1.3 $\mathsf{pair}(x_1, x_2) = \mathsf{pair}(y_1, y_2)$ *implies* $x_1 = y_1$ *and* $x_2 = y_2$*.*

One way of establishing this lemma is to define the corresponding unpairing operations explicitly. The usual mathematical description looks like so:

$$\mathsf{unpair}_1(z) = \begin{cases} \emptyset & \text{if } z = \emptyset, \\ \bigcup \bigcap z & \text{otherwise.} \end{cases} \qquad \mathsf{unpair}_2(z) = \begin{cases} \bigcup(\bigcup z - \bigcap z) & \text{if } \bigcup z - \bigcap z \neq \emptyset, \\ \pi_1(z) & \text{otherwise.} \end{cases}$$

For those who prefer a description that is closer to programming, here is an implementation in SetTran.

```
$\mathsf{unpair}_1(z)$:
    if $z = \emptyset$
    then return $\emptyset$
    else return $\bigcup \bigcap z$

$\mathsf{unpair}_2(z)$:
    $x = \bigcup z - \bigcap z$
    if $x = \emptyset$
    then return $\mathsf{unpair}_1(z)$
    else return $\bigcup x$
```

We leave the correctness proof to the exercises. To maintain some semblance of notational sanity, we will write

$$(x_1, x_2) \qquad \text{instead of the pedantic } \mathsf{pair}(x_1, x_2)$$

Given any two sets A and B, we can now form the collection of all pairs (a, b) where $a \in A$ and $b \in B$.

Definition 1.1.4 *The Cartesian product of two sets A and B is defined as*

$$A \times B = \{\, (a, b) \mid a \in A, b \in B \,\}$$

Incidentally, it follows from the definition that $A \times \emptyset = \emptyset \times B = \emptyset$, but in general $A \times B \neq B \times A$. Now that we have pairs, we can construct lists or k-tuples of length larger than 2 by recursively nesting our pairing operation. We let $\mathsf{list}(x_1, x_2) = \mathsf{pair}(x_1, x_2)$ and handle longer lists like so:

$$\mathsf{list}(x_1, x_2, \dots, x_k) = \mathsf{pair}(x_1, \mathsf{list}(x_2, x_3, \dots, x_k))$$

In other words, we use the right-associative version of pair. As before for pairs, from now on we simply write $(x_1, x_2, \dots, x_k)$ for a list of length k. So all lists of length 2 with elements from A are given by $A \times A$, length 3 by $A \times (A \times A)$, length 4 by $A \times (A \times (A \times A))$ and so on. We are really extending the Cartesian product operation to more than two arguments:

$$A_1 \times A_2 \times \ldots \times A_n = A_1 \times (A_2 \times \ldots \times A_{n-1} \times A_n)$$

More concisely, we can write $\prod_{i=1}^{n} A_i$, at least for $n \geq 2$. Naturally one would like to include the edge cases $n = 0, 1$. For $n = 1$, there is no problem: $\prod_{i=1}^{1} A_i = A_1$ makes sense, but a plausible answer for the case $n = 0$, corresponding to a set that contains only the empty list, requires a bit more groundwork. We will return to this question after we discuss functions.

1.1.4 Natural Numbers as Sets

Recall from the introduction our project of formally defining a Turing machine solely in terms of sets. In the last section we have seen how to handle lists, so it remains to find a way to represent the natural numbers or counting numbers as sets. These are perhaps the most intuitive concept in all of mathematics: we count, as in one, two, three and so on. As it turns out, it is convenient to include zero among the counting numbers [8], so, informally, one has the collection

$$\mathbb{N} : \quad \text{zero}, \text{one}, \text{two}, \text{three}, \ldots$$

Our goal is to define a particular set N_n for every natural number n in the enumeration above. We refer to these sets as numerals. Needless to say, the numeral N_n has to be different from N_m for all $n \neq m$. Apart from that, anything works as a matter of principle, but bear in mind that we also want to be able to express arithmetical operation such as addition, multiplication, comparison and so on.

The key idea is to fix a set for zero and then use a successor function defined on sets to obtain the other numerals. In the informal setting, "one" is the successor of "zero", "two" the successor of "one" and so on. E. Zermelo proposed the following, straightforward approach:

$$Z_0 = \emptyset \qquad Z_{n+1} = \{Z_n\}$$

More generally, we have

$$Z_n \rightsquigarrow \underbrace{\{\{\ldots\{}_{n+1}\underbrace{\}\ldots\}\}}_{n+1}$$

so that, for example, $Z_3 = \{\{\{\emptyset\}\}\}$ would represent the natural number three. The sets are very simple and one might also feel that $\emptyset$ is the most natural choice as the numeral for zero. How about arithmetic, though? One pleasant feature of Zermelo numerals is that the predecessor function is just union:

$$Z_{\mathsf{pred}(n)} = \bigcup Z_n$$

Given this fact, we can implement addition using a while-loop like so:

```
add($x, y$):
    while $x \neq \emptyset$
        $x = \bigcup x$
        $y = \{y\}$
    return $y$
```

[8] Though this addition is somewhat recent and happened during the 20th century.

Alternatively, we could rely on recursion.

```
add($x, y$):
    if $x = \emptyset$
    then $y$
    else $\{\mathsf{add}(\bigcup x, y)\}$
```

The less-than relation can also be handled by a while-loop.

```
less($x, y$):
    while $x \neq \emptyset$
        $x = \bigcup x$
        $y = \bigcup y$
    return $y \neq \emptyset$
```

This works, but Zermelo numerals have one rather awkward feature: with the exception of Z_0, their cardinality is always 1. It would seem more natural to have a numeral N_n with cardinality n. The question then is what the elements of N_n should be, and a reasonable answer is that they should be all the smaller numerals:

$$N_n = \{N_0, N_1, \dots, N_{n-1}\}.$$

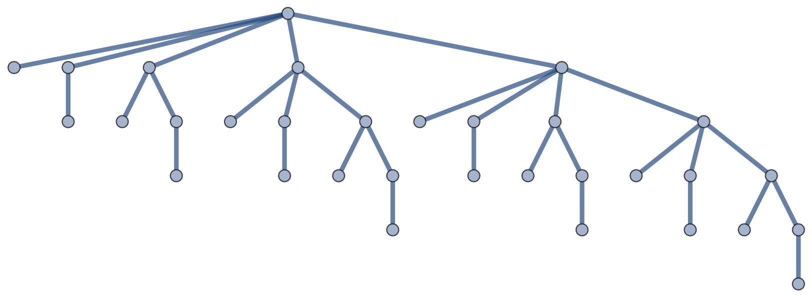
This is the idea behind a construction due to John von Neumann, and is nowadays entirely standard. These numerals are called finite von Neumann ordinals (ignore the ordinal part for the moment[9]). We retain $\emptyset$ as the representation for 0, and then modify the set-theoretic successor function to be $S(x) = x \cup \{x\}$.

$$N_0 = \emptyset \qquad N_{n+1} = S(N_n)$$

Von Neumann numerals are still finite at all levels (see the next section), but quite unwieldy when spelled out in detail. Very soon, one drowns in curly braces.

n	N_n
0	$\emptyset$
1	$\{\emptyset\}$
2	$\{\emptyset, \{\emptyset\}\}$
3	$\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$
4	$\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}$
5	$\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}\}$

Here is a picture of N_5, drawn as a tree that shows the recursive nature of these sets.



[9] Incidentally, von Neumann came up with this definition at the tender age of 19.

The root representing N_5 has 5 subtrees, from left to right corresponding to N_0 through N_4. The recursive structure of our numerals is fairly well visible in the picture. Note that predecessors are still given by $N_{\mathsf{pred}(n)} = \bigcup N_n$, so our programs from above would still work if we replace the old successor function $x \mapsto \{x\}$ by the new one, $x \mapsto x \cup \{x\}$.

If someone feels uneasy about "proof-by-SetTran," here is a more set-theoretic construction of addition for von Neumann numerals. In essence, we have to construct the set $A = \{\, (N_n, N_m, N_{n+m}) \mid n, m \in \mathbb{N} \,\}$, but without any reference to addition on the naturals (in the third component of the triples). To this end, define a set X to be *add-closed* if

- $(N_0, N_m, N_m) \in X$
- $(N_n, N_m, N_k) \in X$ implies $(S(N_n), N_m, S(N_k)) \in X$.

It is clear that $A \subseteq X$ for any add-closed set X. For example, $(N_0, N_5, N_5) \in X$, and therefore $(N_1, N_5, N_6) \in X$, then $(N_2, N_5, N_7) \in X$, and so on. However, there might also be fake elements such as (N_0, N_1, N_0). To exclude those, we take the intersection of all add-closed sets and get $A = \bigcap \{\, X \mid X \text{ add-closed} \,\}$. Multiplication and other arithmetical operations can be handled in a similar manner.

For some operations, though, we don't even need any constructions, they come for free:

$$\begin{aligned} N_m < N_n &\iff N_m \in N_n \\ N_{\max(m,n)} &= N_m \cup N_n \\ N_{\min(m,n)} &= N_m \cap N_n \end{aligned}$$

We still have not explained how to construct the natural numbers as a set of von Neumann numerals. Informally, we would like

$$\mathbb{N} = \{N_0, N_1, N_2, \ldots, N_n, \ldots\}$$

but that is not a definition. Indeed, upon closer inspection, it appears that we need the naturals to make this more precise, perhaps by writing $\mathbb{N} = \{\, N_n \mid n \geq 0 \,\}$. Of course, the last expression also presupposes the natural numbers and is entirely circular. The good news is that set theory makes it possible to remove this kind of circularity. We will use the same method as above for addition. Call a set X inductive if $\emptyset \in X$ and $x \in X$ implies $S(x) \in X$. Hence, any inductive set must contain N_0, therefore N_1, next N_2, and so forth. To eliminate fake entries such as $\{\{\emptyset\}\}$, we again use intersection:

$$\mathbb{N} = \bigcap \{\, X \mid X \text{ inductive} \,\}$$

Now we have a well-defined set and one can verify that it aligns perfectly well with all our intuitions about the natural numbers. If you are interested in a detailed description how this set can be built in a more formal axiomatic setting, see chapter 6.

1.1.5 Exercises

Exercise 1.1.1 Prove all the claims of lemma 1.1.1. OK, how about half of them?

Exercise 1.1.2 Prove that $A \subseteq B$ iff $\mathfrak{P}(A) \subseteq \mathfrak{P}(B)$.

Exercise 1.1.3 Show that $\mathfrak{P}(A) \cap \mathfrak{P}(B) = \mathfrak{P}(A \cap B)$.
Show that $\mathfrak{P}(A) \cup \mathfrak{P}(B) \subseteq \mathfrak{P}(A \cup B)$. How about equality?
Show that $\mathfrak{P}(A) - \mathfrak{P}(B) \subseteq \mathfrak{P}(A - B)$. How about equality?

Exercise 1.1.4 Let $\mathcal{X}$ be a family of sets. Show that $\bigcup \mathcal{X}$ is the least set X such that $\forall\, x \in \mathcal{X}\, (x \subseteq X)$. Prove something analogous for $\bigcap$.

Exercise 1.1.5 Show that the Wiener pairing function uniquely determines its arguments.

Exercise 1.1.6 Verify that the unpairing functions work as intended.

Exercise 1.1.7 Prove that lists of length $k \geq 3$ can be uniquely decomposed into their components.

Exercise 1.1.8 Show that the Wiener pairing function uniquely determines its arguments.

Exercise 1.1.9 Verify that the unpairing functions work as intended.

Exercise 1.1.10 Prove that lists of length $k \geq 3$ can be uniquely decomposed into their components.

Exercise 1.1.11 Prove some of the laws in lemma 1.1.1.

1.2 Discrete Objects

1.2.1 Hereditarily Finite Sets

The von Neumann numerals from the last section are a perfect example of a set representation of finite, discrete objects. More precisely, we are dealing with sets that are finite, all of whose elements are finite, whose elements in turn are finite, and so on, all the way down to the empty set (we are not going to use urelements, so the only indecomposable object is the empty set). The "all the way down" stipulation is supposed to exclude a set like $\{\{\mathbb{N}\}\}$ that is itself finite, has only finite elements, but contains an infinite set two levels down. The sets we have in mind are called hereditarily finite. There is a natural way to construct all hereditarily finite sets in layers. Recall that $\mathfrak{P}(x)$ denotes the power set of x.

$$\begin{aligned} \mathsf{HF}_0 &= \emptyset \\ \mathsf{HF}_{n+1} &= \mathfrak{P}(\mathsf{HF}_n) \\ \mathsf{HF} &= \bigcup_{n \geq 0} \mathsf{HF}_n \end{aligned}$$

The first 4 levels are easy to construct by hand:

$$\begin{aligned} \mathsf{HF}_0 &= \emptyset \\ \mathsf{HF}_1 &= \{\emptyset\} \\ \mathsf{HF}_2 &= \{\emptyset, \{\emptyset\}\} \\ \mathsf{HF}_3 &= \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\} \end{aligned}$$

Note that $\mathsf{HF}_n \subset \mathsf{HF}_{n+1}$. The cardinality of these sets grows wildly exponentially: $|\mathsf{HF}_{n+1}| = 2^{|\mathsf{HF}_n|}$. This rapidly growing function is called super-exponentiation and often written $2 \uparrow\uparrow n$, a stack of n-many 2s. Level 5 is still somewhat manageable at $2^{16} = 65536$ elements, but, at level 6, there are already about 2×10^{19728} sets, a fairly incomprehensibly large number.

There is a surprisingly easy way to enumerate the hereditarily finite sets. For simplicity, let's go in the opposite direction and encode every such set by a unique natural number.

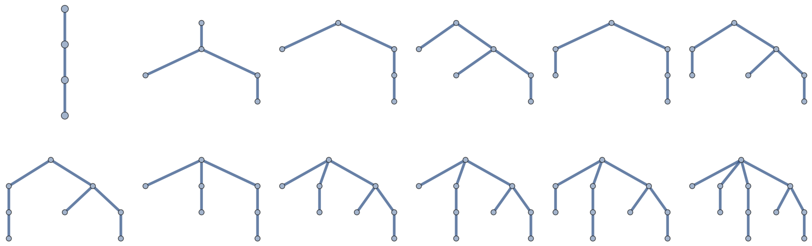
$$\begin{aligned}\alpha(\emptyset) &= 0\\ \alpha(x) &= \sum_{z\in x} 2^{\alpha(z)}\end{aligned}$$

For example, $\alpha(\{\emptyset, \{\emptyset\}\}) = 1 + 2 = 3$. In fact, applying α to all members of HF_n produces the whole initial segment $\{0, 1, \ldots, |\mathsf{HF}_n| - 1\}$. As a consequence, α is a bijection between HF and $\mathbb{N}$.

A good way to analyze the hereditarily finite sets in more detail is to consider their rank. The concept of rank is important in set theory in general, but we will only deal with hereditarily finite sets here. Informally, the rank of a set $x \in \mathsf{HF}$ is the nesting depth of sets in x.

$$\begin{aligned}\rho(\emptyset) &= 0\\ \rho(x) &= 1 + \max\bigl(\,\rho(z) \mid z \in x\,\bigr)\end{aligned}$$

For example, $\rho(\{\emptyset, \{\emptyset\}\}) = 2$. In general, HF_n has rank n and contains all the sets of rank less than n. As an example, in HF_5 there are $1, 1, 2, 12, 65520$ sets of rank 0 through 4, respectively. The next picture shows all the rank 3 sets in HF_4.



The first picture represents $\{\{\{\emptyset\}\}\}$, and the last one, $\{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}$.

1.2.2 Computing with Natural Numbers

Our construction of the natural numbers in the set theory universe in terms of von Neumann numerals is based on the zero-plus-successor approach: first define N_0 and then explain N_{n+1} in terms of N_n and a successor operation. the importance of this method goes far beyond providing a set-theoretic representation, of the naturals, it is the foundation for any discussion of computation. Sets matter little here, which is a good thing; after all, who really wants to compute with sets?

Let us first take a closer look at addition, arguably the most undemanding of arithmetical operations. In fact, everyone learns how to perform addition on arbitrarily large numbers in grade school. To add two natural numbers, we scan across their digits, starting at the least significant one (which, by convention, is the rightmost one) and determine sums and carries:

$$\begin{array}{ccccc} & 2 & 9 & 5 & 1\\ 5_0 & 3_1 & 1_1 & 8_0 & 7\\ \hline 5 & 6 & 1 & 3 & 8\end{array}$$

The subscripts for the second argument indicate the carries. This is the standard grade-school addition algorithm and is based entirely on manipulating digits: one needs to remember the sum and the carry produced by any two digits. Actually, two digits are handled by a so-called *half-adder*; since we also need to take into account the carry, we need to combine two half-adders into a *full-adder*.

Incidentally, the choice of base 10 seems to strike a good balance between limiting the size of the sum/carry lookup table and the number of digits needed to express natural numbers in a reasonable range. On the other hand, in computer science, other bases such as 2, 8 and 16 are perhaps more important. Base 2 in particular makes the lookup table very small; in fact, we can easily write down the lookup table for a full adder that handles not just the two digits to be added, but also the carry.

a	b	c	c'	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Since the digits here are just single bits, one can easily construct a digital circuit that computes c' and s as a function of a, b and c.

Unfortunately, this kind of digit manipulation does not carry over nicely into a different framework such as sets, we really need to explain addition on a more basic level. For example, how do we know that we would get the same result if we use different bases? In fact, what does "same result" even mean? We want a definition of addition that does not depend on a particular string representation.

In order to make addition more transparent we first need to make the natural numbers themselves more transparent, as strange as that may sound. We would like a description that is not based on strings of digits, decimal or otherwise. As it turns out, we can exploit two basic facts: first off, there is a unique smallest natural number 0. Second, given any natural number n, there is always a uniquely determined next natural number, called the successor of n. This number would normally be written $n + 1$, but that suggests that we already understand addition, so it is safer to write Sn, the smallest natural number that is larger than n. Informally, we have $3 = SSS0$. We refer to these objects as numerals in distinction to our intuitive understanding of natural numbers. Informally, our numerals are basically just numbers written in unary notation, but bear with me. Our numerals have two critical properties: first, we cannot have $Sn = Sm$ unless $n = m$. Second, 0 itself is not a successor. It would fly in the face of intuition to allow, say, $SS0 = S0$ or $S0 = 0$: in casual notation this would mean $2 = 1$ and $1 = 0$, respectively. Armed with these two basic facts, we can now attempt a first definition of the natural numbers: it is the set of all numerals.

Definition 1.2.1 (Natural Numbers)
The natural numbers are the smallest set $\mathbb{N}$ containing 0 and closed under successors in the sense that $n \in \mathbb{N}$ implies $Sn \in \mathbb{N}$.

Intuitively, this means that every natural number can be reached from 0 by finitely many applications of S, and can be so reached in exactly one way. This has an important consequence when one tries to prove properties of the naturals or define operations on them. Call a set X of numerals inductive if it contains 0 and is closed under successors: $n \in X$

implies $Sn \in X$ for all n. So $\mathbb{N}$ is the smallest inductive set. If you worry about how to make this construction a bit more explicit, take a look at section 1.1.4.

Theorem 1.2.1 (Induction Principle)
Let $X \subseteq \mathbb{N}$ be any inductive set. Then $X = \mathbb{N}$.

We will show an example of how to apply this induction principle to establish properties of the natural numbers in a moment. First, we need to define arithmetical operations and in particular addition. The mechanism behind our definition leans heavily on induction: we explain what to do when the first argument is 0, and then extend the definition from argument x to Sx. In the context of function definitions this method is usually called recursion.

$$\begin{aligned} \mathsf{add}(0, y) &= y \\ \mathsf{add}(Sx, y) &= S(\mathsf{add}(x, y)) \end{aligned}$$

Think of x and y as being two piles of pebbles. The first equations says that, if the x pile is empty, then the y pile is the result. The second equation stipulates that whenever the x pile is non-empty, we are to move one pebble from x to y and continue applying the same rules again. In some programming languages, these equations could be written almost verbatim to produce a syntactically correct program. Note that we have quietly used our assumption that Sx determines x uniquely, without it the left hand side of the second equation would not determine the right hand side. We can then calculate the sum of two and three as follows:

$$\mathsf{add}(SS0, SSS0) = S\big(\mathsf{add}(S0, SSS0)\big) = SS\big(\mathsf{add}(0, SSS0)\big) = SS\big(SSS0\big) = SSSSS0$$

Success, we have carefully reconstructed the obvious fact that $2 + 3 = 5$. The upside is that there are no more digits or lookup tables, everything we need are the two identities that define addition. This new addition algorithm is hopelessly tedious to apply, but logically it is far less complicated than the standard digit-based method.

One should make sure that this definition really works. The induction principle guarantees that our function $\mathsf{add}(x, y)$ is defined for all natural numbers. For let X be the set of all $x \in \mathbb{N}$ such that $\mathsf{add}(x, y)$ is defined, for all $y \in \mathbb{N}$. Then, by the first equation, $0 \in X$ and, by the second one, X is closed under S. By the induction principle, we get $X = \mathbb{N}$. So far, so good; we also need to make sure that our definition really determines addition uniquely, that there is only one way we can interpret to two equations. The next theorem says that we can combine two already existing operations g and h on numerals into a new one using recursion.

Theorem 1.2.2 (Definition by Recursion) *Given two functions h and g there is a unique function f that satisfies the recursion equations*

$$\begin{aligned} f(0, y) &= g(y) \\ f(Sx, y) &= h(x, f(x, y), y) \end{aligned}$$

In the case of addition, the auxiliary functions are very simple: $g(y) = y$ and $h(x, z, y) = S(z)$.

Proof. For simplicity, let us pretend that the parameter y is fixed; it is not hard to generalize the following argument. We consider functions ϕ whose domains are of the form $n^{\downarrow} = \{0, 1, \ldots, n-1\}$, the ancestors of n. Let us call two such functions *compatible* if they agree on the intersections of their domains (the ancestors of the minimum of the two numbers).

By induction, any two functions ϕ_1 and ϕ_2 that satisfy the recursion equations must be compatible. To see why, let $m^{\downarrow}$ be the common domain and set $X = \{\, x \in \mathbb{N} \mid \phi_1(x) = \phi_2(x) \lor x \geq m \,\}$. By the recursion equations, X is inductive and our two functions are compatible.

Now consider the collection $\mathcal{F}$ of all finite functions ϕ as above and let $f = \bigcup \mathcal{F}$. Because of compatibility, f is a function rather than just a relation. Moreover, the domain of f is all of $\mathbb{N}$ and f satisfies the recursion equations. □

This construction may seem a bit strange, but it is actually quite natural in the context of computation: the finite fragments $\phi : n^{\downarrow} \to \mathbb{N}$ are approximations to the function f we want to define. By applying the second recursion identity to these approximations, we can extend them to a larger domain. Taking into account all possible such extensions we ultimately get the function we are after. So this is the bottom-up style of recursion, often also referred to as dynamic programming.

Armed with the theorem, we can now define other arithmetical functions such as multiplication and factorials using similar recursions.

$$\begin{aligned}\mathsf{mult}(0, y) &= 0\\ \mathsf{mult}(Sx, y) &= \mathsf{add}(\mathsf{mult}(x, y), y)\end{aligned}$$

$$\begin{aligned}\mathsf{fac}(0) &= 1\\ \mathsf{fac}(Sx) &= \mathsf{mult}(Sx, \mathsf{fac}(s))\end{aligned}$$

Functions definable by a simple recursion as above are known as primitive recursive functions and play a major role in computability theory. As a matter of fact, it is quite difficult to come up with a computable arithmetical function that fails to be primitive recursive, try it. More on this later.

How about subtraction? Since we can only subtract smaller naturals from larger ones, one usually writes $x \mathbin{\dot{-}} y$ (pronounced "monus") instead of $x - y$, where informally

$$x \mathbin{\dot{-}} y = \begin{cases} x - y & \text{if } x \geq y\\ 0 & \text{otherwise.}\end{cases}$$

Can we define this operation by recursion? First we define the predecessor operation:

$$\mathsf{pred}(x) = \begin{cases} x - 1 & \text{if } x \neq 0\\ 0 & \text{otherwise.}\end{cases}$$

Formally, this comes down to

$$\begin{aligned}\mathsf{pred}(0) &= 0\\ \mathsf{pred}(Sx) &= x\end{aligned}$$

To get subtraction, we repeatedly apply this predecessor operation.

$$\begin{aligned}\mathsf{sub}(0, y) &= y\\ \mathsf{sub}(Sx, y) &= \mathsf{pred}(\mathsf{sub}(x, y))\end{aligned}$$

Lastly, we set $x \mathbin{\dot{-}} y = \mathsf{sub}(y, x)$.

One pleasant feature of our recursive definition of arithmetical operations is that they make it very easy to reason about the operations, again by induction. We will have much more to say about induction and recursive datatypes later, see section . For the time being, let us just go through a few sanity checks, informally $x+0 = x$, $x+Sy = Sx+y$ and associativity, $x+(y+z) = (x+y)+z$. For clarity, we refer to the two equations in the definition of add as add_1 and add_2.

rectypes

Claim 1: $\mathsf{add}(x, 0) = x$

Consider the set X of all y such that $\mathsf{add}(x, 0) = x$. We need to show that X is inductive.

Since $\mathsf{add}(0, 0) =_{\mathsf{add}_1} 0$ we have $0 \in X$.

Now assume $x \in X$, the so-called induction hypothesis. Then

$$\mathsf{add}(Sx, 0) =_{\mathsf{add}_2} S(\mathsf{add}(x, 0)) =_{\mathrm{IH}} Sx$$

and $Sx \in X$, as required.

Claim 2: $\mathsf{add}(x, Sy) = \mathsf{add}(Sx, y)$

Again, let X the set of all x such that the claim holds, we show X is inductive.

Since $\mathsf{add}(0, Sy) =_{\mathsf{add}_1} Sy =_{\mathsf{add}_1} S(\mathsf{add}(0, y)) =_{\mathsf{add}_2} \mathsf{add}(S0, y)$ we have $0 \in X$.

Now assume $x \in X$. Then

$$\mathsf{add}(Sx, Sy) =_{\mathsf{add}_2} S(\mathsf{add}(x, Sy)) =_{\mathrm{IH}} S(\mathsf{add}(Sx, y)) =_{\mathsf{add}_2} \mathsf{add}(SSx, y)$$

and $Sx \in X$, as required.

Claim 3: $\mathsf{add}(x, \mathsf{add}(y, z)) = \mathsf{add}(\mathsf{add}(x, y), z)$

Again, let X the set of all x such that the claim holds, we show X is inductive.

We have $0 \in X$ since $\mathsf{add}(0, \mathsf{add}(y, z)) =_{\mathsf{add}_1} \mathsf{add}(y, z) =_{\mathsf{add}_1} \mathsf{add}(\mathsf{add}(0, y), z)$.

Now suppose $x \in X$ and argue

$$\begin{aligned}
\mathsf{add}(Sx, \mathsf{add}(y, z)) &=_{\mathsf{add}_2} S(\mathsf{add}(x, \mathsf{add}(y, z))) \\
&=_{\mathsf{IH}} S(\mathsf{add}(\mathsf{add}(x, y), z)) \\
&=_{\mathsf{add}_2} \mathsf{add}(S(\mathsf{add}(x, y)), z) \\
&=_{\mathsf{add}_2} \mathsf{add}(\mathsf{add}(Sx, y), z)
\end{aligned}$$

Hence X is inductive and we are done.

Our formal version of addition behaves as expected. These arguments may appear plodding and overly mechanical, but they are not particularly complicated, induction drives every step. By contrast, try to prove associativity for the digit-based addition algorithm from the beginning of the section. It still requires induction (on the number of digits) and the combinatorial details are much more cumbersome.

1.2.3 Induction on the Naturals

Induction on the natural numbers is arguably the single most important application of the general idea of induction on rectypes, so we want to spend a little more time to discuss some of its features. First off, there is an issue of terminology that we should clean up. In the special case of the natural numbers, one often distinguishes between weak induction and strong induction. In the first case, we require $0 \in X$ and $x \in X \Rightarrow x{+}1 \in X$. The second case corresponds to our general notion of induction, X must contain n whenever it contains all ancestors $m < n$.

Claim: Weak induction and strong induction over $\mathbb{N}$ are equivalent.

First, suppose X is weakly inductive but not strongly so. Let a be the least counterexample, so that $\forall\, z < a(z \in X)$ but $a \notin X$. Then $a \neq 0$, so $a = b + 1$ and we have $\forall\, z \leq b(z \in X)$. In particular $b \in X$, so that $a = b + 1 \in X$, a contradiction.

Second, suppose X is strongly inductive but not weakly so. Since $\{\, z \mid z < 0 \,\} = \emptyset$ we must have $0 \in X$. Now let a be the least counterexample to $x \in X \Rightarrow x + 1 \in X$. But then $\forall\, z \leq a\,(z \in X)$, so $\forall\, z < a + 1\,(z \in X)$ and thus $a + 1 \in X$, a contradiction.

It is an ancient law that that the first example of induction is to establish the perfectly boring summation formula

$$P(n) \;\equiv\; \sum_{i=1}^{n} i = n(n+1)/2.$$

for all $n \in \mathbb{N}$ The proof is to show the property P is inductive and is often expressed in a rather too formulaic[10]manner like so:

base case:	show that $P(0)$
induction step:	show that $P(n) \Rightarrow P(n+1)$

In the induction step, the assumption $P(n)$ is referred to as the induction hypothesis (IH). Importantly, n is a free variable here, standing for a generic natural number, without any special properties whatsoever. In our summation example, the argument then comes down to a straightforward calculation:

$$\begin{array}{ll}
\text{LHS of } P(n{+}1) & \displaystyle\sum_{i=1}^{n+1} i \\
\text{split sum} & \displaystyle= \sum_{i=0}^{n} i + (n+1) \\
\text{IH} & = n(n+1)/2 + (n+1) \\
\text{RHS of } P(n{+}1) & = (n+1)(n+2)/2
\end{array}$$

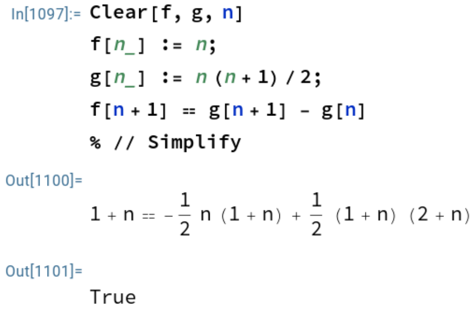
The ultimate reason this calculation works is that summation itself is defined recursively, hence we can split it as in the second line. There are many other examples along these lines, in general one would like to show that $\sum_{i=0}^{n} f(i) = g(n)$ where f and g are arithmetical functions. Expressed in terms of an induction argument, this comes down to showing

$$\begin{gathered}
f(0) = g(0) \\
f(n+1) = g(n+1) - g(n)
\end{gathered}$$

Again, in the second equation, n is a free variable, so purely numerical calculation will not suffice. Many computer algebra systems can verify this equation automatically as long as the functions f and g are not too complicated. In the following dialogue we translate

[10] The reason to complain about this approach is that beginners often lose sight of the actual goal and become lost in little mechanical manipulations when they try to follow the base-case/induction-step template.

the problem into a polynomial identity with free variable `n`, and symbolically verify that it holds.



```
In[1097]:= Clear[f, g, n]
           f[n_] := n;
           g[n_] := n (n + 1) / 2;
           f[n + 1] == g[n + 1] - g[n]
           % // Simplify

Out[1100]=
           1 + n == - 1/2 n (1 + n) + 1/2 (1 + n) (2 + n)

Out[1101]=
           True
```

Of course, we have to trust some fairly large and complicated machinery to interpret this as a proof. In this case, the associated algorithms have not been formally verified, but they have been used many millions of times in many different contexts, so there is some justification in assuming that they are correct, at least for fairly simple inputs. It is a bit embarrassing to use a computer algebra system this way, it is quite capable to find the summation identity all by itself. In fact, the system can simplify expressions that go far beyond the abilities of most mathematicians, a few experts exempted.

The Message

All the set representations we have talked about are of the proof-of-concept type, they are not meant as a practical and/or computational approach. No one ever computes using von Neumann numerals, the sets involved are just way too complicated. As a reminder, our reconstruction of the simple list of integers $(1, 2, 3)$ turns into the horror set

$$\{\{\{\{\{\emptyset\}\}, \{\{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}\}, \{\{\{\{\emptyset\}\}, \{\{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}\}$$

Again, we don't have to deal directly with this kind of set. Most of the time we rely on abstractions such as "number," "list," "graph" and so on. This is the only way to keep one's sanity. In fact, once we have checked that our intuitive assumptions hold up in the formal framework, we can essentially ignore the formal definitions and simply rely on our intuition and informal understanding. Surely open questions in number theory, say, the existence of infinitely many prime twins, can only be resolved without any reference to our set interpretation.

Things become much more complicated in classical continuous mathematics, in particular in analysis where real numbers are involved. There, set-theoretic foundations can and do play a major rule. For example, assuming the axiom of choice, one can construct sets of reals that fail to be Lebesgue measurable. On the other, one can also construct a set theory universe where all sets of reals are measurable. None of this is a concern to us in our discrete and countable world.

Aside

Von Neumann numerals have obvious advantages over Zermelo's numerals, but the real reason they are the standard choice of representation is that they extend easily into the transfinite realm. After all, the real strength of set theory is that it helps to clarify the nature of infinity. At any rate, in the context of ordinals, one often writes ω rather than $\mathbb{N}$ to indicate the first infinite ordinal. The point is that ω shares many of the properties of the N_n; for example, ω is the set of all smaller ordinals. And we can define $\omega + 1$ to be $S(\omega)$, $\omega + 2$ to be $S^2(\omega)$, and so on. It is not too hard to make sense of $\omega + \omega$, $\omega\omega$, ω^ω, and so on. So von Neumann's idea produces an arithmetic on transfinite numbers. This turns out to be very useful for lots of constructions in mathematics. We will return to ordinals in section 5.2.

1.2.4 Exercises

Exercise 1.2.1 Show that α is a bijection between HF and $\mathbb{N}$. Describe the inverse function $\alpha^{-1} : \mathbb{N} \to \mathsf{HF}$.

Exercise 1.2.2 Show that HF_n consists exactly of all sets of rank less than n. For this to be true, it is critical that we do not allow urelements.

Exercise 1.2.3 Count the number of sets of rank n.

Exercise 1.2.4 Figure out how the preceding tree corresponds to the given set.

Exercise 1.2.5 What is the rank of the set representing $(1, 2, \ldots, n)$?

Exercise 1.2.6 Implement the operations α and ρ from section 1.2.1. in SetTran, using von Neumann numerals.

Exercise 1.2.7 Construct Boolean circuits for the sum/carry table for binary addition. Try to make your circuit as small as ever possible.

Exercise 1.2.8 Check carefully that our definition of subtraction works as intended.

Exercise 1.2.9 Find a way to express exponentiation and GCD by recursion.

Exercise 1.2.10 Give a recursive definition of $\sum_{i=0}^{n} a_i$ where the a_i are arbitrary natural numbers.

Two

Relations

The last chapter gave a detailed explanation of the notion of a set and showed how represent a number of common mathematical objects as sets, and in particular natural numbers and lists. Our next goal is extend our representations to relations, and later to functions. The notion of a relation is a key concept in mathematics that revolve around connections, or the lack thereor, between separate entities. For example, two triangles may have the same area, a natural number may divide another, a real number may be smaller than another, three points may be collinear, a proof may establish a theorem, a Turing machine may halt on some particular input, and so on. Functions are special kinds of relation that we will address later. In order to describe a relation formally, we need to spell out which objects are involved in general, and precisely which pairs of objects are indeed related to each other.

2.1 Relations Defined

2.1.1 Relations and Cartesian Products

Relations come in many different types, the most basic one being a connection between two objects, a so-called binary relation. We will address the other forms later. Looking back at our development of set theory so far, it seems fairly straightforward to come up with a formal definition of this idea.

Definition 2.1.1 (Relations) *A binary relation ρ from A to B is a triple (ρ, A, B) where $\rho \subseteq A \times B$. A is the domain, B is the codomain and ρ is the graph of the relation, in symbols $\mathsf{dom}(\rho)$, $\mathsf{cod}(\rho)$ and $\mathsf{grf}(\rho)$. The domain of definition or support of the relation is the set $\mathsf{spt}(\rho) = \{\, a \in A \mid \exists\, b \in B\,(a \;\rho\; b)\,\}$ and the range (or image) of ρ is the set $\mathsf{rng}(\rho) = \{\, b \in B \mid \exists\, a \in A\,(a \;\rho\; b)\,\}$. A relation is total if its domain and its support coincide.*

If the domain and codomain of a relation are the same set A, we will speak of an endorelation or a relation on A. A is the carrier set or underlying set of the relation.

For example, the “less-than” relation on the set $A = \{1, 2, 3\}$ is given formally by the triple of sets

$$\big(\{(1,2),(1,3),(2,3)\}, \{1,2,3\}, \{1,2,3\}\big)$$

We will abuse notation slightly and simply write ρ instead of the full triple (ρ, A, B), domain and codomain are usually clear from context. Another standard piece of notation is to write $\rho : A \to B$ to indicate that ρ is supposed to be a relation from domain A to codomain B[1]. To express that a pair of elements is related by ρ we can write $(a, b) \in \rho$, $\rho(a, b)$ or $a \;\rho\; b$,

[1] Notation warning: Some authors refer to what we call domain of definition simply as “domain.” As a consequence, they do not have a name for the set A, a major inconvenience in our view.

using infix notation. The support and range of a relation are simply the projections of the graph on the first, respectively second, component: $\mathsf{spt}(\rho) = \{\, x \in A \mid \exists\, b \in B\,(a\ \rho\ b)\,\}$.

A few relations that appear frequently deserve their own notation. The identity or diagonal relation on A is the binary relation $I_A = \{\, (x,x) \mid x \in A \,\} \subseteq A \times A$, the universal relation is $U_A = A \times A$ and The empty relation is $\emptyset_A = \emptyset$. The last definition may look a bit strange, one might expect to find some reference to A on the right hand side. No such luck, there is only one empty set to go around. As usual, we omit the subscripts when no confusion is likely to arise.

Binary relations are arguably the most import type of relation, but there are others where more than just two objects are associated with one another. For example, one often wants to label the edges of a graph, a situation that can be modeled with a ternary relation $\rho \subseteq V \times \Sigma \times V$: V is the set of vertices and Σ the set of labels. More generally, we can use Cartesian products as discussed in section 1.1.3 to describe k-ary relations where $k \geq 1$: $\rho \subseteq A_1 \times A_2 \times \ldots \times A_k$. The special case $k = 1$ is really nothing new: a unary relation on a set A is just a subset of A. How about nullary relations? Since there are no arguments, there can be only two of those: one that is universally true and one that is universally false.

As an aside, we note that in database theory and in machine learning one naturally encounters relations of high arities. For instance, in order to create a student record, we may want to associate the student name, an id-number, an email address, an academic department, a college, a course name, a course number, a grade, and so on and so forth. In machine learning, we can think of unsupervised learning as an attempt to analyze relations of possibly high arity. These relations produce a number of interesting algorithmic challenges, but we will ignore them here.

Relations versus Functions

A relation is supposed to express some sort of connection between several objects, and it is natural to use pairs or k-tuples to express this connection. Alternatively we could think of a relation as some sort of query: given concrete objects, we can ask wether they are related to each other, or not. This query mechanism can be expressed in terms of a characteristic function, a $A \times B \to \mathbf{2}$. To wit, to model relation $\rho \subseteq A \times B$ we use the function $\mathsf{char}_\rho : A \times B \to \mathbf{2}$ defined by

$$\mathsf{char}_\rho(a,b) = \begin{cases} 1 & \text{if } a\ \rho\ b, \\ 0 & \text{otherwise.} \end{cases}$$

When A and B are both finite, the characteristic function can be though of as a Boolean matrix. On occasion, some insight in a relation can be gleaned from a plot of this matrix. For example, the relation "$1 \leq x, y \leq 100$ are coprime" is shown in figure 2.1.

K. Iverson introduced a device in the programming language APL that represents characteristic functions for relations, the Iverson bracket. Given any predicate $P(x, y, \ldots)$, the Iverson bracket $[P(x, y, \ldots)]$ evaluates to 1 if $P(x, y, \ldots)$ holds, and 0 otherwise. E.g., the prime counting function can be expressed like so: $\pi(n) = \sum_{x \in [n]} \mathsf{char}_{\mathsf{prime}}(x)$.

Relations versus Graphs

The gentle reader might wonder if there is any connection between a relation on A and a directed graph? After all, we have defined $\rho \subseteq A \times A$ to be the *graph* of the relation. Indeed, we could think of A as the vertex set and interpret $(x, y) \in \rho$ as a directed edge $x \to y$ in the combinatorial sense. And we could similarly translate a directed graph into a

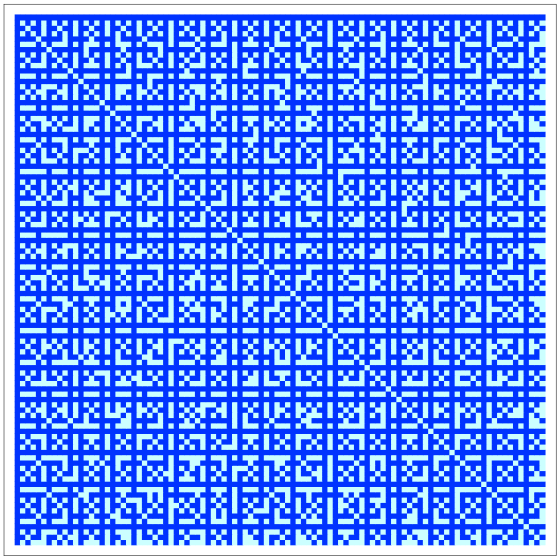
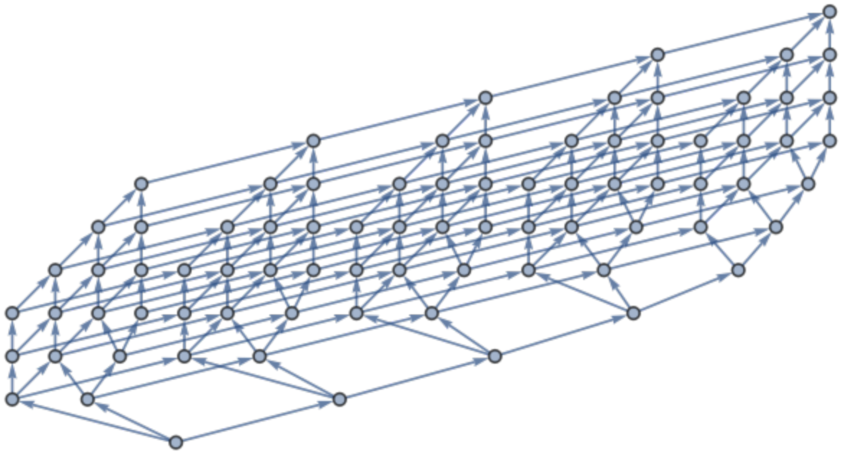


Figure 2.1: A visualization of coprimality for [100].

corresponding relation. Why bother with two concepts that are essentially the same? The answer is a little subtle, and probably not very convincing without some good amount of experience. The two concepts are indeed very similar, but they are used in distinct contexts. The word “graph” invokes images of points and arrows in the place or in space, it has a strong geometric flavor. One would not typically think of the “less-than” relation on the naturals as a graph, one thinks about arithmetic. One advantage of the similarity is that we can exploit graphs to produce images of relations, occasionally to great advantage.

For example, the following picture displays the divisibility relation between the divisors of 148176. It is a good exercise to try to figure out what some of the vertices are and exactly how divisibility is defined here.



One might object that graphs only make sense for endorelations, there is only one vertex set as opposed to separate and possibly distinct domains and codomains. This issue is not hard to circumvent, though: a relation $\rho : A \to B$ can always be thought of as an endorelation on the disjoint union of A and B. Perhaps a poor choice algorithmically, but logically everything is fine. At any rate, we will be fairly casual about the distinction between binary relations and graphs.

2.1.2 Operations on Relations

Boolean Operations

In analogy to the logical relations we have seen for sets, there are logical operations on relations. Suppose ρ and σ are two relations from A to B. We can then form the disjunction and conjunction of the relations, as well as the negation:

$$\begin{aligned} x\ (\rho \sqcup \sigma)\ y &\iff x\ \rho\ y \lor x\ \sigma\ y \\ x\ (\rho \sqcap \sigma)\ y &\iff x\ \rho\ y \land x\ \sigma\ y \\ x\ \rho^-\ y &\iff \text{not } x\ \rho\ y \end{aligned}$$

These operations are referred to as the join, meet and complement of the given relations. For example, let $\leq$ and $<$ be the usual order relations on the natural numbers. Then

$$\leq\ =\ < \sqcup I_{\mathbb{N}} \qquad <\ =\ \leq \sqcap I_{\mathbb{N}}{}^-$$

This identities look a bit bizarre, but there no reason why a relation should not appear in an equation, much as numbers could appear. Since we have defined the graph of a relation as a subset of $A \times B$ we can easily realize these operations in set theory: $\rho \cup \sigma$, $\rho \cap \sigma$ and $A \times B - \rho$. The latter is the total complement, we can also define a relative complement $\rho - \sigma = \rho \sqcap \sigma^-$. Then why introduce yet more notation and not simply reuse the set-theoretic operations? The reason it is better to introduce new symbols for join and meet will become clear in our discussion of equivalence relations below.

There is one other operation that is directly motivated by logic: interchange of the arguments. Let ρ be a binary relation from A to B. The converse relation of ρ, a relation from B to A, is defined by

$$x\ \rho^{\mathrm{op}}\ y \iff y\ \rho\ x$$

In terms of the graph, this comes down to reversing all the edges. Forming the converse is an example of an involution, an operation that, when applied twice, produces the original argument. In this case, $(\rho^{\mathrm{op}})^{\mathrm{op}} = \rho$. Returning to our example of order relations on $\mathbb{N}$, we have $<^{\mathrm{op}} = >$, $\leq \sqcap \geq\ = I_{\mathbb{N}}$ and $\leq \sqcap >\ = \emptyset_{\mathbb{N}}$.

Composition of Relations

Our operations on relations so far are fairly pedestrian. The next one is very important and forms the foundation for many other concepts and arguments. Consider relations ρ and σ with matching domains and codomains in the sense that ρ is from A to B, and σ from B to C.

Definition 2.1.2 (Composition of Relations) *The composition of ρ and σ, a relation from A to C, is defined by*

$$x\ (\rho \diamond \sigma)\ y \iff \exists\, z\, \big(x\ \rho\ z \land z\ \tau\ y\big)$$

The intermediate element $z \in B$ is a witness or certificate for the assertion that $(x, y) \in \rho \diamond \sigma$.

Early Warning

We have used natural, diagrammatic notation for $\rho \diamond \sigma$: apply ρ first, then σ, as expressed in the following diagram.

$$\begin{tikzcd}
A \arrow[r, "\rho"] \arrow[rr, bend right, "\rho\diamond\sigma"'] & B \arrow[r, "\sigma"] & C
\end{tikzcd}$$

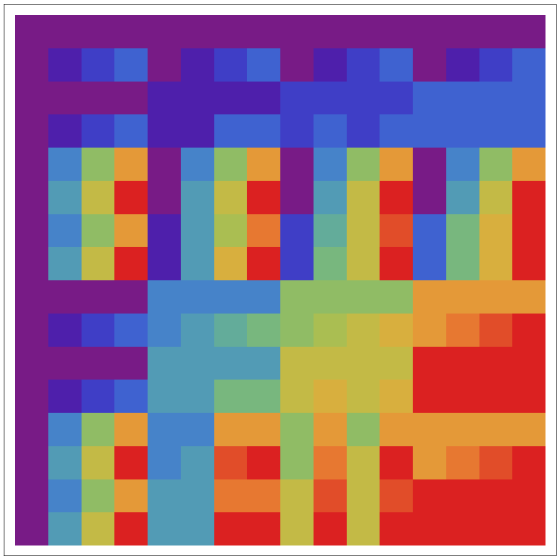
This should be easy to remember: diagrammatic as in diamond. It is quite natural in computer science where sequential composition of program is usually written `P; Q`, first `P`, then `Q`; Alas, there will be a bit of friction when we formally introduce the standard notation for composition of functions in the next chapter; there, $f \circ g$ is the function obtained by first applying g and then f so that $(f \circ g)(x) = f(g(x))$. These conflicting conventions are almost universal, there really is no way to avoid them.

Example 2.1.1 Let τ be the "parent of" relation. Then τ^{op} corresponds to "child of", $\tau \diamond \tau$ corresponds to "grandparent of", and $\tau^{\mathrm{op}} \diamond \tau$ corresponds to "sibling of" (more or less, people do not usually consider themselves to be their own siblings).

Example 2.1.2 Let τ be the divisibility relation on $\mathbb{N}$, $x \;\tau\; y \Leftrightarrow \exists\, z\,(x\,z = y)$. Then $\tau \diamond \tau = \tau$ and τ^{op} means "x is a multiple of y"

Example 2.1.3 Let $<$ be the standard "less than" relation on $\mathbb{N}$. This time $< \diamond < \subsetneq <$. However, if we switch the carrier set to $\mathbb{Q}$ or $\mathbb{R}$, $< \diamond <$ is the same as $<$.

There are $2^4 = 16$ relations on [2] and their multiplication table with respect to composition looks like so:



Now that we have enough operations on relations, we can start to accumulate basic facts about the way they interact. This can be done conveniently in terms of equations involving relations.

Lemma 2.1.1 *Suppose $\rho : A \to B$, $\sigma : B \to C$, and $\tau : C \to D$ are relations. Then*

1. $\rho \diamond (\sigma \diamond \tau) = (\rho \diamond \sigma) \diamond \tau$.
2. $\rho \diamond I_B = I_A \diamond \rho = \rho$.
3. $\rho \diamond \emptyset = \emptyset \diamond \rho = \emptyset$.
4. $(\rho \diamond \sigma)^{\mathrm{op}} = \sigma^{\mathrm{op}} \diamond \rho^{\mathrm{op}}$.

Proof. The proofs are quite straightforward from the definitions. For example, associativity of composition can be established as follows.

$$\begin{aligned} x\,(\rho \diamond (\sigma \diamond \tau))\,y &\iff \\ \exists\, u\, \big(x\ \rho\ u\,(\sigma \diamond \tau)\,y\big) &\iff \\ \exists\, u, v\, \big(x\ \rho\ u\ \sigma\ v\ \tau\ y\big) &\iff \\ \exists\, v\, \big(x\,(\rho \diamond \sigma)\,v\ \tau\ y\big) &\iff \\ x\,((\rho \diamond \sigma) \diamond \tau)\,y & \end{aligned}$$

The other arguments are left as an exercise. □

Returning to our graph model, we can think of a relation as expressing a single edge in a graph. It is often important to understand paths, sequences of edges that lead from one point to another. We can use composition of relations to express this idea in terms of relations.

Let ρ be an endorelation on A. The powers or iterates ρ^i, $i \geq 0$, of ρ are defined by

$$\begin{aligned} \rho^0 &= I_A \\ \rho^{k+1} &= \rho \diamond \rho^k \end{aligned}$$

In slight abuse of notation, we write a chain of related elements as

$$x_0\ \rho\ x_1\ \rho\ x_2 \ldots x_{n-1}\ \rho\ x_n$$

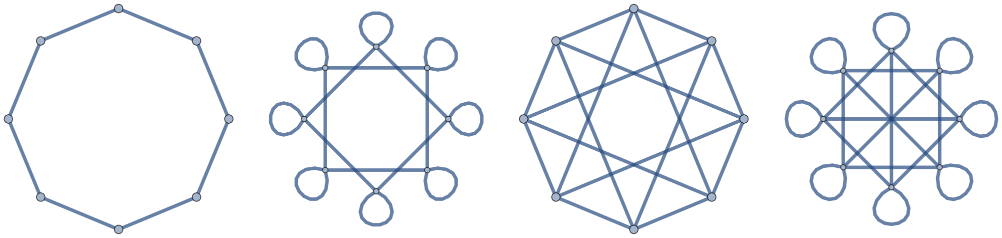
rather than the cumbersome conjunction $x_0\ \rho\ x_1$, $x_1\ \rho\ x_2$, $\ldots$, $x_{n-1}\ \rho\ x_n$. Then $x\ \rho^n\ y$ simply means that there is a chain of of related elements of length n:

$$x = x_0\ \rho\ x_1\ \rho\ x_2\ \rho \ldots \rho\ x_{k-1}\ \rho\ x_k = y.$$

Example 2.1.4 On $\mathbb{N}$, we have $x <^k y$ iff $x + k \leq y$, for all $k > 0$.

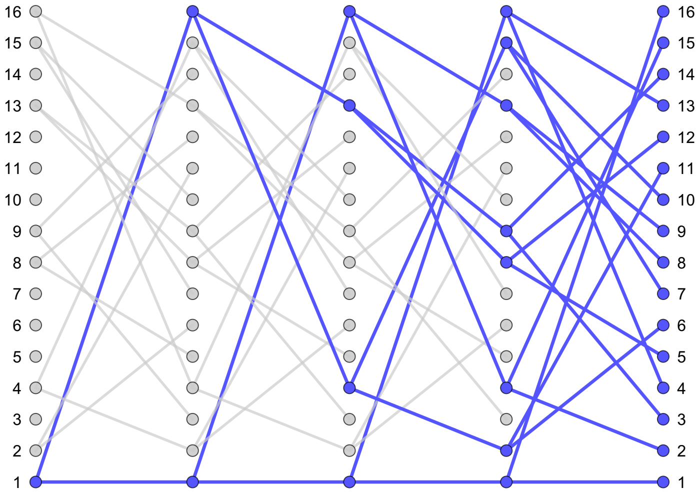
On the other hand, on $\mathbb{R}$ we get $<^k = <$, for all $k > 0$.

Example 2.1.5 Think of the cycle on 8 points as a relation. Below are the first 4 positive powers of this relation; the last two repeat forever.



For the next example, the visualization bears a little explanation. We are dealing with a relation on the carrier set [16]; this set is represented 5 times in 5 separate columns. The gray lines represent the relationships between the elements in adjacent columns. The blue lines trace all chains starting at point 1 in the first column.

Example 2.1.6 Consider multiplication modulo 17 on the carrier set [16]. Define $x \ \rho \ y$ to mean that x is a square root of y: $y^2 = x \pmod{17}$. Then $1 \ \rho^4 \ z$ for all z in [16].



The basic arithmetic laws of relational composition are fairly easy to state. Note that the first claim below is not just a repetition of the definition of iteration; there, we used the opposite order.

Lemma 2.1.2 *Suppose ρ is a endorelation on A. Then*

1. $\rho^n \diamond \rho = \rho^{n+1}$
2. $\rho^n \diamond \rho^m = \rho^{n+m}$
3. $(\rho^n)^m = \rho^{n \cdot m}$

Proof. For part one, use induction on n. The base case $n = 0$ follows from $\rho^0 = I_A$. For the induction step note

$$\begin{aligned} \rho^{n+1} \diamond \rho &= (\rho \diamond \rho^n) \diamond \rho && \text{def.} \\ &= \rho \diamond (\rho^n \diamond \rho) && \text{assoc.} \\ &= \rho \diamond \rho^{n+1} && \text{IH} \\ &= \rho^{n+2} \end{aligned}$$

□

Needless to say, these properties are exactly the same as for exponentiation of numbers: $x^n \cdot x = x^{n+1}$, $x^n \cdot x^m = x^{n+m}$, $(x^n)^m = x^{n \cdot m}$. This will be important later since it gives rise to a fast algorithm for computing relational closures (on finite sets).

Products of Relations

We mention one last operation on relations that amounts to a sort of product. This construction is useful for example in the study of finite state machines.

Definition 2.1.3 (Products of Relations) *Suppose ρ and σ be endorelations on A and B, respectively. The direct product of ρ and σ is defined by*

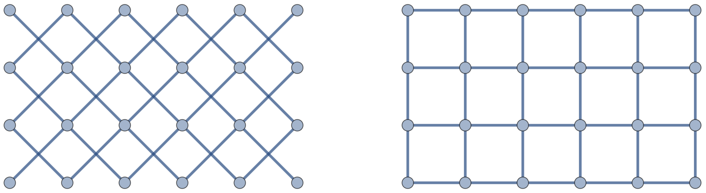
$$(a, b)(\rho \otimes \sigma)(a', b') \iff (a \; \rho \; a' \wedge b \; \sigma \; b').$$

The Cartesian product of ρ and σ is defined by

$$(a, b)(\rho \times \sigma)(a', b') \iff (a \; \rho \; a' \wedge b = b') \vee (a = a' \wedge b \; \sigma \; b').$$

The direct product is also called the tensor product or even the Kronecker product, see the next section.

As and example, let us define the path relation of length n, in symbols P_n, as the symmetric relation on $[n]$ defined by $x \; \rho \; y$ if $|x - y| = 1$. The direct and Cartesian product of P_4 and P_6 are shown below.



Note that the direct product on the left has two connected components. On the right, we have the standard 4×6 grid, all nodes are connected to each other.

Boolean Matrices

Relations are similar to directed graphs, and directed graphs can be represented by adjacency matrices. Naturally, we can also use them represent relations. Suppose ρ is an endorelation on $[n]$. The same construction works for any finite carrier set A, but it is a little bit easier to see what is going on if we restrict ourselves to $[n]$. We define an $n \times n$ Boolean matrix M_ρ or $\mathsf{M}(\rho)$ as follows:

$$\mathsf{M}_\rho(i, j) = \begin{cases} 1 & \text{if } i \; \rho \; j \\ 0 & \text{otherwise.} \end{cases}$$

Here we follow the usual hack in some programming languages, we write 0 instead of `false`, and 1 instead of `true`. So far, M_ρ is just a way to represent ρ; for this representation to be useful, we need to examine how operations of relations translate to the world of Boolean matrices. For join, meet and complement this is entirely straightforward, for instance:

$$\mathsf{M}(\rho \sqcup \sigma) = \mathsf{M}_\rho + \mathsf{M}_\sigma$$

where the plus on the right hand side is interpreted as bitwise or. Fine, but the reason this representation is truly useful is that composition of relations translates directly into multiplication of Boolean matrices.

How does one multiply two $n \times n$ Boolean matrices A and B? Call the product C, in symbols $C = A \cdot B$. The correct definition is

$$C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

In this definition, addition is to be interpreted as logical disjunction, and multiplication as logical conjunction.

Lemma 2.1.3 *Let ρ and σ be endorelations on $[n]$. Then $\mathsf{M}(\rho \diamond \sigma) = \mathsf{M}_\rho \cdot \mathsf{M}_\sigma$. Similarly $\mathsf{M}(\rho^k) = \mathsf{M}_\rho^k$.*

If this sounds a little abstract, think about the connection between relations and directed graphs. Say, $G = \langle [n], E \rangle$ is a directed graph on n vertices. If we think of E as a binary relation, the corresponding Boolean matrix $A = \mathsf{M}_E$ is the ordinary adjacency matrix of G. Thus, A represents all paths of length 1 in the graph. It is easy to see that $A \cdot A$ represents all paths of length 2: any such path must have the form $i \to k \to j$ and that will force a 1 in position (i, j) in A^2. By induction

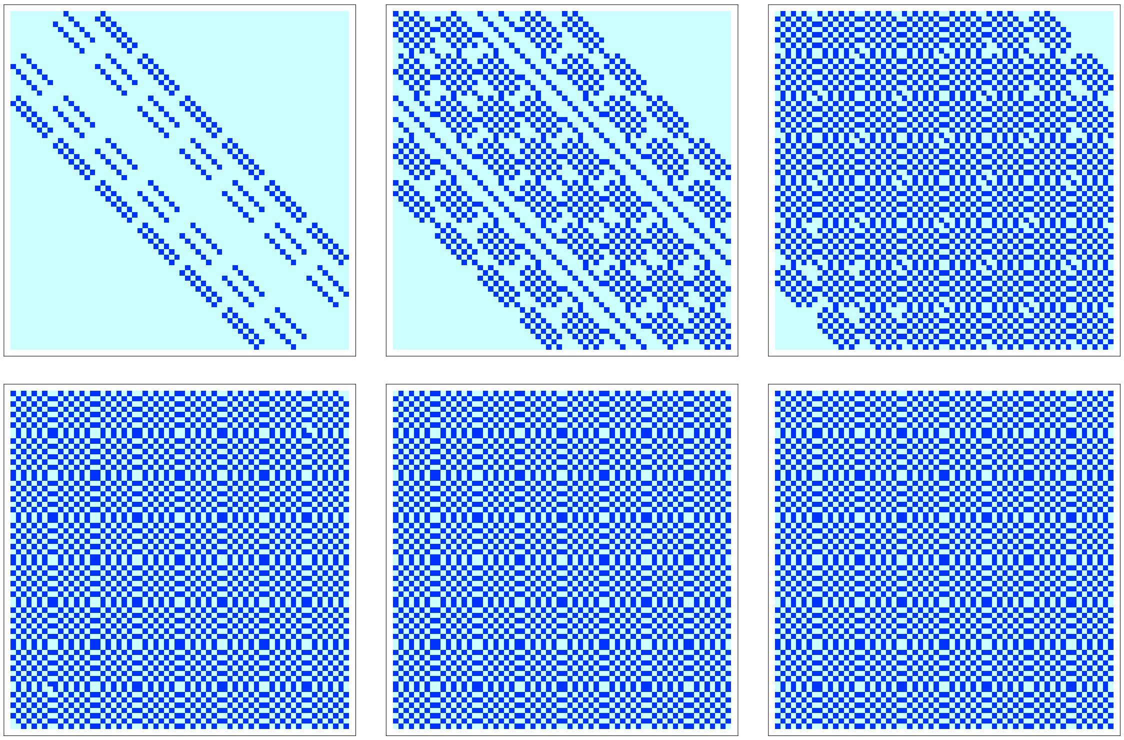
$$A^k_{ij} = 1 \iff \text{there is a path of length } k \text{ from } i \text{ to } j$$

Since any simple path in G has length at most n, we can set $B = \sum_{k=0}^{n-1} A^k$ to solve the reachability problem:

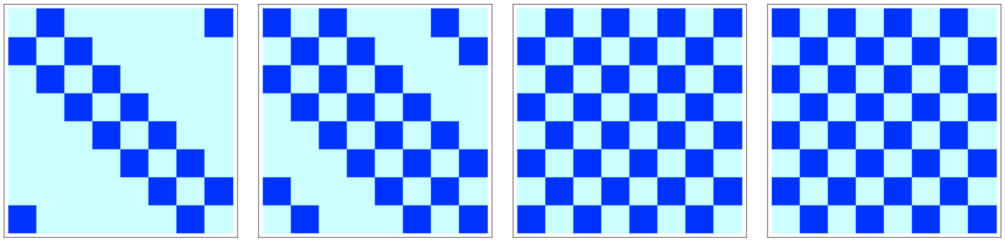
$$B_{ij} = 1 \iff \text{there is a path from } i \text{ to } j$$

So we can solve the reachability problem in graphs by arithmetic on Boolean matrices. This problem can also be tackled by graph exploration algorithms such as depth-first search or bread-first search. Using the standard matrix multiplication algorithm we can compute the product of an n by m and an m by p matrix in $O(nmp)$ steps. In particular for square matrices the standard algorithm is $O(n^3)$. There are faster algorithms, though, and we can perform matrix multiplication in about $O(n^{2.37})$ steps. Unfortunately, these asymptotically faster algorithms are difficult to implement and seem to require fairly large input matrices to outperform the standard algorithm.

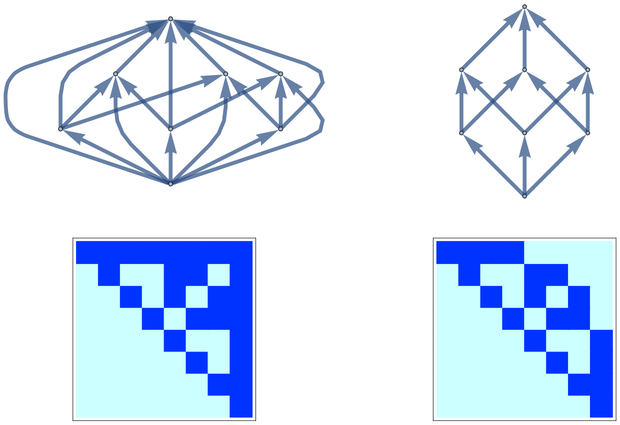
Example 2.1.7 Consider the moves of a knight on a chessboard. The carrier set is the collection of all 64 squares in standard lexicographical order, and $x \, \rho \, y$ means that a knight can move from square x to square y in one move. Here are the pictures for the first few powers of ρ. Note that the last two are complementary, so a knight can move from any position on the board to any other position.



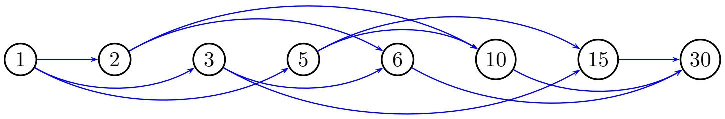
Example 2.1.8 Again the first 4 powers of the cycle C_8, this time as Boolean matrices.



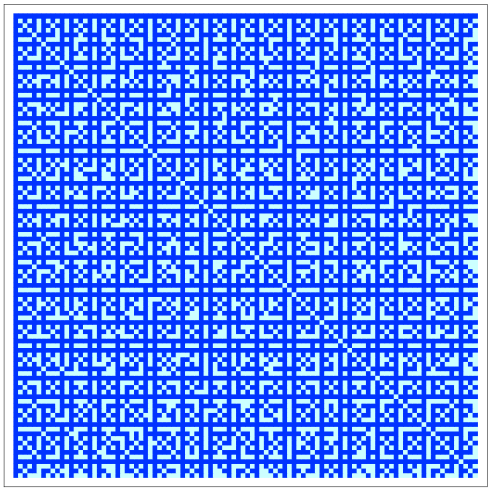
Example 2.1.9 The divisibility relation on the divisors of 30 is shown below. Self-loops are omitted in the graph images to avoid visual clutter.



On the left is the full relation, on the right we have only retained edges that cannot be inferred from others. We will come back to this reduction in our discussion of transitivity in section 2.3. Of course, there are other choices to visualize this relation:

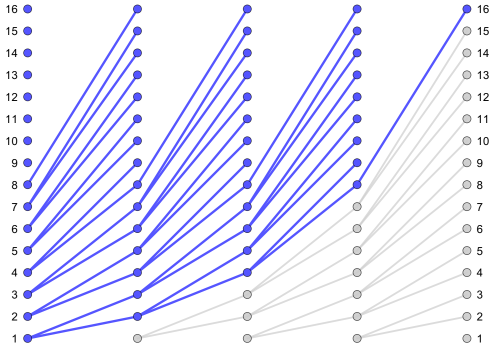


Example 2.1.10 The coprimality relation on [100] in Boolean matrix form.



The image suggests that there are some regularities in the distribution of coprime numbers, but overall the structure is rather complicated. For example, consider the following experiment: choose two numbers x and y at random, $1 \leq x, y \leq n$. What is the likelihood of x and y being coprime for large n? In other words, what is the density of 1's in the matrix? Surprisingly, for sufficiently large n the answer turns out to be about $6/\pi^2 \approx 0.6079$.

Example 2.1.11 Lastly, consider the relation $x \;\rho\; y \iff y \text{ div } 2 = x$ on [16]. Here is the picture for ρ^4.



From the picture, $1 \;\rho^4\; 16$, but no other elements are related under ρ^4.

The fact that composition of relations translates into multiplication of Boolean matrices is encouraging, but there are more reasons why this representation is interesting.

Definition 2.1.4 *Suppose we have an $n \times m$ matrix A and an $n' \times m'$ matrix B. The Kronecker product of A and B is the $nn' \times mm'$ matrix $C = A \otimes B$ defined by (we assume 0-indexing):*

$$C(i,j) = A(i \text{ div } n, j \text{ div } m) \cdot B(i \text{ mod } n', j \text{ mod } m')$$

The definition is a little opaque, it comes down to replacing each element a_{ij} in A by copy of B, multiplied by a_{ij}. For example, for A 2×2 and B 3×3 we get a 6×6 matrix consisting of 4 blocks:

$$\left(\begin{array}{c|c} a_{0,0}B & a_{0,1}B \\ \hline a_{1,0}B & a_{1,1}B \end{array}\right) = \left(\begin{array}{ccc|ccc} a_{0,0}b_{0,0} & a_{0,0}b_{0,1} & a_{0,0}b_{0,2} & a_{0,1}b_{0,0} & a_{0,1}b_{0,1} & a_{0,1}b_{0,2} \\ a_{0,0}b_{1,0} & a_{0,0}b_{1,1} & a_{0,0}b_{1,2} & a_{0,1}b_{1,0} & a_{0,1}b_{1,1} & a_{0,1}b_{1,2} \\ a_{0,0}b_{2,0} & a_{0,0}b_{2,1} & a_{0,0}b_{2,2} & a_{0,1}b_{2,0} & a_{0,1}b_{2,1} & a_{0,1}b_{2,2} \\ \hline a_{1,0}b_{0,0} & a_{1,0}b_{0,1} & a_{1,0}b_{0,2} & a_{1,1}b_{0,0} & a_{1,1}b_{0,1} & a_{1,1}b_{0,2} \\ a_{1,0}b_{1,0} & a_{1,0}b_{1,1} & a_{1,0}b_{1,2} & a_{1,1}b_{1,0} & a_{1,1}b_{1,1} & a_{1,1}b_{1,2} \\ a_{1,0}b_{2,0} & a_{1,0}b_{2,1} & a_{1,0}b_{2,2} & a_{1,1}b_{2,0} & a_{1,1}b_{2,1} & a_{1,1}b_{2,2} \end{array}\right)$$

Lemma 2.1.4

$$\begin{aligned} A \otimes (B \otimes C) &= (A \otimes B) \otimes C \\ A \otimes (B + C) &= A \otimes B + A \otimes C \\ (B + C) \otimes A &= B \otimes A + C \otimes A \\ (A \otimes B) \cdot (C \otimes D) &= (A \cdot C) \otimes (B \cdot D) \\ (A \otimes B)^{-1} &= A^{-1} \otimes B^{-1} \\ \mathsf{rnk}(A \otimes B) &= \mathsf{rnk}(A)\ \mathsf{rnk}(B) \end{aligned}$$

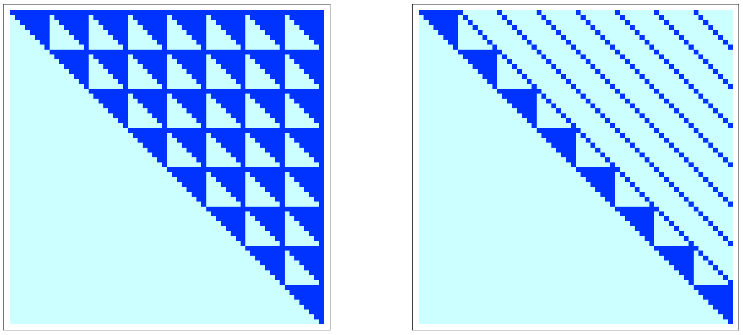
The reason the direct product of relations is also referred to as Kronecker product is the identity

$$\mathsf{M}(\rho \otimes \sigma) = \mathsf{M}_\rho \otimes \mathsf{M}_\sigma.$$

Similarly, the Boolean matrices for Cartesian products can be written as

$$\mathsf{M}(\rho \oplus \sigma) = \mathsf{M}_\rho \otimes I + I \otimes \mathsf{M}_\sigma$$

The next image shows the two products applied to the standard $\leq$ relation on $[8]$.



The carrier set is ordered by standard lexicographical order. Incidentally, the lexicographic order an $A \times A$ can similarly be defined by Kronecker produces: $\mathsf{M}_{\rho'} \otimes U + I \otimes \mathsf{M}_\rho$ where ρ' is the irreflexive version of the order ρ and U the universal relation.

Comparing Relations

There is a natural partial order on relations that explains the Boolean operations and has important applications in algorithms dealing with relations. Informally, we are interested in the situation where one relation implies another. For example, $x < y$ implies $x \leq y$.

Definition 2.1.5 *Let ρ and σ be two endorelations on A. Then ρ is finer than σ if $x \ \rho \ y$ implies $x \ \sigma \ y$ for all x and y. We also say that σ is coarser than ρ and write $\rho \sqsubseteq \sigma$.*

Note that we use "finer" in the sense of "strictly finer or equal" and likewise for "coarser." In this terminology, $\emptyset_A \sqsubseteq \rho \sqsubseteq U_A$ for any relation ρ on A.

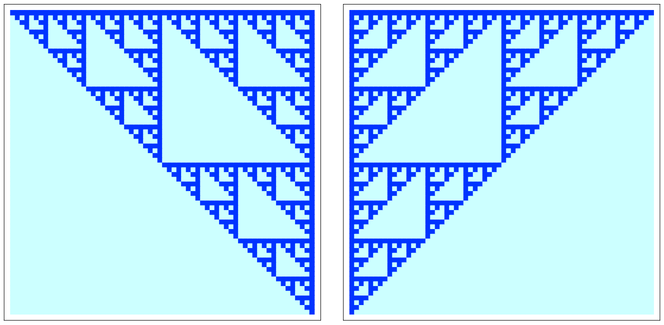
How does this relate to our Boolean operations? Certainly, $\rho, \sigma \sqsubseteq \rho \sqcup \sigma$. But more is true: $\rho \sqcup \sigma$ is the coarsest relation that refines both ρ and σ. Similarly, $\rho \sqcap \sigma$ is the finest relation that is coarser than both ρ and σ. The only relation finer than the total complement $A \times B - \rho$ is the empty relation, and the only one coarser is the universal relation.

There are several reasons why the idea of comparing relations is important. Abstractly, it provides a way to talk about relations in a more algebraic setting. Furthermore, iterative refinement of relations is an important algorithmic technique: we start with a rough approximation to the relation we want to compute, and then keep refining what we have, until we get to our goal. This method is usef for instance to minimize finite state machines.

2.1.3 Exercises

Exercise 2.1.1 Determine the number of relations from A to B when both sets are finite.

Exercise 2.1.2 Below are the Boolean matrix representation of two relations on binary lists of length $n = 6$. The first is $K \leq L$, lexicographic order on lists. The second is $K + L \leq \mathbf{1}$, again with pointwise comparisons..



Explain the pictures. What should one expect for other values of n?

Exercise 2.1.3 What do the divisibility pictures for arbitrary n look like.

Exercise 2.1.4 Find some natural examples of relations in geometry and physics.

Exercise 2.1.5 Finish the proof in lemma 2.1.1.

Exercise 2.1.6 Is composition of endorelations commutative?

Exercise 2.1.7 Show that for two endorelations ρ and σ: $(\rho \sqcup \sigma)^{\mathrm{op}} = \rho^{\mathrm{op}} \sqcup \sigma^{\mathrm{op}}$, $(\rho \diamond \sigma)^{\mathrm{op}} = \sigma^{\mathrm{op}} \diamond \rho^{\mathrm{op}}$ and $(\rho^{\mathrm{op}})^{-} = (\rho^{-})^{\mathrm{op}}$ and

2.2 Types of Relations

Functions will be introduced in chapter 3 and are perhaps the most important type of relation; other than that, there are two types of relations that turn out to be particularly important.

- Order relations: these express comparisons of some sort, like "less-than," "divides" or "contains."
- Equivalence relations: these express some kind of similarity, like "same-area," "same-length" or "congruent."

In this section we will describe these types of relations axiomatically and develop some tools for their classification.

Properties of Relations

In order to give coherent definitions of equivalence and order relations, we first discuss are a few basic properties of relations. These will turn out to be very useful in classifying relations.

Definition 2.2.1 *Let ρ be a endorelation on A. ρ is reflexive if $x \ \rho \ x$ for all x in A, and irreflexive if $\neg(x \ \rho \ x)$ for all x in A. ρ is symmetric if $x \ \rho \ y$ implies $y \ \rho \ x$ for all x, y in A, asymmetric if $\neg(x \ \rho \ y)$ or $\neg(y \ \rho \ x)$, and antisymmetric if $x \ \rho \ y$ and $y \ \rho \ x$ implies $x = y$. Lastly, ρ is transitive if $x \ \rho \ y$ and $y \ \rho \ z$ implies $x \ \rho \ z$.*

Alternatively we can characterize all these properties by using our calculus of relations based on logical operations and composition as follows.

property	condition
reflexive	$I \sqsubseteq \rho$
irreflexive	$I \sqcap \rho = \emptyset$
symmetric	$\rho^{\mathrm{op}} \sqsubseteq \rho$
asymmetric	$\rho \sqcap \rho^{\mathrm{op}} = \emptyset$
antisymmetric	$\rho \sqcap \rho^{\mathrm{op}} \sqsubseteq I$
transitive	$\rho \diamond \rho \sqsubseteq \rho$

Mote that "irreflexive" is not the negation of "reflexive" Make sure to convince yourself that this description is indeed equivalent to the original definition.

Example 2.2.1 Here is the classification of a few relations according to our taxonomy.

- equal-to, subset-of and divides are reflexive
- less-than, proper-subset-of and parent-of are irreflexive
- equal-to and coprime are symmetric
- less-than and parent-of are asymmetric
- less-than-or-equal, subset-of and divides are antisymmetric
- equal-to, subset-of, divides and ancestor-of are transitive
- parent-of and coprime are not transitive

2.2.1 Order Relations

Some relations such as the order relations on the natural numbers or rationals, or the subset relation on a power set are clearly similar in certain ways. What are the crucial properties that make them similar, and where do they differ?

Definition 2.2.2 (Order Relations)

Let ρ be a relation on A. ρ is a preorder if it is both reflexive and transitive. ρ is a partial order if it is a preorder and antisymmetric. ρ is a total order, linear order or simply an order if it is a partial order and all elements of A are comparable:

$$\forall\, x, y \in A\, (x\ \rho\ y \lor y\ \rho\ x).$$

A partial order ρ together with its carrier set A is often called a poset, written $\langle A, \rho \rangle$.

he comparability condition can also be written as $\rho \sqcup \rho^{\mathrm{op}} = U_A$. The orders that we have just defined are non-strict, they correspond to $a \leq b$. There is an analogous development for strict orders that correspond to $a < b$. A strict order is required to be irreflexive and transitive. In addition, we have to change antisymmetry to asymmetry and we have to replace comparability by trichotomy: $x\ \rho\ y \lor x = y \lor y\ \rho\ x$, equivalently $\rho \sqcup I_A \sqcup \rho^{\mathrm{op}} = U_A$.

Which version is more convenient depends very much on circumstances. At any rate, they are closely related and we can define one version from the other, by adding or removing the identity relation. Complements also take us from one kind to the other, $\leq^-$ is $>$. We will always follow the custom to indicate the distinction between strict and non-strict by the presence or absence of a little bar under the relation symbol: $\leq$ versus $<$, $\subseteq$ versus $\subset$. If we define some preorder $\preceq$, the relation $\prec$ is understood to be the strict version of $\preceq$, and conversely.

Example 2.2.2 The standard example for a poset that fails to be a linear order is the subset relation on the power set of some ground set B. If B has cardinality at least two, there are incomparable elements in this poset.

Example 2.2.3

- The usual order relations on numbers are total orders.
- Divisibility of natural numbers or polynomials is a partial order.
- Ordering polynomials by their degree produces a preorder.
- Cardinality is a preorder (and has full comparability).
- In geometry, “north-east-of” in the plane is a partial order.
- Lexicographic order on words is a total order.
- The substring order is a partial order on strings: $x \preceq y \iff \exists\, u, v\, (y = uxv)$.

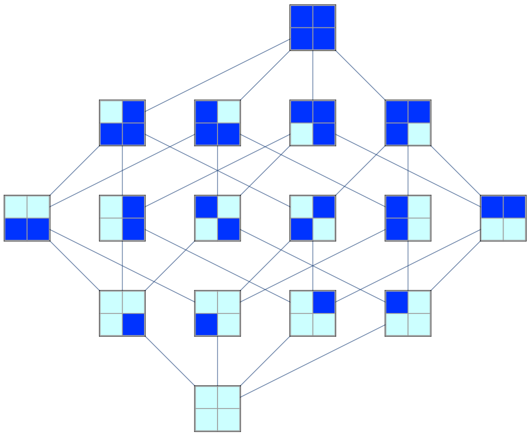
Here is bit more notation and terminology for posets that will come up frequently. We will often write $\perp$ for the least element of an order, and $\top$ for the greatest element (assuming they exist). An element x of a poset $\langle A, \leq \rangle$ is greatest if $z \leq x$ for all z in A. It is maximal if $x \leq z$ implies that $x = z$. The notions of least and minimal element are defined analogously.

Example 2.2.4 The poset $\langle \mathfrak{P}(A), \subseteq \rangle$ always has $\emptyset$ as its least element, and A as its greatest element. Now suppose we remove $\emptyset$ from this poset. Then all the singleton sets $\{a\}$, $a \in A$, are minimal.

Let $\langle A, \leq \rangle$ be a poset and $x \leq y$. Define the interval $[x, y]$ to be $\{\, z \in A \mid x \leq z \leq y \,\}$. For $x < y$ we say that y covers x if $[x, y] = \{x, y\}$. If y covers x it is also customary to call y a successor of x, and x a predecessor of y: there are no intermediate elements between x and y with respect to the given order. A poset is locally finite if every one of its intervals is finite.

The integers are the classical example for a locally finite order where every element has a successor and a predecessor; the rationals are the standard counterexample.

Example 2.2.5 Let A be the collection of all binary relations on $[2]$. The refinement relation $\rho \sqsubseteq \sigma$ is a partial order on A. To visualize this order, it is best to plot the relations as Boolean matrices and indicate only links to successors, rather than all of $\sqsubseteq$.



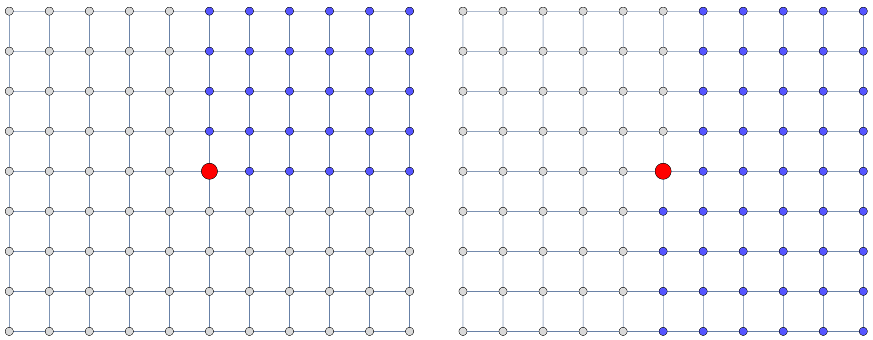
Ordering Pairs and Sequences

It is a standard problem to extend a given order on some set A to sequences of elements of A. For example, in geometry one often would like to order points in the plane in some reasonable fashion. Unfortunately, there is no easy way to lift the total order on the real numbers to a total order on points, i.e., on pairs of real numbers.

Definition 2.2.3 *Let $\langle A, \leq_A \rangle$ and $\langle B, \leq_B \rangle$ be two posets. We define two orders on $A \times B$ as follows. The product order on $A \times B$ is given by $(a_1, b_1) \leq (a_2, b_2)$ if $a_1 \leq_A a_2$ and $b_1 \leq_B b_2$. The lexicographic order is given by $(a_1, b_1) \leq (a_2, b_2)$ if $a_1 <_A a_2$ or $a_1 = a_2 \wedge b_1 \leq_B b_2$.*

These definitions use the two products of relations that we introduced in section 2.1.2; the product order is based on the direct product and the lexicographic order, on the Cartesian product. Note that $\langle A \times B, \leq \rangle$ is indeed a poset. However, even if both $\langle A, \leq_A \rangle$ and $\langle B, \leq_B \rangle$ are total orders, their product order will in general be a poset rather than a total order. The lexicographic order is total, on the other hand.

Example 2.2.6 Let $\leq$ be the product order of the standard order on $\mathbb{R}$ with itself. This produces the "north-and-east-of" relation in plane geometry: $\{ (x, y) \in \mathbb{Z}^2 \mid (0, 0) \leq (x, y) \}$ is the first quadrant. On the other hand, the lexicographic order $\preceq$ corresponds to "strictly-east-of-or-directly-below": $\{ (x, y) \in \mathbb{Z}^2 \mid (0, 0) \preceq (x, y) \}$ is the open right half-plane plus the ray starting at the origin and pointing down.



How about ordering all finite sequences over some set A? In particular when A is finite, one often refers to the finite sequences over A as words or strings over A. Write $|u|$ for the length of word u and ε for the empty word. Concatenation of words is expressed by juxtaposition, so xy stands for the word x immediately followed by word y. Suppose we have a total order $<$ on A. In the examples below we define a few popular orders on words. We say that u precedes v in the

1. Prefix order if $v = ux$ for some word x.
2. Lexicographic order if $v = ux$ for some word $x \neq \varepsilon$ or if $u = xay$, $v = xbz$ and $a < b$, where x, y, z are words, $a, b \in A$.
3. Length order if $|u| < |v|$.
4. Length-lex order if $|u| < |v|$ or $|u| = |v|$ and u precedes v lexicographically.

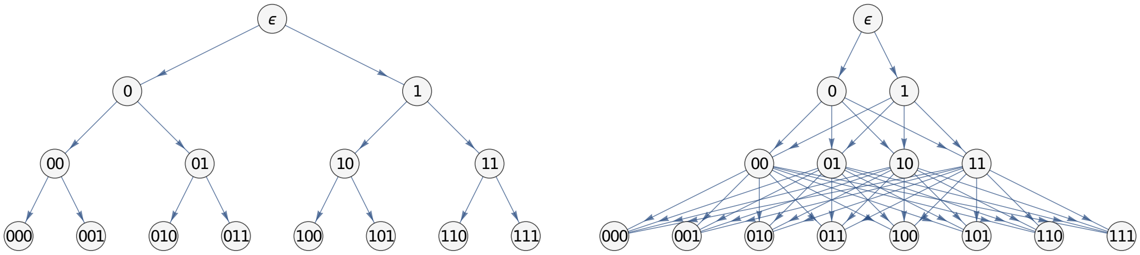
There are no generally agreed upon symbols for these orders; in these notes we will use $x \leq_\ell y$ for lexicographical order and $x \leq_{\ell\ell} y$ for length-lex order. Prefix order is a partial order and length order is a preorder (unless the alphabet has size 1). On the other hand, lexicographic order and length-lex order are total orders on words. Lexicographic order is well-known from dictionaries for natural languages. One of the reasons to prefer lexicographic order in this context that it is often possible to find a word even when its precise spelling is unknown. Length-lex order is the basis for many algorithms on strings. On words over the alphabet $\{0, 1\}$, length-lex order produces

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111$$

In lexicographic order, these few words would be arranged as

$$\varepsilon, 0, 00, 000, 001, 01, 010, 011, 1, 10, 100, 101, 11, 110, 111$$

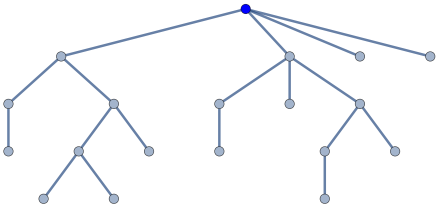
Prefix order and length order on these words are visualized in the next image.



Thus, the all words of length k appear at level k in either case, but prefix order produces a binary tree, whereas length order has all possible edges from level to level.

Trees as Posets

In the chapter on graphs there is a definition of rooted trees in terms of pointed directed graphs. These are often visualized by putting the root on top, its children one level down, their children another level lower, and so on. Undirected edges work fine in this setting, the tacit assumption is that they are all pointing downward.



Here is an alternative definition of rooted tree that is based on partial orders.

Definition 2.2.4 (Poset Trees) *A tree is a poset T with a least element $\bot$, called the root, such that for all elements x the interval $[\bot, x]$ is a finite total order.*

This definition clashes mildly with convetion to draw trees with the root on top, but that is a minor issue. It is clear that the maximal elements of T are the leaves of the tree, all others are interior nodes. The order relation is allowed to branch when moving from smaller to larger elements, but not the other way around. Hence, from any element x, there is a unique chain

$$x = x_0 > x_1 > x_2 > \ldots x_{n-1} > x_n = \bot$$

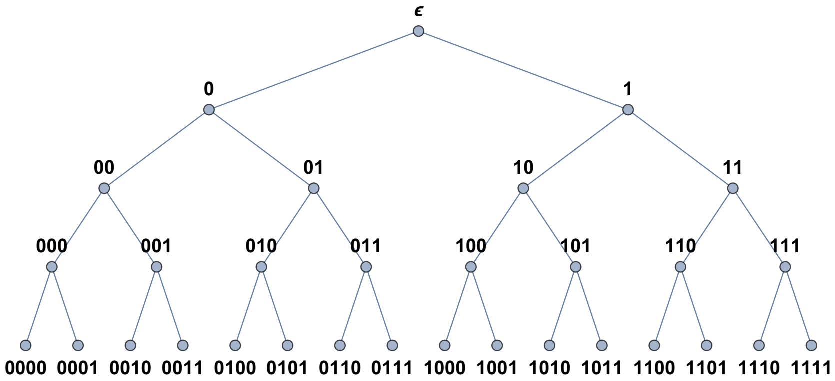
where x_i covers x_{i+1}, and n is the depth of x. Thus, the depth of element x in T is the length of the longest chain from r to x. The only real difference between the graph definition and the poset definition is that the first focuses on successors, whereas the second starts with a transitive relation. In some cases, this is a bit more natural.

Example 2.2.7 Think of $\mathbb{N}$ as an infinite alphabet and consider $\mathbb{N}^\star$, the set of all words over $\mathbb{N}$, together with prefix order. It is clear that ε is the least element in this poset, and the interval $[\varepsilon, x]$ for $x = x_1, \ldots, x_n$ has the form

$$\varepsilon, x_1, x_1x_2, x_1x_2x_3, \cdots, x_1, \ldots, x_{n-1}, x_1, \ldots, x_n.$$

Thus the length of a sequence in $\mathbb{N}^\star$ is its depth in this tree. Every node in this tree has infinitely many successors.

We will have more to say about trees later on, for the time being we mention just one important result. A tree is finitely branching if every node has only finitely many successors: for every x there are only finitely many y that cover x. For example, by restricting the alphabet in the last example to $\{0, 1\}$ we get complete binary tree.



Lemma 2.2.1 (Kőnig's Infinity Lemma) *Suppose T is an infinite but finitely branching tree. Then T contains an infinite branch.*

Proof. Let x_0 be the root of T and $y_1, \ldots, y_k$ all its successors. Since the tree located at x_0 is infinite, there must be at least one y_i so that the subtree with root y_i is also infinite. Set $x_1 = y_i$, and continue inductively. □

The proof method is highly non-constructive; in general we have no effective way to pick an appropriate successor for the current node.

2.2.2 Equivalence Relations

Equivalence relations formalize the notion of two objects being similar in a certain sense while ignoring irrelevant details. For example, we might consider two triangles as similar if one can be moved on top of the other by a rigid motion of the plane; or we could consider triangles similar if they have the same area. Two words could be considered similar if they have the same length, or if they have the same number of occurrences of the letters in the alphabet.

Definition 2.2.5 *A relation ρ on A is an equivalence relation if ρ is symmetric, reflexive, and transitive. Suppose ρ equivalence relation. The equivalence class of $a \in A$ or the block of a is defined by*

$$[a]_\rho = \{\, x \in A \mid a \;\rho\; x \,\}.$$

The collection of all such classes, $A/\rho = \{\, [a]_\rho \mid a \in A \,\}$, is called the quotient set or the quotient of A by ρ. The cardinality of A/ρ is the index of ρ.

If there is no danger of confusion we usually write $[a]$ instead of $[a]_\rho$. The index is particularly interesting when the quotient is finite, even though the carrier set is infinite.

Example 2.2.8 Both the equality relation I_A and and the universal relation U_A are equivalence relations. Their equivalence classes are trivial: $[a]_{I_A} = \{a\}$ and $[a]_{U_A} = A$.

Example 2.2.9 Here is a list of some equivalence relations.

- same-weight is an equivalence relation on dumbbells
- same-parents is an equivalence relation on humans
- same-cardinality is an equivalence relation on sets
- same-area is an equivalence relation on polygons
- same-input-output is an equivalence relation on programs

Example 2.2.10 Modular Arithmetic

The following equivalence relation is the foundation for many cryptographic methods such as RSA, and was first studied in great detail by Gauss.

$$x \equiv_m y \iff m \text{ divides } x - y$$

where $m \geq 0$ is some fixed integer, the so-called modulus. To emphasize the idea of replacing ordinary equality by a different equivalence relation one often writes $x = y \pmod{m}$ or $x \equiv y \pmod{m}$ to express that x and y are related by $\equiv_m$. $\equiv_m$ has index m.

For any equivalence relation we always have $a \in [a]$ by reflexivity. Furthermore, $[a] = [b]$ iff $a \; \rho \; b$. Here is the single-most important property of equivalence classes.

Lemma 2.2.2 *Let ρ be an equivalence relation on A. For all $a, b \in A$:*

$$[a] = [b] \quad \textit{or} \quad [a] \cap [b] = \emptyset.$$

Proof. If $c \in [a] \cap [b]$, then by symmetry for any $z \in [a]$: $z \; \rho \; a \; \rho \; c \; \rho \; b$. Hence $z \in [b]$ by transitivity. But then $[a] \subseteq [b]$. By a symmetric argument, $[b] \subseteq [a]$, done. □

By the lemma, all blocks in an equivalence relation are disjoint. Thus, they form a partition of the carrier set.

Definition 2.2.6 *A partition of a set A is a list $P = (P_i)_{i \in I}$ of non-empty subsets of A are pairwise disjoint, and whose union is A: $P_i \neq \emptyset$ for all i, $i \neq j \to P_i \cap P_j = \emptyset$ and $\bigcup_{i \in I} P_i = A$. The P_i are called the blocks of the partition.*

Lemma 2.2.3 *Equivalence relations correspond exactly to partitions.*

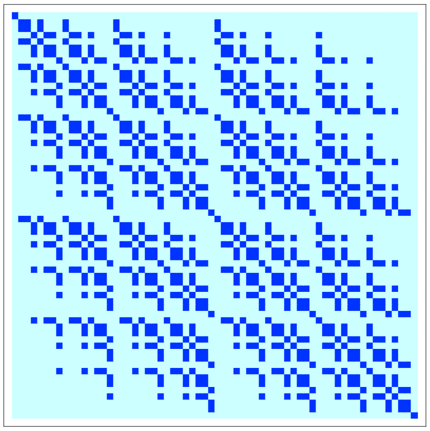
Proof. Suppose ρ is an equivalence relation. The equivalence classes $[a]$ of ρ produce a partition by the last lemma. On the other hand, suppose $P = (P_i)_{i \in I}$ is a partition. Define a relation

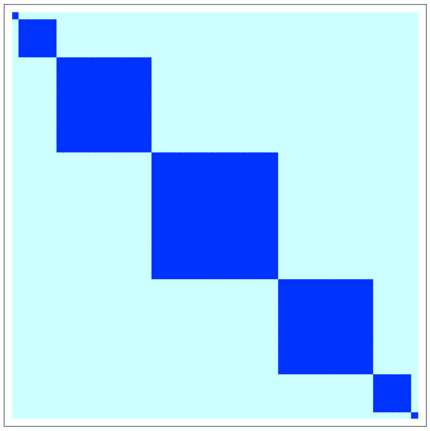
$$x \; \rho \; y \iff \exists \, i \, (x \in P_i \wedge y \in P_i).$$

It is easy to check that ρ is an equivalence relation. □

The two operations in the last proof transforming equivalence relations into partitions and back are mutually inverse.

Equivalence relations on a finite set can be recognized easily from their Boolean matrix representation–but only if the set is properly enumerated. For example, consider $\mathbf{2}^6$, the set of all binary lists of length 6. Define $x \; \rho \; y$ to mean that lists x and y have the same number of 1's. Clearly ρ is an equivalence relation. Here are two Boolean matrices representing this relation.





The matrix on the left is based on standard lexicographic ordering of binary lists. It shows reflexivity and symmetry, but transitivity is far from clear. However, we can reorder the carrier set to obtain the matrix on the right. The blocks of size 1, 6, 15, 20, 15, 6, 1 are clearly visible.

The next equivalence relation is drawn from geometry. We will refrain from giving detailed definitions and appeal to common sense instead.

Definition 2.2.7 *Two polygons P and Q are equidecomposable if P can be cut up into finitely many triangles which can be reassembled to form Q.*

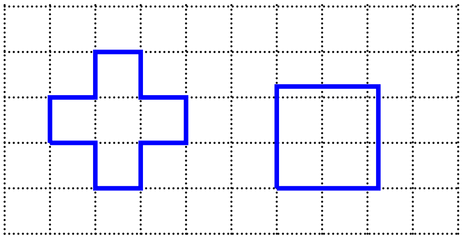
Lemma 2.2.4 *Equidecomposability is an equivalence relation.*

Proof. Reflexivity and symmetry are obvious. But transitivity is not: we need the fact that the intersection of two polygons is another polygon (or a set of polygons) and that all polygons can be triangulated. □

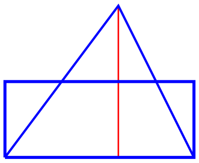
Clearly, any two equidecomposable polygons have the same area. There is a well-known theorem that asserts that the converse is also true.

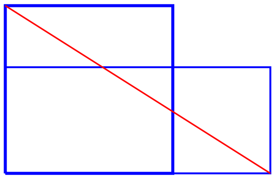
Theorem 2.2.1 (Bolyai-Gerwien Theorem) *Let P and Q be two polygons of equal area. Then P can be partitioned into finitely many triangles that can be reassembled to form Q.*

For example, in the figure below both polygons have an area of 5 units. It is an excellent exercise to try to find a concrete decomposition that will take the cross to the square. The fewer pieces the better.

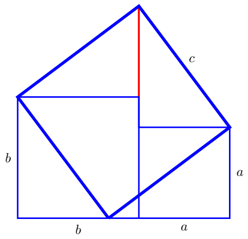


Proof. To prove the Bolyai-Gerwien theorem it suffices to show that every polygon is equidecomposable with a square: there is only one square for each area. We will show first how to translate triangles to rectangles, then rectangles to squares, and lastly how to add squares. Rather than giving detailed geometric proofs we will argue in terms of pictures and leave it to the reader to fill in the numerous details. The first two steps are indicated in the figures below.





These are not straightedge and compass constructions, we only need to know that a decomposition exists, we do not have to worry about constructing it. Also, these pictures are somewhat misleading, there is no explanation of how to deal with obtuse triangles, or in the second step what to when the longer side of the rectangles is less than 4 times the shorter one. Make sure you understand how to deal with all these other cases. In the last step we use the Pythagorean theorem $a^2 + b^2 = c^2$.



So suppose we have two polygons P and Q of equal area. We triangulate P, convert the triangles into rectangles, and the rectangles into squares by steps 1 and 2. The squares are then added up as in step 3. Thus, we obtain a square with the same area as P. The same argument applies to Q, and we are done. □

2.2.3 Exercises

Exercise 2.2.1 Suppose ρ and σ are binary relations on A. Which of the following claims are true:

- ρ and ρ symmetric implies $\rho \sqcup \sigma$ symmetric.
- ρ and ρ symmetric implies $\rho \sqcap \sigma$ symmetric.
- ρ and ρ transitive implies $\rho \sqcup \sigma$ transitive.
- ρ and ρ transitive implies $\rho \sqcap \sigma$ transitive.

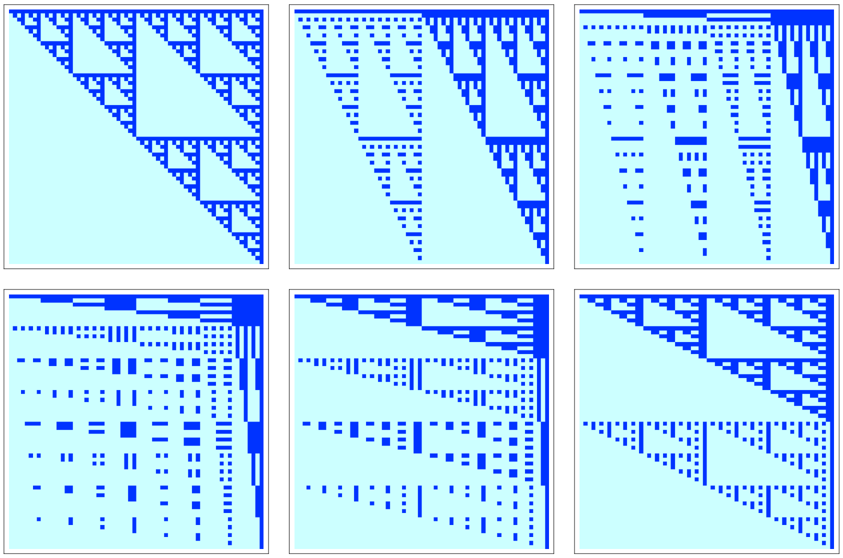
Exercise 2.2.2 Show that a asymmetric relation is antisymmetric. How about the opposite direction?

Exercise 2.2.3 Suppose a relation is given by a Boolean matrix. How can one check whether the relation is transitive? What is the running time of your algorithm?

Exercise 2.2.4 The converse of a pre/partial/total order is again pre/partial/total order.

Exercise 2.2.5 Show that greatest (maximal) elements in a poset are least (minimal) in the converse order.

Exercise 2.2.6 Consider the partial order on binary lists of length n defined by $K \leq L$ iff $K \leq \mathsf{rot}_i(L)$ where rot_i indicates cyclic rotation by i places. As always, the order is understood to be bitwise. Here is the Boolean matrix representation for $n = 6$ and $i = 0, \ldots, 5$.



Explain the picture sequence.

Exercise 2.2.7 Show that lexicographic order is a total order on the set of all words. Show that length-lex order is a total order on the set of all words.

Exercise 2.2.8 For which of the word orders just defined is it true that any word is larger than only finitely many words?

Exercise 2.2.9 Is the product order on $\mathbb{N} \times \mathbb{N}$ a well-order?

Exercise 2.2.10 Consider the chess board $C = [8] \times [8]$. Define a relation ρ on C by: $x \; \rho \; y$ if a knight can move from x to y (in a sequence of single moves). Show that ρ is an equivalence relation and determine the equivalence classes. Repeat for a rook, king, queen. How about pawns?

Exercise 2.2.11 What are the sizes of these blocks when the carrier set is all binary lists of length n? What is a reordering of the carrier set that produces the clear picture?

Exercise 2.2.12 Sorting algorithms often take as input a list-type container and a user define comparison operation $\mathrm{cmp}(x, y)$ that takes as input two objects of whatever type we are trying to sort, and returns a Boolean. Here is a description of the required properties of cmp in a book by Stroustrup.

(1) $\mathrm{cmp}(x, x)$ is false.
(2) If $\mathrm{cmp}(x, y)$ and $\mathrm{cmp}(y, z)$, then $\mathrm{cmp}(x, z)$.
(3) Define $\mathrm{equiv}(x, y)$ to be $!(\mathrm{cmp}(x, y)||\mathrm{cmp}(y, x))$. If $\mathrm{equiv}(x, y)$ and $\mathrm{equiv}(y, z)$, then $\mathrm{equiv}(x, z)$.

What does this mean in our framework?

2.3 Transitive Closure

2.3.1 Closure Operations

Suppose P is some property of relations on a set A and let $(\rho_i)_{i\in I}$ be a family of relations. We say that P is intersection-closed if

$$\text{for all } i \in I : P(\rho_i) \quad \text{implies} \quad P\left(\bigcap\nolimits_{i\in I} \rho_i\right)$$

For example, the property "is transitive" is intersection-closed. The same is true for "is an equivalence relation." Now suppose we have a relation that fails to have property P, and we would like to inflate it a little to ensure P holds. We can do this for intersection-closed properties by applying a closure operation:

$$\mathsf{clos}_P(\rho) = \bigcap \{ \sigma \mid \rho \sqsubseteq \sigma, P(\sigma) \}$$

To see why this works, first consider an empty family of relations. Since our property is intersection-closed, the universal relation U_A must have property P; hence there is at least one candidate σ on the right hand side. But then the big intersection indeed produces the least relation containing ρ with property P.

Definition 2.3.1 (Transitive Closure) *Let ρ be a relation on A. Define the transitive reflexive closure to be the finest relation, coarser than ρ, that is reflexive and transitive. The transitive closure is the finest relation, coarser than ρ, that is transitive.*

We write $\mathsf{TRC}(\rho)$ or $\rho^\star$ for the reflexive transitive closure and $\mathsf{TC}(\rho)$ or ρ^+ for the transitive closure.[2]. For instance, $\rho^+ = \rho^\star \diamond \rho$.

The property of being transitive is clearly intersection-closed, so the closure operation from above works. The construction is elegant, but has little algorithmic meaning, even when A is finite. A better characterization is given in the next lemma.

Lemma 2.3.1

$$\mathsf{TRC}(\rho) = \bigcup_{k\geq 0} \rho^k = I_A \cup \rho \cup \rho^2 \cup \rho^3 \cup \dots$$
$$\mathsf{TC}(\rho) = \bigcup_{k>0} \rho^k = \rho \cup \rho^2 \cup \rho^3 \cup \dots$$

[2] The $\rho^\star$ notation can be misleading since it is also used to denote an entirely different operation, the so-called Kleene star operation.

Proof. It suffices to establish the claim for the transitive closure. By induction on k we can show that $\rho^k \sqsubseteq \tau$ for all $k \geq 1$ whenever τ is a transitive relation coarser than ρ. But $\mathsf{TC}(\rho)$ is the finest such relation and we are done. □

Example 2.3.1 As an example in genealogy, the transitive closure of the ’parent-of’ relation is ’ancestor-of’ (also known as the ancestral relation).

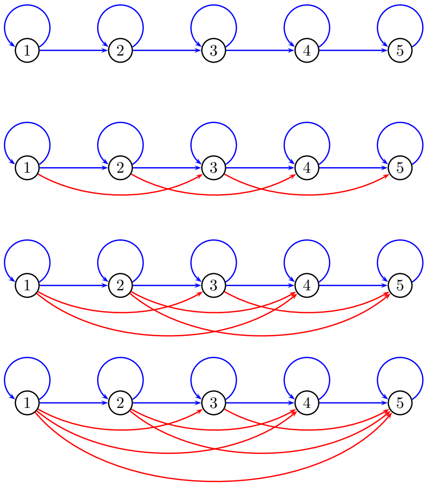
Example 2.3.2 Consider the successor relation on $\mathbb{N}$: $x \; \rho \; y \iff y = x + 1$. Then $x \; \rho^k \; y \iff y = x + k$, $x\rho^+ y \iff x < y$ and $x\rho^\star y \iff x \leq y$.

Example 2.3.3 Let A by all binary sequences of length n. Define

$$x \; \rho \; y \iff \exists \, i \, (x_i = 0 \wedge y_i = 1).$$

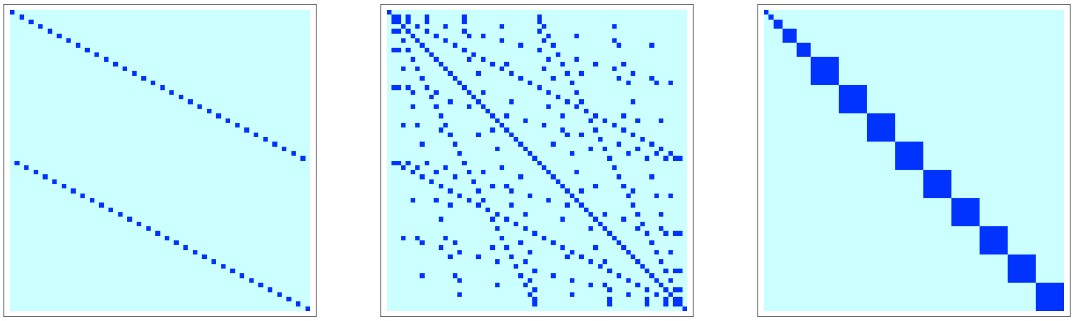
Then $\rho^\star$ corresponds to the subset relation if we think of the sequences as bitvectors.

Example 2.3.4 Let us return to the successor relation. This time we consider the reflexive version on $[5]$. We display only the powers up to ρ^3, nothing changes beyond this point. The red arrows indicate the new relationships produced by the closure process.



It is easy to show that if ρ is reflexive then $\rho^k \subseteq \rho^{k+1}$, which fact is responsible for the steady increase in the number of arrows in the last example. This suggests a way to improve the computation: instead of composing ρ^k with ρ, we might square the relation at each step: $\rho^{2k} = \rho^k \diamond \rho^k$.

Example 2.3.5 Let A be the collection of binary lists of length 6 and define a relation ρ as follows: $K \; \rho \; L$ if K is obtained from L by cyclic rotation by one place to the left. For instance, $110001 \; \rho \; 111000$. In the standard lexicographic enumeration of A the relation ρ and its transitive reflexive closure look like the first two matrices.



If we rearrange A properly, we get the third matrix on the right: each equivalence class is now represented by a square centered on the diagonal. It is a good exercise to determine the equivalence classes.

Here are a few basic properties of the transitive closure operation.

Lemma 2.3.2 *Let ρ and σ be two relations on A.*

1. *ρ reflexive implies* $\mathsf{TC}(\rho) = \mathsf{TRC}(\rho)$
2. $\mathsf{TC}(\rho) = \rho \diamond \mathsf{TRC}(\rho) = \mathsf{TRC}(\rho) \diamond \rho$
3. $\mathsf{TRC}(\rho) = \mathsf{RC}(\mathsf{TC}(\rho)) = \mathsf{TC}(\mathsf{RC}(\rho))$
4. $\mathsf{TRC}(\mathsf{TRC}(\rho)) = \mathsf{TRC}(\rho)$
5. $\mathsf{TRC}(\rho \sqcup \sigma) = \mathsf{TRC}(\mathsf{TRC}(\rho) \diamond \mathsf{TRC}(\sigma))$

Proof. Argue with chains, and a little induction. E.g., $x \; \mathsf{TRC}(\mathsf{TRC}(\rho)) \; y$ means there is a $\mathsf{TRC}(\rho)$-chain from x to y. But each step in the chain can be replaced by a ρ-chain. Joining up all the pieces we get a ρ-chain from x to y, hence $x \; \mathsf{TRC}(\rho) \; y$. □

The last two items in the lemma make it possible in certain circumstances to eliminate nested closure operations.

2.3.2 Transitive Reduction

A relation that has some special properties such as symmetry or transitivity can be represented more concisely by eliminating parts of the relation that can be deduced from other parts and these special properties. In the case of symmetry, it suffices have either $a \; \rho \; b$ or $b \; \rho \; a$. Transitivity is far more interesting.

Definition 2.3.2 *Let ρ be a transitive relation on a finite set A. τ is a transitive reduction of ρ if the transitive closure of τ is ρ. A minimal transitive reduction of ρ is a transitive reduction of minimal cardinality.*

In general, the minimal transitive reduction of a relation need not be uniquely determined. For example, on carrier set $[6]$, suppose all points in $\{1, 2, 3\}$ are related, as are all points in $\{4, 5, 6\}$; moreover, $1 \; \rho \; 4$ and $2 \; \rho \; 5$. There are two transitive reductions of minimal cardinality. However, if we exclude cycles, we have uniqueness.

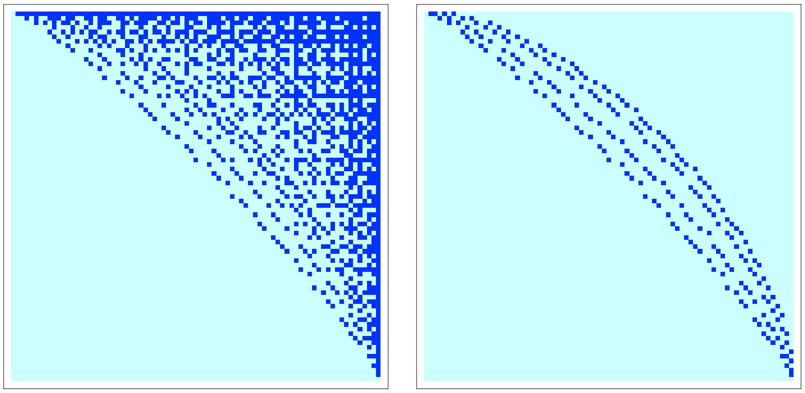
Lemma 2.3.3 *Let ρ be a partial order on a finite set A. Then ρ has a uniquely determined minimal transitive reduction.*

Proof. Let τ be a transitive reduction of minimal cardinality, and σ any transitive reduction. It suffices to show that $\tau \subseteq \sigma$. Suppose that τ contains a pair (a, b) not in σ. Then there is a σ-chain from a to b: $a = x_0 \; \sigma \; x_1 \; \sigma \; x_2 \; \sigma \ldots x_{k-1} \; \sigma \; x_k = b$. But since the closure of τ is ρ we must have $x_i \tau^* x_{i+1}$ for all i. None of these τ-chains can contain (a, b) since otherwise all elements would have to be equal. But then there is a τ-chain from a to b, contradicting the minimality of τ. □

This uniquely determined minimal transitive reduction of a poset is usually called the Hasse diagram of the poset. We can characterize the Hasse diagram as the intersection $\bigcap\{\, \sigma \mid \mathsf{TC}(\sigma) = \rho \,\}$. Another way to describe the Hasse diagram is to note that it consists of all pairs (x, y) where y covers x. The question arises how one can compute the minimal reduction efficiently. Here is a Surprising method that uses the Boolean matrix representation $A = \mathsf{M}_\rho$ of ρ.

```
$H = A$
**for** $i = 1, \ldots, n$ **do**
**for** $j = 1, \ldots, n$ **do**
**for** $k = 1, \ldots, n$ **do**
    **if** $A[i, j] \wedge A[j, k]$ **then** $H[i, k] = 0$
```

Example 2.3.6 Let A be the set of divisors of 44100, in standard order. The (proper) divisibility relation on A is shown on the left, and the corresponding Hasse diagram on the right.



The cardinality of the relation is reduced from 1215 to 216.

2.3.3 Exercises

Exercise 2.3.1 Show that the Hasse diagram of a finite poset consists of all pairs (x, y) where y covers x. Conclude that the "reverse" Warshall algorithm from above correctly computes the Hasse diagram of a finite poset.

Exercise 2.3.2 Transitive reductions make sense for some relations on infinite carriers. For example, on the natural numbers the standard order is the closure of the successor relation. On the other hand for the rationals there is no reduction. Find a description of orders on infinite sets that allow transitive reduction.

Exercise 2.3.3 Suppose we are given a sequence of integers, implemented as a linked list. Show how to use a the repeated squaring idea from the transitive closure algorithm to compute all prefix sums of the integer sequence. This is not particularly interesting for a single-processor machine, but should work in logarithmic time on a multi-processor machine.

Exercise 2.3.4 Make sure the transitive reflexive closure of the rotation relation is really an equivalence relation. Can you determine the number and size of the equivalence classes?

Exercise 2.3.5 Explain how to handle the case where n is odd in the divide-and-conquer algorithm for transitive closure.

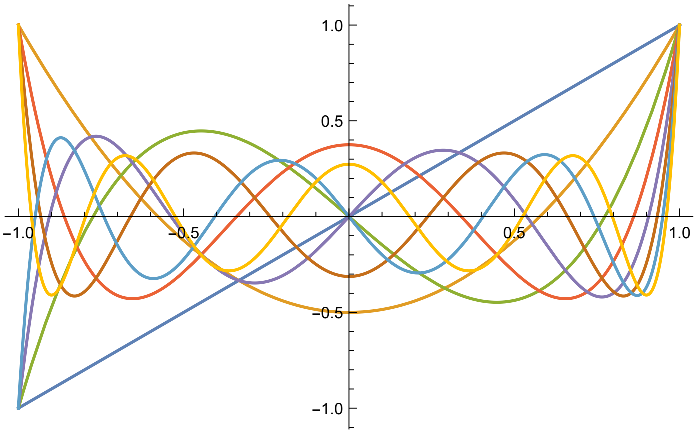
Exercise 2.3.6 What would the chessboard pictures look like for a king, a rook or a bishop? How about a pawn? What changes if we consider boards of general size $n \times m$?

Three

Functions

3.1 Functions Defined

Informally, a function is a rule that associates an input object with a uniquely determined output object. Functions are quite familiar from calculus; for example, here is a plot of the first few Legendre polynomials over the interval $[-1, 1]$.



To specify particular Legendre polynomial, say, P_8, one usually writes

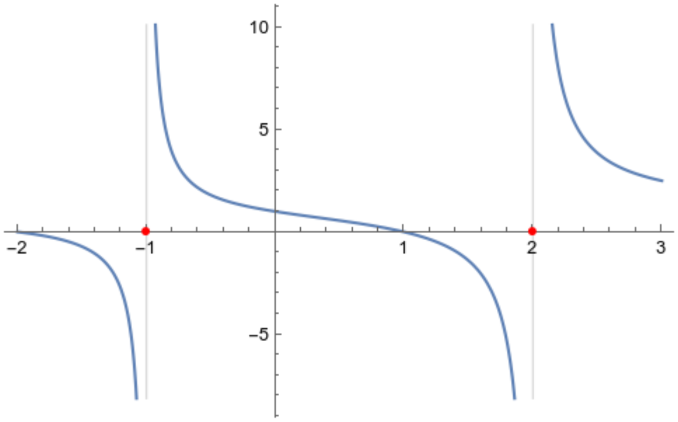
$$P_8(x) = \frac{1}{128}(35 - 1260x^2 + 6930x^4 - 12012x^6 + 6435x^8)$$

The expression on the right hand side determines how to compute the function value, given some particular argument x. Here it is tacitly assumed that the input object x is a real number, and we can calculate the value simply by substituting some specific real such $\sqrt{2}$ for x and then performing the necessary arithmetic. In this case, the result is 32099/128.

We may also encounter a definition along the lines of

$$f(x) = \frac{x^2 + x - 2}{x^2 - x - 2}$$

Presumably, the intended domain is still the reals, but that does not quite work: we must exclude $x = -1$ and $x = 2$ where the function has singularities. Finding these holes requires a bit of effort, but is easy to see from the plot:



Of course, that just means that the plotting algorithm has done all the work. At any rate, in areas such as algebra or computer science one usually deals with many different types of objects, it is better to include information about input and output as part of the definition. We already know how to do this, we model functions as a particular type of relation $f : A \to B$.

There is another, more subtle problem with these calculus-style definitions: the function is specified by an expression, essentially a simple straight-line program. Given a particular argument a, we can calculate the function value $f(a)$ by substituting a for x in the expression and evaluating. This amounts to an intensional definition of a function, we specify a particular computation that leads to the desired result. In computer science this perspective seems most natural: we often think of a function as an algorithm and expect there to be a program that implements it. However, it has become clear in the latter part of the 19th century that this perspective is a bit too narrow. One can benefit greatly from a more general definition that ignores all the internals of a function and simply pins down the most basic aspect: the external behavior of the function, the observable connection between input and output pairs. This is known as the extensional point of view: a function is an input/output relation, a collection of pairs, together with a specification of the domain and codomain. The collection of pairs is just a set, the graph of the function, and can be defined in many ways that do not involve the specification of a particular rule or algorithm. connecting input to output. This type of function exists in some set-theoretic universe, possibly far away from the realm of computation. This may seem like a bold step in the wrong direction, but it works very well in practice. In fact, there is just no way around it. In the chapter on computability, we will see that, even in arithmetic, non-computable functions are no less important than computable ones.

Definition 3.1.1 (Functions) *A relation $\rho : A \to B$ is a function (or map, mapping) if*

1. *ρ is single-valued or functional: $x \,\rho\, u \wedge x \,\rho\, v \Rightarrow u = v$.*
2. *ρ is total: $\forall\, x \in A \,\exists\, y \in B \,(x \,\rho\, y)$.*

A relation that is single-valued, but not necessarily total, is a partial function, in symbols $f : A \nrightarrow B$. The collection of all functions from A to B is written $A \to B$ or B^A. For partial functions we similarly write $A \nrightarrow B$; $f(x)\downarrow$ indicates that x is in the support of f and we say that f converges on x.

Observe that, according to this definition, the calculus examples from above are not fully specified, they lack explicit descriptions of domain and codomain—the reader is supposed to infer them from context. We insist that they be made explicit. This kind of elaborate description should be familiar from many programming languages, where we have to say clearly that some function maps, say, `string` to `int`. Having said that, as a practical matter, we may relax a bit if domain and codomain are entirely obvious from context; in that case it is fine to just specify the graph.

As a simple notational device, we will usually write f, g, h and so on for functions, and denote arbitrary relations by ρ, σ, τ and so on. Also, when f is a function we write $f(x) = y$, rather than $x \; f \; y$, let alone $(x, y) \in f$. This is justified by single-valuedness, since there is only one y such that $x \; f \; y$, we might as well write $f(x)$ for it. Functions are but a special kinds of relations, so all concepts associated with relations still apply to functions. For example, we can still talk about the range of f, a subset of the codomain. The definition also settles annoying edge cases. For example, $\emptyset : \emptyset \to B$ is a function, the empty function. On the other hand, for $A \neq \emptyset$, there is no function $A \to \emptyset$. However, there is a number of differences between relations in general and functions in particular, and it is worth pointing them out.

In a sense, we could eliminate partial functions by turning them into total ones: given $f : A \nrightarrow B$, pick a new element $\perp$ not in B and adjoin it to the codomain: let $B_\perp = B \cup \{\perp\}$ and define a new total function $f_\perp : A \to B_\perp$ by

$$f_\perp(x) = \begin{cases} f(x) & \text{if } f(x)\downarrow, \\ \perp & \text{otherwise.} \end{cases}$$

One might think that $f_\perp$ is "essentially" the same as f, but that turns out to be quite false. As we will see in the chapter on computability, this little maneuver wreaks havoc with computable functions: there are computable f for which $f_\perp$ fails to be computable. The problem is that, in general, we cannot determine on which inputs a computable function converges. This is the infamous Halting Problem and plays a central role in computability theory.

3.1.1 Converse and Inverse

The converse f^{op} of a function is often written f^{-1} to emphasize that f is a function. Note well, f^{-1} usually fails to be single-valued or total, it is just a relation, called the inverse of f. If we wanted to insist that f^{-1} should somehow be a function, we would need to adjust the codomain and allow for partial functions: $f^{-1} : B \nrightarrow \mathfrak{P}(A)$ where $f^{-1}(b) = \{\, a \in A \mid f(a) = b \,\}$. The sets $f^{-1}(b)$ are called the inverse image, pre-image or fiber of $b \in B$ under f. For example, for $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x^2$, we have $f^{-1}(b) = \{\pm\sqrt{b}\}$ or $\emptyset$, depending on whether $b \geq 0$. Thus, there are fibers of cardinality 0, 1 and 2. In one special case we can get rid of the powerset, though: if all the values of f^{-1} are singletons, we can also think of f^{-1} as a partial function $f^{-1} : B \nrightarrow A$. And, if the range of f is B, we actually wind up with a function $f^{-1} : B \to A$. Modifying the last example slightly, let $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x^3$. Then $f^{-1}(b) = \mathsf{sign}(b)\, |b|^{1/3}$ is a function. The correct interpretation will always be clear from context.

To summarize:

- If $f : A \to B$ is bijective then $f^{-1} : B \to A$ is a function and even a bijection.
- If $f : A \to B$ is injective then $f^{-1} : B \nrightarrow A$ is a partial function with support $\mathsf{rng}\, f$. This partial function can be extended to a function $f^{-1} : B \to A$ by picking arbitrary values for points in $B - \mathsf{rng}(f)$.

3.1.2 Sets of Arguments

For a function $f : A \to B$, it is often convenient to apply f not just to elements of A, but also to subsets of the domain. In functional programming languages, this is handled by a

map operation. To this end, let $X \subseteq A$ and $Y \subseteq B$ and define[1]

$$f(X) = \{\, f(x) \mid x \in X \,\} \subseteq B$$
$$f^{-1}(Y) = \{\, x \in A \mid f(x) \in Y \,\} \subseteq A$$

We can also explain application of f to sets of arguments as lifting the function $f : A \to B$ to the power set to obtain a new function $F : \mathfrak{P}(A) \to \mathfrak{P}(A)$. More interesting is that we can also obtain an inverse function

$$F^{-1} : \mathfrak{P}(B) \to \mathfrak{P}(A)$$

Note the switch in the order of domain and codomain (a contravariant operation as opposed to the covariant F). The point here is that F^{-1} is a bonified function, even if f is not injective. If $Y \cap \mathsf{rng}(f) = \emptyset$, we have $f^{-1}(Y) = \emptyset$, but there is no problem with that. And, of course, $F^{-1}(\{b\})$ may well contain more that one element.

3.1.3 Characteristic Functions

Fix some set A. We can describe subsets of A by means of so-called characteristic functions, maps $A \to \mathbf{2}$. To wit, given any subset $B \subseteq A$, we define a map $\mathsf{char}_B : A \to \mathbf{2}$ by

$$\mathsf{char}_B(x) = \begin{cases} 1 & \text{if } x \in B, \\ 0 & \text{otherwise.} \end{cases}$$

If A is finite, we can determine the number of elements in B by summing $\sum_{i \in [n]} \mathsf{char}_B(i)$.

There is a nifty device introduced by K. Iverson in the programming language APL, the Iverson bracket, that makes it a bit easier to use characteristic functions. Given any predicate $P(x, y, \ldots)$, Iverson defines $[P(x, y, \ldots)]$ to be 1 if $P(x, y, \ldots)$ holds, and 0 otherwise[2]. Here $P(x, y, \ldots)$ is any predicate with variables x, y, ... Iverson brackets establish a connection between logic and arithmetic:

Claim: $[P(x) \land Q(x)] = [P(x)][Q(x)]$ and $[\neg P(x)] = 1 - [P(x)]$.

It follows that $[P(x) \lor Q(x)] = [P(x)] + [Q(x)] - [P(x)][Q(x)]$. See the exercises for similar properties of Iverson brackets. Here are some applications.

For $A = \mathbb{R}$, $[x \in \mathbb{Q}]$ gives a 1-bit answer to the question: "Is x a rational number?" This function is due to Dirichlet and you probably have encountered it in calculus: the definition of the function is relatively simple, yet it is nowhere continuous; it is often given as an example for a function that is Lebesgue integrable but not Riemann integrable.

Any characteristic function can then be written as $\mathsf{char}_B(x) = [x \in B]$, so we can express the number of primes up to n simply as $\sum_{x \leq n} [x \text{ prime}]$.

In the special case where $A = [n]$, and n is not too large, a characteristic function is just a bitvector and can be implemented efficiently as an array of bits (or, more realistically, and array of bytes or unsigned integers). Note that on a 64-bit architecture an array of just 16 unsigned integers suffices to represent subsets of a universe of cardinality 1024. Operations on these sets can be implemented by performing point-wise logical operations on the integers in the array, so bit-parallelism can be used to speed up these operations.

[1] Some authors write $f[X]$ or $f^{-1}[Y]$ for emphasis, we prefer to use square brackets for other purposes.

[2] You may have encountered a weaker version of this in algebra, the venerable Kronecker delta, which is defined as $\delta_{xy} = 1$ if $x = y$, 0 otherwise.

3.1.4 Sequences and Words

Functions can be used to define sequences or lists of objects. We have already seen how to handle finite sequences nicely as a recursive datatype, but, at least for the infinite variety, the function approach is better. Recall that we write $[n]$ for the set $\{1, 2, \ldots, n\}$.

Definition 3.1.2 *A finite sequence over a groundset A is a function $s : [n] \to A$ where $n \in \mathbb{N}$ is the length of the sequence. An infinite sequence over a groundset A is a function $s : \mathbb{N} \to A$. The set of all finite [infinite] sequences over A is written $A^\star$ [A^ω].*

One usually writes sequences with subscripts rather than function application:

$$s_1, s_2, \ldots, s_{n-1}, s_n$$

The length of a sequence s is usually indicated by $|s|$. From the definition, it follows that there is a unique sequence of length 0. We will denote this empty sequence by nil, so $|\mathsf{nil}| = 0$. Note that nil is none other than the function $\emptyset \to A$.

We can define a natural operation of concatenation on $A^\star$ as follows. Let $s, t \in A^\star$ and set $k = |s|$.

$$(s \cdot t)(i) = \begin{cases} s(i) & \text{if } i \leq k, \\ t(i-k) & \text{otherwise.} \end{cases}$$

Informally, this simply means that

$$s \cdot t = s_1 s_2 \ldots s_k \, t_1 t_2 \ldots t_{k'}$$

where $k' = |t|$, we append one sequence at the end of another.

An important special case arises when the ground set A is finite and consists of letters or letter-like symbols. In this case we refer to A as an alphabet, and to finite sequences over A as words. The empty word is written ε in this setting and one expresses concatenation by plain juxtaposition: uv instead of $u \cdot v$. So the word 'alphabet' is the concatenation of 'alpha' and 'bet'. Typical examples for alphabets are $\{a, b, c, \ldots, y, z\}$, $\{0, 1\}$ or the collection of all ASCII characters. Clearly, $|uv| = |u| + |v|$. Take a look at the chapter on finite state machines for much more information on words and how to compute with them.

One unpleasant topic in conjunction with sequences is the choice of a starting point for the enumeration. Arguably the most intuitive starting point is 1, signifying the position of the first element. Position 2 is where the second element goes, and so forth. A sequence of length n duly looks like $a_1, a_2, \ldots, a_n$. We refer to these sequences as 1-indexed. Unfortunately, there are several occasions when it is more convenient to start at 0, which produces a 0-indexed sequence. Many programming languages use 0-indexing for arrays, simply because this convention fits better with the memory model. Infinite sequences are most naturally modeled as maps $\mathbb{N} \to A$ and thus 0-indexed. We will use both approaches, whichever works better for the application at hand. Lastly, a useful notational trick is to exploit negative positions to indicate elements in a finite sequence starting from the end. Thus a_{-1} denotes the last element, a_{-2} the second to the last, and so on.

3.1.5 Exercises

Exercise 3.1.1 Suppose A and B are both finite. What is the cardinality of the set of functions from A to B?

Exercise 3.1.2 Given a function f, when is f^{-1} again a function? When is f^{-1} a partial function?

Exercise 3.1.3 Show that a relation f is a function from A to B if, and only if, $I_A \subseteq f \diamond f^c$ and $f^c \diamond f \subseteq I_B$.

Exercise 3.1.4 Show that concatenation on $A^\star$ is associative.

Exercise 3.1.5 What does the functional digraph of a constant function look like? Of the identity, of a bijection?

Exercise 3.1.6 Prove lemma 3.3.2 which characterizes surjectivity.

Exercise 3.1.7 Find a set $\mathbf{1}$ such that $A \to \mathbf{1}$ has cardinality 1 for all sets A. Discuss the maps $A \to \mathbf{1}$ and $\mathbf{1} \to A$.

Exercise 3.1.8 Let $f : A \to B$ and $X \subseteq A$, $Y \subseteq B$. Show that $f(f^{-1}(Y)) \subseteq Y$, with equality for f surjective. Show that $X \subseteq f^{-1}(f(X))$, with equality for f injective.

Exercise 3.1.9 Let $f : A \to B$ and $X_i \subseteq A$. Show that $f(X_1 \cup X_1) = f(X_0) \cup f(X_1)$. Show that $f(X_1 \cap X_1) \subseteq f(X_0) \cap f(X_1)$ with equality for f injective. Show that $f(X_1 - X_1) \subseteq f(X_0) - f(X_1)$ with equality for f injective.

Exercise 3.1.10 Let $f : A \to B$ and $Y_i \subseteq B$. Show that $f^{-1}(Y_1 \cup Y_1) = f^{-1}(Y_0) \cup f^{-1}(Y_1)$. Show that $f^{-1}(Y_1 \cap Y_1) = f^{-1}(Y_0) \cap f^{-1}(Y_1)$. Show that $f^{-1}(Y_1 - Y_1) = f^{-1}(Y_0) - f^{-1}(Y_1)$.

3.2 Operations of Functions

We consider functions to be special kinds of relations, so all the operations on relations automatically carry over the functions. However, some of these operations destroy functionality and are not particularly useful. For instance, the join of two functions $f \sqcup g$ is presumably of little interest. We take a closer look at operations that do make sense for functions.

3.2.1 Application and Composition ☠

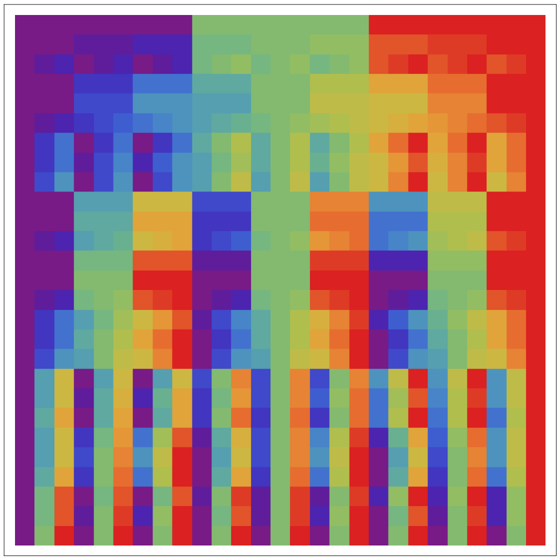
We now come to a slightly thorny issue, we would like to define the composition of functions analogous to the composition of relations from section 2.1.2. Recall that we write function application on the left, $f(x) = y$, though this convention is not without detractors[1] The problem is that there is a major clash between composition of functions as in $(g \circ f)(x) = g(f(x))$ and relations as in $(f \diamond g)(x) = g(f(x))$:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[rr, bend right, "g \circ f"'] & B \arrow[r, "g"] & C
\end{tikzcd}$$

[1] In 1976, the category theorist Charles Wells wrote "Observe that I write functions on the right and functional composition from left to right. This is undoubtedly the *Wave of the Future*. It makes functional diagrams easier to read and corresponds to the natural order of doing things on a pocket calculator." Alas, we are still waiting for Wells's wave to crest, or even to form.

Function composition also clashes with the standard convention for sequential composition of programs: `P;Q` is always interpreted as "execute `P` first, then `Q`." Alas, the convention to write functional composition in the opposite way of relational composition is firmly entrenched, there is no way to dislodge it now. We are trying to sidestep the issue as much as possible by using a different symbol for relational composition. Be warned, though, that most authors use the same symbol $\circ$ for both purposes. And some even mess up relational composition.

The multiplication table for composition of all functions on [3].



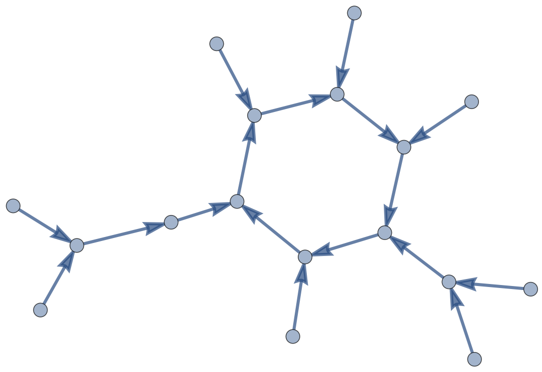
It is tempting to try to analyze the patterns in this picture, but we will hold off till the discussion of semigroups in the chapter on automata theory.

3.2.2 Iteration

Transient and Period

Recall that every binary relation on A can be construed as a directed graph with vertex set A. What are the special properties of functional digraphs, i.e., the directed graphs corresponding to functions? For the time being, assume that A is finite and consider some function $f : A \to A$. Write $G_f = \langle V, E \rangle$ for the functional digraph of f, so that the edges are of the form $x \to f(x)$. Since f is total and single-valued, every node in G_f has outdegree exactly 1. However, the indegree of a node may vary from 0 to $|A|$.

Here is an example, the functional digraph of the map $x \mapsto x^2 + 1 \bmod 10$, defined on the modular numbers mod 10. The nodes are unlabeled, how many of them can you identify?



What happens if we trace a path in a functional digraph starting from some vertex x? We obtain an infinite sequence of points

$$a, f(a), f(f(a)), \dots, f^k(a), \dots$$

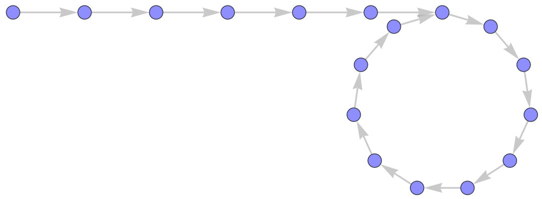
called the orbit or trajectory of f on $a \in A$. We can think of a pebble being placed at position a at time 0, and moving from position x to $f(x)$ at each step. In other words, we traverse the unique edge with source x. In the important case when A is finite, this path must ultimately wrap around and produce a cycle, the so-called limit cycle:

$$f^t(a) = f^{t+p}(a)$$

for some $t \geq 0$, $p > 0$. For fixed f, both these parameters depend on x.

Definition 3.2.1 *The least t and p such that $f^t(x) = f^{t+p}(x)$ is the transient and the period of x (with respect to f).*

Note that for infinite carrier sets, the transient of a point may well become infinite, and there may not be a periodic part; the successor function on $\mathbb{N}$ is a perfect example. At any rate, here is the typical lasso picture of an orbit on a finite carrier set, transient is 5 and period is 8:



3.2.3 Exercises

3.3 Types of Functions

We have seen that there are special types of relations such as equivalence relations or orders that are particularly important. Here is a first step towards a producing a similarly useful classification for functions.

Definition 3.3.1 (Sur/In/Bi-Jections) *Let $f : A \to B$ be a function.*

1. f *is surjective (or a surjection) if* $\forall\, y \in B\, \exists\, x \in A\, (f(x) = y)$,
2. f *is injective (or an injection) if* $f(x) = f(x') \Rightarrow x = x'$,
3. f *is bijective (or a bijection) if* f *is injective and surjective.*

Surjective functions are also called *onto* and injective functions are called *one-one*, terminology we will avoid.

The key idea behind injectivity is that one can recover the input x from the output $y = f(x)$; there is no loss of information when f is applied to an argument. Hence, the real function $x \mapsto x^3$ is injective, but $x \mapsto x^2$ is not. One also speaks of a reversible operation. We can tell from the graph of a function whether it is injective or not. This is not the case for surjectivity, a property that depends on the choice of codomain: the successor function from $\mathbb{N}$ to $\mathbb{N}$ fails to be surjective, but it is surjective as a map $\mathbb{N} \to \mathbb{N}_+$.

Observe that there is a basic asymmetry in the definition of a function: we require the support to be the whole domain, but the range may well be a proper subset of the codomain. Perhaps surprisingly, this convention turns out to be quite convenient. First, injectivity can be characterized in several useful ways.

Lemma 3.3.1 *Let $f : A \to B$ be a function. The following are equivalent:*

1. f *is injective.*
2. $f \diamond f^{\mathrm{op}} = I_A$.
3. f *has a left-inverse:* $\exists\, F : B \to A\ (F \circ f = I_A)$.
4. f *has the left-cancellation property:* $\forall\, g, h : C \to A\ (f \circ g = f \circ h \Rightarrow g = h)$.

Proof.

($1 \Rightarrow 2$) We have $a\ (f \diamond f^{\mathrm{op}})\ b$ iff $f(a) = c = f(b)$. Since f is injective, iff $a = b$.

($2 \Rightarrow 3$) By (2), f^{op} is a partial function $B \nrightarrow A$ with support $\mathsf{rng}\, f$. Pick some $a_0 \in A$ and set $F(b) = f^{op}(b)$ if b is in the range of f; otherwise set $F(b) = a_0$.

($3 \Rightarrow 4$) Assume (3) and let F, g and h as given. Then $g \circ f = h \circ f$ implies $F \circ (g \circ f) = F \circ (h \circ f)$. By associativity, $g = I_A \circ g = F \circ (f \circ g) = F \circ (f \circ h) = I_A \circ h = h$.

($4 \Rightarrow 1$) Assume (4) and let $a, b \in A$ such that $f(a) = f(b)$. Set $C = \{\bullet\}$ and let $g(\bullet) = a$, $h(\bullet) = b$. Then $f \circ g = f \circ h$, hence $g = h$ and thus $a = b$. □

As already mentioned, injectivity is clearly an intrinsic property of a function, while surjectivity seems a bit haphazard: it depends on the choice of codomain. Yet, there is a characterization of surjectivity that is "dual" to the characterization of injectivity.

Lemma 3.3.2 *The following are equivalent:*

1. f *is surjective.*
2. $I_B \subseteq f^{\mathrm{op}} \diamond f$.
3. f *has a right-inverse:* $\exists\, F : B \to A\ (f \circ F = I_B)$.
4. f *has the right-cancellation property:* $\forall\, g, h : B \to C\ (g \circ f = h \circ f \Rightarrow g = h)$.

3.3.1 Kernel Relations

Here is another application of functions to a concept from basic set theory: we can represent equivalence relations as functions. This is suggested by the observation that many equivalence relations are obtained from functions in the first place: we may consider triangles of the same area, sets of the same cardinality, binary lists with the same number of 1's, integers with the same remainder modulo m, and so on. In fact, upon closer inspection, it seems difficult to come up with an equivalence relation that does not arise from some associated function.

Definition 3.3.2 *Let $f : A \to B$ be a function. Define the kernel relation of f by*

$$x \,\rho\, y \iff f(x) = f(y)$$

ρ is usually written K_f.

It is very easy to check that K_f is an equivalence relation on A.

Example 3.3.1 Let A be all polygons, $B = \mathbb{R}$ and $f(x)$ the area of x. Then K_f is the "same-area" relation on polygons.

Let $f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = x \bmod m$. Then K_f is congruence modulo m.

How would we show that any equivalence relation is already a kernel relation? Suppose $\rho \subseteq A \times A$ is an equivalence relation. As a first attempt, we could set

$$f : A \to \mathfrak{P}(A) \qquad f(x) = [x]_\rho$$

This clearly works, but the codomain is uncomfortably large. For computational applications this is a bad solution. A better way is to keep the codomain down to A, and instead of mapping x to the whole equivalence class $[x]_\rho$ we choose one particular element in each class, say $x_0 \in [x]_\rho$ and map all of $[x]_\rho$ to that element.

$$f : A \to A \qquad f(x) = x_0$$

Better, but which element should one choose and how exactly can we do this in set theory? More formally, we need a choice function $f : A \to A$ for ρ, a function with the property that $\rho = K_f$ and $f(x) \,\rho\, x$. In the absence of any further information, we need the axiom of choice to produce the a choice function.

Theorem 3.3.1 *(AC)*
Every equivalence relation is a kernel relation.

Proof. Let ρ be an equivalence relation on A and consider the corresponding partition $(P_i)_{i \in I}$ whose blocks are the equivalence classes of ρ. By (AC), there is a choice set C such that $P_i \cap C = \{a_i\}$ for all i. We can then define a relation f on A by

$$x \; f \; y \quad \text{iff} \quad x \,\rho\, y \wedge y \in C$$

We have to check that f is really a function (not just a relation) and that $\rho = K_f$. First off, the support of f is A since $A = \bigcup_{i \in I} P_i$ and C intersects all P_i. But f is also single-valued: $x \,\rho\, y \in C \wedge x \,\rho\, z \in C$ implies $y \,\rho\, z$. Now C intersects each block in just one element, so $y = z$. Lastly, $\rho = K_f$ is clear:

$$x \,\rho\, y \iff \exists\, i \,(x, y \in P_i) \iff f(x) = f(y)$$

and we are done. □

Our first application of kernel relations is the following decomposition theorem that shows that all functions can be broken down into a surjection followed by an injection in an interesting way.

Theorem 3.3.2 *Every function is a composition of a surjective function and an injective function.*

Proof. Suppose the function is $f : A \to B$ and consider the kernel relation K_f. Define two new functions $g : A \to A/K_f$ and $h : A/K_f \to B$ by

$$g(x) = [x]_\rho \qquad h([x]_\rho) = f(x)$$

One can check that h is well-defined and that $f = h \circ g$. Moreover, h is injective, and g is surjective. □

One might wonder whether the application of the Axiom of Choice in this very general setting is of any relevance in computer science. Needless to say, for the sets one encounters in computer science, the axiom is usually not needed (never needed?), most of the time there is a natural order that makes it possible to pick “the least” element in $[x]_\rho$. In other words, there is a canonical choice function given by $f(x) = \min[x]_\rho$. This idea leads to a very nice data structure for equivalence relations, in particular in the standard case where the carrier set is (n) or $[n]$: we can represent the function as a plain array of integers. Note that we can test equivalence in constant time in this setting.

Example 3.3.2 Congruence modulo 4 on [10] produces the canonical choice function

x	1	2	3	4	5	6	7	8	9	10
$f(x)$	1	2	3	4	1	2	3	4	1	2

It is easy to check whether a function on $[n]$ is a canonical choice function: it needs to be idempotent $f \circ f = f$ and deflationary $f(x) \leq x$.

3.3.2 Permutations

When the function in question is a bijection and the carrier set is finite, all the transients are 0 and the functional digraph is a collection of disjoint cycles. These functions play a major role in combinatorics, algebra and set theory.

Definition 3.3.3 *A permutation is a bijection $f : A \to A$, in particular when A is a finite set.*

With a view towards implementation, we will focus on $A = [n]$. First, the inevitable counting argument.

Lemma 3.3.3 *There are $n!$ permutations on $[n]$.*

Proof. Here is the cookie cutter proof by induction on n. The base case $n = 1$ is obvious. For the step from $n-1$ to n, note that we can insert the new element n in n different places into any permutation of $n-1$, indicated below by vertical bars:

$$|a_1|a_2|a_3|\ldots|a_{n-2}|a_{n-1}|.$$

By IH, this produces $(n-1)! \cdot n = n!$ permutations. □

The lemma yields an algorithm to generate all permutations, albeit a fairly clumsy one. We will later see better methods.

We can represent a permutation f by a $2 \times n$ matrix of the form

$$\begin{pmatrix} 1 & 2 & 3 & \dots & n-1 & n \\ f(1) & f(2) & f(3) & \dots & f(n-1) & f(n) \end{pmatrix}$$

This is the so-called two-line representation of f. Needless to say, the first row in this matrix is really redundant, but this redundancy makes it a bit easier to read off specific values. Alternatively, we can use the obvious one-line representation:

$$\big(f(1), f(2), \dots, f(n-1), f(n)\big)$$

Since lists are hopelessly overloaded this can be a bit ambiguous, we will make sure to point out when we use this notation. As already mentioned, the functional digraph of a permutation f is particularly simple: it consists only of cycles, there are no transients anywhere. Thus, at least in the finite case, a complete description (up to isomorphism) can be given by counting all cycles of all lengths.

The usual reachability relation in the digraph for f can be characterized as

$$x \mathrel{\rho} y \iff \exists\, i \geq 0 \,(f^i(x) = y).$$

It is clear that ρ is reflexive and transitive for any map f. If f is in addition a permutation, then ρ is an equivalence relation: since every point x in A belongs to exactly one cycle of f we naturally obtain a partition of A. More precisely, to see that ρ is symmetric, assume $f^i(x) = y$. Thus, x and y lie on the same cycle; let k be the length of this cycle. Then $f^{k-i}(y) = x$.

The last observation produces a method to compute the inverse of a permutation, albeit a rather inefficient one. Suppose the cycles of permutation f have lengths $\ell_1, \dots, \ell_k$.

Lemma 3.3.4 *Let m be the least common multiple of $\ell_1, \dots, \ell_k$. Then $f^m = I$.*

This has the consequence that $f^{-1} = f^{m-1}$. Hence we can compute the inverse by computing an iterate of f. The number m from the lemma is none other than the order of f in the permutation group on $[n]$.

3.3.3 Exercises

Exercise 3.3.1 Find a useful way of computing the inverse of a permutation. Find another way to compute the inverse of a permutation based on sorting.

3.4 ∗ Functions and Diagrams

One of the many interesting discoveries of 20th century mathematics was that it can be very productive to present objects together with functions between them, to consider both as first-class citizens. In algebra, for example, it makes sense to discuss groups together with homomorphisms, maps between groups that preserve algebraic structure. This insight lead to the development of category theory, another foundational framework. We are not going to get involved with category theory, but we want to show in one small example how the use of functions can help explain the structure of objects, particular when diagrams are used to visualize the connections between various domains and codomains.

As a case in point, let us return to our definition of the Cartesian product in set theory. The appropriate types of functions are simply all set-theoretic functions, so there is no need for extra caution. In our original definition of the Cartesian product, we used Kuratowski's pairing function to form ordered pairs of objects. Collecting all these pairs into a set $A \times B$ is then simply a matter of set formation, and it turns out the resulting set has lots of useful properties. For example, we have used Cartesian products to define relations and functions. Alas, it is not unreasonable to object that this whole approach is a bit contrived and ad hoc. It would have been better to focus on the basic concept of a *product of sets* and then shown that our construction can serve to implement the concept in terms of sets. After all, this is the very criticism we raised against the digit-based approach to defining arithmetical operations.

So what exactly are the critical properties of a product of sets? First, we should be able to project from some putative product $C = A * B$ down to A and B, we want there to be *projection maps* $p_1 : C \to A$ and $p_2 : C \to B$. We could try to express the properties of these functions more carefully, but this turns out to be unnecessary, our description of a product below will also take care of the projections.

Now suppose we have a map $h : X \to C$ where X is some arbitrary set. We can get maps from X to A and B by composing h with the projections:

$$f = p_1 \circ h \qquad g = p_2 \circ h.$$

Here is the critical and far from obvious insight: for a reasonable product, this also must work in the opposite direction. More precisely, given the two maps f and g there is a unique map h so that these identities hold, we can combine the two maps into a single one with codomain the Cartesian product. This is best stated in the form of a little diagram.

$$\begin{tikzcd}
 & X \arrow[dl, "f"'] \arrow[d, "\exists h"] \arrow[dr, "g"] & \\
A & C \arrow[l, "p_1"] \arrow[r, "p_2"'] & B
\end{tikzcd}$$

The diagram simply expresses the identities $f = p_1 \circ h$ and $g = p_2 \circ h$. This type of diagram is called commutative, since it does not matter which path we choose from one point to another, the result is always the same. In our case, we have only two identities, so the diagram may not seem particularly helpful, but when one deals with many maps and identities visualization really helps. Such diagrams are often much easier to read than the corresponding equations.

At any rate, we now can enshrine the essential features of a set product in a definition. We want a set C depending on A and B such that the following holds.

Universal Property of Products

For any set X and maps $f : X \to A$ and $g : X \to B$, there is a unique map $h : X \to C$ such that $p_1 \circ h = f$ and $p_2 \circ h = g$.

The map h here is required to be unique, so we can write it as a product $f * g$. It is easy to check that our old Cartesian product has this property, $C = A \times B$ works, but there are other ways to obtain the universal property.

Let us try to determine what the universal property tells us about the product of two simple sets, say, $A = \{a_1, a_2\}$ and $B = \{b_1, b_2, b_3\}$. Define $X = \{\bullet\}$, a one-element set, $f_i(\bullet) = a_i$, $g_i(\bullet) = b_i$. The maps $f_i \times g_j$ must all be distinct, so C must contain distinct elements $(f_i \times g_j)(\bullet)$. These represent the corresponding pairs (a_i, b_j). Moreover, there cannot be any other elements. For assume $\Box$ is another element in C. The projections must be defined on $\Box$, say, $p_1(\Box) = a_1$ and $p_2(\Box) = b_2$. But then the map $f_1 \times g_2$ is no longer uniquely defined, contradicting the universal property. Thus we can conclude that C is a set of 6 "pairs," without any reference to the implementation details.

This approach may seem a bit overly abstract, but it actually works quite well. First, pin down the essential properties, then worry about an actual realization. The same principle applies to writing programs, or at least is should.

Now we can tackle another issue with our definition of products in terms of Cartesian products. In analogy to sum and product operators in algebra, we would like to have a definition of $\prod_{i=1}^{n} A_i$ for all $n \geq 0$. Whatever the definition says, we want the universal property to hold: we can bundle up n functions $f_i : X \to A_i$ into single function $X \to \bigtimes_{i=1}^{n} A_i$. The special case $n = 0$ requires a bit of attention. Taking inspiration from algebra, we would like $\prod_{i=1}^{0} A_i$ to be a "neutral element" with respect to products, just the way $1 \in \mathbb{R}$ is a neutral element with respect to products of reals. Thus, we need some set $\mathbf{1}$ such that $\mathbf{1} \times A$ is the "same" as A. The reason for the quotation marks is we actually don't need the two sets to be identical, it is good enough if they are isomorphic, that we can easily identify them. For sets, that simply means that there is a bijection between them (and, more pragmatically, the bijection should be quite simple). So what should this object $\mathbf{1}$ be? A little bit of thought reveals that the critical property of $\mathbf{1}$ is that there is exactly one map $X \to \mathbf{1}$ for any set X. It category theory, this is called a final object. Such sets exist, in fact, any one-element set will work. In the past, we have written $\{\bullet\}$ for such a set.

We can implement a useful product operation using Cartesian products as follows.

$$\bigtimes_{i=1}^{0} A_i = \mathbf{1}$$

$$\bigtimes_{i=1}^{1} A_i = A_1$$

$$\bigtimes_{i=1}^{n+1} A_i = \big(\bigtimes_{i=1}^{n} A_i\big) \times A_{n+1}$$

This works fine, but note that there are many other ways of implementing products. We could have chosen a different kind of pairing operation, or we could have split off the first set rather than the last. Since $A \times (B \times C) \neq (A \times B) \times C$ in general we get another product. But none of these choices matter, there is a simple bijection $A \times (B \times C) \longleftrightarrow (A \times B) \times C$ and, most importantly, the universal property holds.

Incidentally, we can avoid an inductive definition entirely by using sets of sequences:

$$\prod_{i=1}^{n} A_i = \{\, f : [n] \to \bigcup A_i \mid f(i) \in A_i \,\}$$

In other words, we use sequences of length n whose ith element belongs to A_i. This is really quite natural, for example, vector spaces can be constructed this way. One reason this approach is of interest is that it naturally generalizes to infinite products:

$$\prod_{i \in I} A_i = \{\, f : I \to \bigcup A_i \mid f(i) \in A_i \,\}$$

where I is an arbitrary index set, often a cardinal. One can verify that the universal property still holds for this type of product. Infinite index sets cause a foundational problem, we need AC to show that $\prod A_i$ is not empty whenever all the components A_i are non-empty. Again, what really matters is the universal property.

Where there are products, there are sums. In our case, we often would like to form a union $A \cup B$, but in such a way that there is no overlap between the two constituent sets. If the two sets are disjoint, there is no issue, we can identify which set a generic element $x \in A \cup B$ belongs to. If they fail to be disjoint, we need to modify the sets a bit before taking the union: for example, let $A' = \{\emptyset\} \times A$ and $B' = \{\{\emptyset\}\} \times B$, using Cartesian products. Then $A' \cup B'$ works as desired and is often written $A + B$. Note that there is no problem when $A = B$, we make a green copy of A and a red one, and put those together.

As before in the case of products, the implementation details are less important than the key properties. We want a set C, depending on A and B and injections $\iota_1 : A \to C$ and $\iota_2 : B \to C$.

Universal Property of Sums

For any set X and maps $f : A \to X$ and $g : B \to X$, there is a unique map $h : C \to X$ such that $h \circ \iota_1 = f$ and $h \circ \iota_2 = g$.

The new map is usually written $f + g$. Again we can express these identities in terms of a little diagram:

$$\begin{tikzcd}
 & X & \\
A \arrow[ur, "f"] \arrow[r, "\iota_1"'] & C \arrow[u, "\exists h"'] & B \arrow[ul, "g"'] \arrow[l, "\iota_2"]
\end{tikzcd}$$

Note the similarity to the diagram for products, we only need to flip all the arrows. As one might suspect, we encounter similar difficulties as with products. For example, according to our implementation, we have $A + B \neq B + A$, though the two sets are clearly isomorphic in some natural sense. Also, what should $\mathbf{0} = \sum_{i=1}^{0} A_i$ be? One can check that the empty set works (the initial object in the category of sets, there is exactly one map $\emptyset \to X$ for any set X), but will not get involved with the details.

3.4.1 Exercises

Exercise 3.4.1 Show that the Cartesian product is in fact a product in the sense of the universal property. Why does $\mathbf{1}$ behave as intended?

Exercise 3.4.2 Show that the sequence based product is in fact a product in the sense of the universal property.

Four

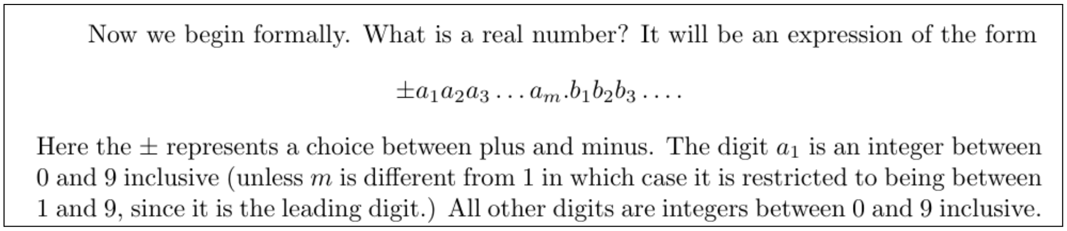
The Reals

4.1 Reals and StringWorld

We are now going to show how the reals can be formally defined in set theory. It is convenient to break up the construction into three steps: first, we build the integers from the naturals, second, the rationals from the integers and, lastly, the reals from the rationals. The first two steps are substantially less complicated, we are still very close to hereditarily finite sets. For the reals, infinite sets will be at the core of the construction.

Why would we bother with all this precision? Would it not be easier and more productive to rely on our fairly accurate intuition, much of it coming from geometry, and think of the reals as the *numberline*? After all, this approach worked amazingly well for Isaac Newton or Leonhard Euler. Unfortunately, when it comes to the reals our intuition is no longer fully reliable. For example, suppose we have two polygons A and B that have the same area. As we say in the Bolyai-Gerwien theorem, we can chop up A into finitely many triangles that can be rearranged to form B. It seems reasonable that a similar result holds for 3-dimensional polyhedra. This question appeared on Hilbert's famous list of open problems in 1900, in fact, it was the first problem on the list to be solved, albeit in the negative. One would think that any continuous function on the reals should be differentiable almost everywhere. Again, false. One last example: one can decompose a sphere of radius 1 into finitely many pieces that can then be rearranged to form a sphere of radius 2, a famous theorem by Banach and Tarski. This last result flies in the face of intuition, it seems obviously wrong.

If we accept the need for precision, the next question is: How should we go about achieving it? Here is an example of one possible approach.



Now we begin formally. What is a real number? It will be an expression of the form

$$\pm a_1 a_2 a_3 \ldots a_m . b_1 b_2 b_3 \ldots .$$

Here the ± represents a choice between plus and minus. The digit a_1 is an integer between 0 and 9 inclusive (unless m is different from 1 in which case it is restricted to being between 1 and 9, since it is the leading digit.) All other digits are integers between 0 and 9 inclusive.

The author claims to provide a formal definition, but does nothing of the sort. Our key objection is that the definition identities a real number with an *expression*, a denizen of StringWorld, albeit an infinite one. More precisely, the second line in the snippet above can be interpreted as a regular expression for ω-languages, see the chapter on automata theory for details. On that understanding, a real number is any string that matches this regular expression. In particular the real number π is just the string

$$3.14159265358979323846264338327950288 4197\ldots$$

where we have omitted a few digits at the end to save paper. In other words, a real number is no more than a string, a sequence of letters from a particular alphabet, subject to certain syntactic constraints. The alphabet is quite straightforward: it consists of the decimal digits, augmented by a decimal point and plus/minus signs. Furthermore, the constraints are so simple that they can be checked by a finite state machine. Ironically, there is no finite state machine that could check whether a given string represents a rational number. It seems difficult to take this definition seriously. Surely there is more to π or e or even just $\sqrt{2}$ than just a bunch of digits. These particular and well-known reals convey an idea, it feels too constricting and mechanical to think of them as no more than strings.

What exactly are the problems in StringWorld? For one thing, the choice of base 10 is quite artificial. It makes perfect sense for humans because of our anatomical features, but what does that have to do with the reals? If the mathematicians over at Alpha Centauri had 7 or 2 fingers, would they work with the same reals? Then there are pesky problems about how the rational numbers fit into this framework. The perfectly simple fraction 1/3 turns into the infinite string 0.33333… Multiplying by 3, we get 0.99999… which string needs to be equal to 1.00000… to make sense. Even more complicated is the problem of introducing arithmetical operations. Since the reals themselves here are just strings of digits, the operations must be based entirely on manipulating digits. That is admittedly of great interest when it comes to the design of algorithms, but it completely obscures the nature of these operations. Why should we use one way to mangle the digits and not another?

As we will see, there is a perfectly good way to build up all arithmetical operations starting at the naturals. The intuitive meaning of the number systems and the operations is nicely preserved, and everything has computational content: for example, the problem of multiplying two rationals directly reduces to operations on the natural numbers. For the reals, a bit more effort is needed, but the principle is very much the same. We prefer to interpret the expressions in the boxed definition as *representations* of actual real numbers. As such, they are very important, and can be used to explain all kinds of algorithms operating on reals. They even provide some hints at technical properties. For example, we can approximate any real to arbitrary precision by rationals: just truncate the infinite string. Still, the expressions are secondary, the reals come first.

4.2 Integers and Rationals

Our first two steps are fairly direct, we will use equivalence classes of natural numbers to define the integers, and then equivalence classes of integers to represent rationals. The equivalence classes themselves are infinite, but one can think of their basic elements as hereditarily finite sets.

4.2.1 Naturals to Integers

The natural numbers support addition and multiplication, but subtraction and division fail to be defined in general. Let's ignore multiplication for the time being and focus on addition. Over the natural numbers, we can only solve equations of the form

$$X + a = b \qquad : \mathbb{N}$$

when $a \leq b$, in which case the intuitive answer is $X = b \mathbin{\dot{-}} a$. Since we are dealing with different kinds of number systems in the following sections, we have added the type information $: \mathbb{N}$ to make clear exactly which domain we are interested in at this point.

How could we handle the case where $a > b$ so that $b \mathbin{\dot{-}} a = 0$ fails to be a solution? Think of the solution in the easy case as a pair of natural numbers (b, a), rather than the difference $b \mathbin{\dot{-}} a$. Here comes the critical idea: we declare this pair to be the right answer no matter what the order relation between a and b is. So the "solution" of $X + 5 = 2$ is $(2, 5)$.

There are several problems with this approach. First, we now have two kinds of numbers to deal with, ordinary naturals and our pair-integers. Intuitively, naturals should just be a special kind of integer. This is fairly straightforward to handle: we can identify a natural number b with a the pair $(b, 0)$. Second, we need to introduce arithmetical operations on our pair-integers. Addition is fairly easy, but multiplication takes a bit more effort.

$$(u, v) +_{\mathbb{N}^2} (x, y) = (u + x, v + y)$$
$$(u, v) *_{\mathbb{N}^2} (x, y) = (ux + vy, uy + vx)$$

We have written $+_{\mathbb{N}^2}$ and $*_{\mathbb{N}^2}$ to make clear that these are the new operations on pairs of natural numbers, as opposed to the standard addition and multiplication on $\mathbb{N}$ on the right hand side. To see that this makes sense, think about the case where the second component is 0; we duly get back ordinary arithmetic on the naturals. In the general case, cheat and rewrite the pairs as differences. Multiply everything out and collect the positive and negative terms and you will get the right hand side.

There is one other problem, though: some of our pair-integers have the same intuitive value: since $X + a = b$ has the same solution as $X + (a + d) = (b + d)$, the pairs (b, a) and $(b + d, a + d)$ should be the same. We can handle this identification by introducing an equivalence relation and then working with equivalence classes rather than single pairs:

$$(u, v) \equiv (x, y) \iff u + y = v + x$$

It is easy to check that this definition really produces an equivalence relation. We can now stipulate that integers are the equivalence classes of this relation.

Definition 4.2.1 *The integers are the quotient set*

$$\mathbb{Z} = \mathbb{N} \times \mathbb{N}/\equiv$$

It is important to realize that it is not quite enough that $\equiv$ is an equivalence relation, we need slightly more. We want to inherit our definitions of addition and multiplication from above:

$$[(u, v)]_{\equiv} +_{\mathbb{Z}} [(x, y)]_{\equiv} = [(u + x, v + y)]_{\equiv}$$

For this to make any sense, we need to show that changing (u, v) and (x, y) to equivalent pairs on the left hand side produces an equivalent result on the right hand side. This is expressed by saying that equivalence needs to be a congruence, a relation that is compatible with other operations. You will encounter congruences in several places in the future. In this case, we have to make sure that $(u+x, v+y) \equiv (u'+x', v'+y')$ whenever $(u, v) \equiv (u', v')$ and $(x, y) \equiv (x', y')$. So we have $u + v' = v + u'$ and $x + y' = x' + y$ and therefore

$$(u + x) + (v' + y') = (u + v') + (x + y') = (u' + v) + (x' + y) = (v + y) + (u' + x')$$

The argument for multiplication is entirely similar.

With this identification in place, we can solve all equations of the form

$$X +_{\mathbb{Z}} [(a, a')]_{\equiv} = [(b, b')]_{\equiv} \quad \rightsquigarrow \quad X = [(b + a', a + b')]_{\equiv}$$

It should be clear by now that writing integers as equivalence classes $[(u,v)]_\equiv$ and addition as $+_\mathbb{Z}$ is a major nuisance. To get a reasonable notation system, we choose convenient representatives for the classes: we write n for $[(n,0)]_\equiv$ and $-n$ for $[(0,n)]_\equiv$, and we drop the subscript $\mathbb{Z}$. The step forward from plain natural numbers is that now we can subtract in general:

$$n - m = n + (-m) = (n,0) + (0,m) = (n,m) = \begin{cases} (n \dot{-} m, 0) & \text{if } n \geq 0 \\ (0, m \dot{-} n) & \text{otherwise.} \end{cases}$$

A detractor might point out that our n and $-m$ notation is pretty much a relapse into StringWorld. Why don't we simply define the integers to be *expressions* of the form n and $-m$ and then explain how to do arithmetic on them? That would seem to require none of the heavy machinery above. But that is simply not true, the definitions of arithmetic operations then become much messier. Explaining addition and multiplication solely in terms of digit manipulations is a terrible idea, the nature of the operations becomes much clearer when the are based on the underlying properties of the actual number objects[1]. Proofs of basic properties such as associativity also become harder and provide no insight. In the end, nothing is gained and much is lost by setting up camp in StringWorld.

4.2.2 Integers to Rationals

Over the integers, we can handle additive equations, but we cannot generally solve equations of the form

$$X * a = b \qquad : \mathbb{Z}$$

In fact, this type of equation hardly ever has a solution: we need a to divide b, a rather rigid constraint. It should be no major surprise at this point that we can use a method similar to the one that took us from the naturals to the integers. Here, we would like to introduce fractions a/b for $a, b \in \mathbb{Z}$ and $b > 0$, rather than differences as in the last section. So, we fake fractions as pairs:

$$Q = \{\, (x,y) \in \mathbb{Z} \times \mathbb{Z} \mid y > 0 \,\}$$

and define a relation $\equiv$ on Q by

$$(u,v) \equiv (x,y) \iff uy = vx.$$

It is easy to see that $\equiv$ is an equivalence relation.

Definition 4.2.2 *The rationals are the quotient set*

$$\mathbb{Q} = Q/\equiv$$

Since we have seen the use of pesky equivalence classes in great detail in the last section, we will avoid them here and choose a member of a class as a representative for the whole class. Instead of using pairs, we follow convention and write elements of $\mathbb{Q}$ as fractions:

$$\frac{a}{b} \qquad \text{or} \qquad a/b$$

[1] More generally, it is almost always a bad idea to explain an operation in terms of an algorithm, and in particular in terms of a complicated algorithm. There has to be some purpose behind the operation that needs to be explained first, the algorithm is then a question of computational details. Think about the greatest common divisor and the Euclidean algorithm.

where (a, b) is an arbitrary representative; a is the numerator and b the denominator of the fraction. It turns out that there is a particularly nice representative. We say that $\frac{a}{b}$ is in lowest common terms or reduced if a and b are coprime. To convert a fraction a/b into reduced form, we have to compute the largest d that divides both a and b, the greatest common divisor. We can then factor $a = da'$ and $b = db'$ and the fraction a'/b' is equivalent to a/b and reduced.

To transfer the integers into the realm of rationals, we identify $a \in \mathbb{Z}$ with the reduced fraction $\frac{a}{1}$. Similarly we can lift the arithmetical operations from $\mathbb{Z}$ to $\mathbb{Q}$ as follows. Again, just this time, we will use a subscript to indicate the new domain:

$$\frac{u}{v} +_{\mathbb{Q}} \frac{x}{y} = \frac{uy + vx}{vy}$$
$$\frac{u}{v} *_{\mathbb{Q}} \frac{x}{y} = \frac{ux}{vy}$$

The operations in the numerators and denominators on the right are all over $\mathbb{Z}$, there is no circularity in these equations. Of course, we have to check that these definitions are compatible with our equivalence relation: replacing the fractions on the left by equivalent ones must produce equivalent ones on the right.

In general, the results in these operations will not be reduced, even if the the arguments are. We need to simplify them by calculating the greatest common divisor and factoring it out.

4.2.3 Exercises

Exercise 4.2.1 Explain how to implement subtraction on $\mathbb{Z} = \mathbb{N} \times \mathbb{N}/\equiv$.

Exercise 4.2.2 How would multiplication work on $\mathbb{Z}$?

Exercise 4.2.3 Show that each equivalence class contains exactly one fraction in lowest common terms.

Exercise 4.2.4 Check that the arithmetical operations on $\mathbb{Q}$ are well-defined (that they do not depend on the choice of representative).

Exercise 4.2.5 Explain how to define an order relation $<$ on the rationals.

4.3 The Reals

4.3.1 Gaps and Limits

Continuing with our project of finding number systems that can solve more and more equations, our next major obstacle is

$$X^2 = a \qquad : \mathbb{Q}$$

This type of equation has only trivial solutions in the sense that we need $a = r^2/s^2$ where $r, s \in \mathbb{N}$, $s \neq 0$. For example, the equation $x^2 = 2$ famously has no solution over $\mathbb{Q}$: informally, $\sqrt{2}$ is irrational. To see why, assume otherwise. Then there are coprime r and s

such that $2 = r^2/s^2$ and thus $2\,s^2 = r^2$. Since 2 is prime, it divides r and we get $s^2 = 2\,r_0^2$. But then 2 also divides s, a contradiction.

While $\sqrt{2}$ fails to be rational, we can approximate it to arbitrary precision. The following table shows a few approximations (a_n) with an accuracy of 10^{-n}; the fractions are chosen in a way that uses the smallest denominator that produces this accuracy. Think about how one could compute these fractions, it is quite challenging.

n	a_n	Δ
0	1	0.414213562373
1	$\frac{3}{2}$	0.085786437626
2	$\frac{17}{12}$	0.002453104293
3	$\frac{41}{29}$	0.000420458924
4	$\frac{99}{70}$	0.000072151912
5	$\frac{577}{408}$	$2.123901414755 \times 10^{-6}$
6	$\frac{1393}{985}$	$3.644035519015 \times 10^{-7}$

Geometrically, we can think of $\sqrt{2}$ as being a gap in the rationals. Hence, our task of constructing the reals comes down to filling all these gaps. Of course, we also have to make sure that arithmetic carries over to the new set of numbers. There are two basic ways to tackle this problem, one based on working directly with approximating sequences, and the other by using a set of approximations. These two methods will be discussed in the following sections.

Cauchy Sequences

The approximation sequence (a_n) from above converges to $\sqrt{2}$, it has $\sqrt{2}$ as its limit, in the following technical sense: $|\sqrt{2} - a_n| < 10^{-n}$. This condition is a bit rigid, it would be enough to have

$$\forall\, \varepsilon > 0\, \exists\, \ell\, \forall\, n \geq \ell \left(|\sqrt{2} - a_n| < \varepsilon\right)$$

Given some precision ε, we can omit finitely many terms from the front of the sequence, so that all remaining terms are closer to $\sqrt{2}$ than ε.

Now pretend for a moment that we already have an extension $\mathbb{R}$ of $\mathbb{Q}$. What can we say about sequences of rationals that converge to some real r? Since the terms get arbitrarily close to r, they must also get arbitrarily close to each other. Such sequences are called Cauchy sequences and are defined like so:

Definition 4.3.1 (Cauchy Sequences) *A sequence (a_n) of rationals is Cauchy if*

$$\forall\, \varepsilon > 0\, \exists\, \ell\, \forall\, n, m \geq \ell \left(|a_n - a_m| < \varepsilon\right)$$

It is easy to see that every convergent sequence must be Cauchy.

Here is the key idea: we want exactly all Cauchy sequences to converge. Once the goal has been stated this way, a radical solution comes into view: we could *declare* all Cauchy sequences of rationals to be the reals. As stated, this will not quite work since multiple Cauchy sequences may have the same limit. For example, the sequences $(0)_n$, $(2^{-n})_n$, $((-1)^n 2^{-n})_n$ and $(10^{-n})_n$ all have limit 0. Such sequences are (Cauchy) null sequences. It is intuitively clear that, given a Cauchy sequence (a_n) and a null sequence (ν_n), the sequence $(a_n + \nu_n)$ is also Cauchy and should have the same limit. To express this idea more formally, it is best to use yet another equivalence relation: two Cauchy sequences (a_n) and (b_n) are said to be (Cauchy) equivalent if

$$\forall \varepsilon > 0 \, \exists \ell \, \forall n, m \geq \ell \, \big(|a_n - b_m| < \varepsilon \big).$$

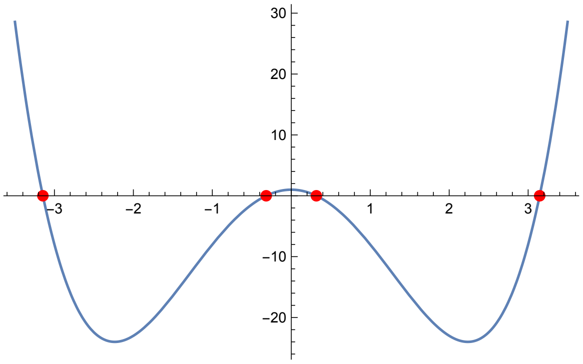
Now we can formally represent the reals by convergent sequences of rationals.

Definition 4.3.2 (Real Numbers)
The reals $\mathbb{R}$ are the set of all Cauchy sequences over the rationals, modulo Cauchy equivalence.

This definition is quite a bit more abstract and impenetrable than our previous examples. To make sure we still have the rationals available, note that constant sequences $(a)_{n\geq 0}$ are trivially Cauchy. And any Cauchy sequence converges to its own equivalence class, no problem. The next step is to define a reasonable arithmetic on the reals. As before, we want to lift addition and multiplication from the rationals. As usual, we spell out the new operations just once. Also, we will ignore annoying equivalence classes and set

$$(a_n) +_{\mathbb{R}} (b_n) = (a_n + b_n)$$
$$(a_n) *_{\mathbb{R}} (b_n) = (a_n * b_n)$$

One needs to show that the sequences on the right are still Cauchy, and that Cauchy equivalence is a congruence. Beyond these sanity checks, we also want to make sure the definitions align with our intuition. For example, the $\sqrt{2}$ approximation sequence (a_n) from above should produce $(a_n) \cdot (a_n) = (2)$, up to Cauchy equivalence. In a similar manner, the reals have square roots for all positive rationals. We even have roots of a great many polynomials with rational coefficients. For example, any odd-degree polynomial such as is guaranteed to have a root and some even degree ones do, too. For instance, the polynomial $1 - 10x^2 + x^4$ has the following 4 real roots: $-\sqrt{2} - \sqrt{3}, \sqrt{2} - \sqrt{3}, \sqrt{3} - \sqrt{2}, \sqrt{2} + \sqrt{3}$



This looks promising, but there is still one vexing question: are there any gaps in the reals? In other words, could there be Cauchy sequences of reals, rather than rationals, whose limits fail to be real? To answer this question, we need a bit more terminology to talk about possible gaps. A set $X \subseteq \mathbb{R}$ is bounded (from above) if there is some real b such that $x \leq b$ for all $x \in X$. Similarly we can talk about a set be bounded from below. Now suppose X is bounded. A real B is a least upper bound for X if it is a bound, and any other bound is already larger than B:

$$\forall x \in X \, (x \leq B) \wedge \forall b \, \big(\forall x \in X \, (x \leq b) \Rightarrow B \leq b \big)$$

If a least upper bound exists it is also called the supremum and written $\sup X$. The existence of least upper bounds is arguably the critical property of the reals.

Theorem 4.3.1 *Every bounded set in $\mathbb{R}$ has a least upper bound.*

We will not give a proof here, take a look at the next section how this result can be handled very easily given a different definition of the reals. We can define lower bounds, the greatest lower bound or infimum in an entirely symmetric way; an analogous result holds for those. Using this upper bound principle, we can show that our construction works as intended, there are no gaps left in the reals.

Claim: Every Cauchy sequence of reals converges to a real.

Here is a sketch of the proof. Consider a Cauchy sequence (r_n) of reals and choose, for each n, a rational Cauchy sequence $(a^n_m)_{m \geq 0}$ such that $\lim_{m \to \infty} a^n_m = r_n$. It is tempting to take the limit of the rational diagonal sequence (a^n_n) with the hope of showing that this limit is the same as the limit for (r_n). Alas, this fails: the diagonal $(a^n_n)_{n \geq 0}$ may not be a Cauchy sequence. The problem is that the rate of convergence of the individual sequences $(a^n_m)_{m \geq 0}$ may get slower and slower as n increases.

To address this problem, we can show that each real number can be obtained by a special Cauchy sequence (b_n) such that

$$\forall\, k \in \mathbb{N}\, \forall\, n, m \geq k \left(|b_n - b_m| < 2^{-k}\right)$$

By choosing (r_n) and the approximating sequences to be of this particular type, we can check that the rational diagonal sequence (a^n_n) is Cauchy, and has the right limit.

Dedekind Cuts

The machinery of Cauchy sequences fits nicely with standard methods in analysis. Still, it is a bit messy and requires an uncomfortable level of abstraction: an equivalence class of Cauchy sequences is not a very intuitive object. One might wonder whether there is another, hopefully less painful, way to construct the reals. There is, but it relies more directly on set theory and one might feel that the definitions of arithmetical operations is less elegant than in the Cauchy sequence approach. For what it's worth, here is an alternative method due to Richard Dedekind where a real is represented by a plain set of rationals.

Definition 4.3.3 *A Dedekind cut is a subset $\alpha \subseteq \mathbb{Q}$ of the rationals such that $\emptyset \neq \alpha \neq \mathbb{Q}$, α is downward closed, and has no largest element.*

The set of reals $\mathbb{R}$ is the collection of all Dedekind cuts.

Downward closed means that $x < y$, $y \in \alpha$, implies that $x \in \alpha$. There are two basic possibilities: the complement $\mathbb{Q} - \alpha$ has a least element, or it does not. In the first case, α just represents the rational number $\min \mathbb{Q} - \alpha$; in the second case, however, we get an irrational number. Using our old $\sqrt{2}$ example, we have the Dedekind cut $\alpha = \{\, x \in \mathbb{Q} \mid x^2 < 2 \,\}$ that represents $\sqrt{2}$.

We can immediately see one advantage of this approach: there are no equivalence classes, there is exactly one Dedekind cut for each real. As before, we can embed $\mathbb{Q}$ into $\mathbb{R}$: write $q^{\downarrow} = \{\, r \in \mathbb{Q} \mid r < q \,\}$ for any rational q. Then the map $q \mapsto q^{\downarrow}$ is an embedding. Similarly it is easy to define the usual order on $\mathbb{R}$:

$$\alpha < \beta \quad \Leftrightarrow \quad \alpha \subset \beta$$

So far, this is quite a bit easier than having to wrestle with Cauchy sequences. As one might suspect, there must be some aspect where Dedekind cuts are messier than Cauchy sequences: the definition of arithmetic becomes a bit more involved. Here is addition:

$$\begin{aligned} \alpha + \beta &= \{\, x + y \mid x \in \alpha, y \in \beta \,\} \\ -\alpha &= \{\, x \mid \exists\, r < 0\, (r - x \notin \alpha) \,\} \end{aligned}$$

One can check that $-\alpha$ is indeed a cut, and $(-\alpha)+\alpha = 0^{\downarrow}$. Similarly, multiplication requires more work: we need to resort to distinguishing the sign of a real. We may safely assume that $\alpha, \beta > 0$, otherwise we can use the additive inverse. E.g., $\alpha * (-\beta) = -(\alpha * \beta)$.

$$\alpha * \beta = \{\, x * y \mid x, y > 0, x \in \alpha, y \in \beta \,\} \cup \mathbb{Q}_{\leq 0}$$

One can check that this works as intended, see the exercises, but it is a bit strange.

We end on a high note: the least upper bound principle is totally straightforward in this context.

Claim: Every bounded set in $\mathbb{R}$ has a least upper bound.

Let A be a bounded set of Dedekind reals. Since our reals are just Dedekind cuts, we can define $\beta = \bigcup A$. We assume A to be bounded, so β is also a cut. Certainly $\alpha \in A$ implies $\alpha \subseteq \beta$, whence β is an upper bound for A. Let γ be another upper bound. But then we must have $\alpha \subseteq \gamma$ for all $\alpha \in A$, so $\beta \leq \gamma$.

Similarly, it is easy to see that trying to apply the cut construction a second time to $\mathbb{R}$ will not produce anything new, there are no gaps in the Dedekind reals.

4.3.2 Comments

Since we have offered two ways to construct the reals, we hasten to point out that the result of both constructions is essentially the same: the two structures are isomorphic. This means that we can translate back and forth between Cauchy reals and Dedekind reals, and the translation is fully compatible with the arithmetic. The two methods appear to be rather dissimilar, but they are actually closely related. For example, suppose α is a Dedekind cut. To obtain a corresponding Cauchy sequence we can choose any sequence (a_n) that is cofinal in α, meaning that (a_n) is strictly increasing and

$$\forall\, a \in \alpha\, \exists\, \ell\, \forall\, n \geq \ell\, \big(a_n > a\big)$$

The opposite direction is a bit more complicated, try it. At any rate, the fact that both constructions yield essentially the same result is a good sign, our reconstruction of the reals as sets appears to be fairly robust.

Recall the StringWorld definition of reals that we mentioned in the introduction, where real numbers were defined in terms of a regular expression:

$$\pm a_1 a_2 \ldots a_m . b_1 b_2 b_3 \ldots$$

Supposedly this produces yet another way to define the reals. Given our construction of the reals, we can now explain and justify this representation. First, we may safely assume out given real x is non-negative, otherwise we can simple switch the sign. Second, let $n \leq x$ be the largest natural number not larger than x, the integer part of x. The decimal expansion of n will produce the a_i digits and is unique as long as we rule out leading 0s. The only interesting part is the fractional part $r = x - n$ of x. To obtain the first digit b_1, we compute the integer part of $10r$ which is guaranteed to be a decimal digit since $0 \leq 10r < 10$. We replace r by $r/10$ and continue. We have not fully developed arithmetic on the reals, but one can verify that all these operations are perfectly justified and produce the results that one would expect intuitively. Incidentally, computing the digits this way automatically eliminates the trailing-9s problem.

To be clear, when it comes to using the reals, the decimal expansion is quite useful and much easier to handle than our towers-of-sets representation. But now we can easily show that, say, the choice of base 10 is quite irrelevant and we can justify digit-based arithmetical algorithms in terms of our prior definitions of arithmetical operations on the reals. We do not have to blindly manipulate infinite strings, hoping that the operations somehow make sense.

As before when we reconstructed the reals as sets, we do not claim that set-theoretic definitions somehow explain the nature of the corresponding concept in a profound ontological sense, or even, more pragmatically, in a cognitively appealing way. The point is that a well-constructed set-theoretic definition can help greatly to clarify the concept in question. It does not in any way replace the need for an intuitive interpretation. Lastly, there are perfectly reasonable questions in analysis where our intuition ultimately fails. For example, are all sets of real Lebesgue measurable? In the presence of the Axiom of Choice the answer is no, but, giving up on this axiom (and assuming the existence of a large cardinal), one can construct a universe where all sets of reals are measurable. This was shown by Robert Solovay in 1970 and promptly ignored in the world of analysis.

4.3.3 Exercises

Exercise 4.3.1 Carry out the details of the construction of the reals using Cauchy sequences. Make sure to give a proof of the LUB principle.

Exercise 4.3.2 Show that for any Cauchy sequence, there are uncountably many others that converge to the same limit.

Exercise 4.3.3 Check that Dedekind's construction conforms nicely to intuiting about the reals.

Exercise 4.3.4 Show that the Dedekind cut definition of $\mathbb{R}$ produces a field that is isomorphic to the one obtained by Cauchy sequences.

Five

Induction and Rectypes

In this chapter we are going to take a closer look at induction and recursion. We have briefly encountered both concepts earlier on, but they are of such fundamental importance in both mathematics and computer science that a detailed discussion is required. We will start from the computer science perspective where inductively defined data types are both ubiquitous and easily motivated and understood. In the second part of the chapter we will try to understand the mathematical principles that induction rest on. Both ideas will be front and center again in the chapters on computation.

5.1 Induction Systems

The natural numbers are classical example of a structure that supports induction and recursion, as laid out in section 1.2.2: we can think of the naturals as being defined in terms of a special element 0 and a successor operation S. In this framework, the natural numbers take the form

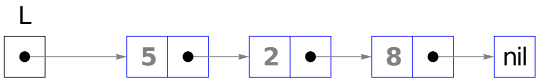
$$\mathbb{N} = \bigcap \{\, X \mid 0 \in X, X \ S\text{-closed} \,\}$$

where S-closed means that $x \in X$ implies that $Sx \in X$. We can then define, say, addition by

$$\mathsf{add}(0, y) = y \qquad \mathsf{add}(Sx, y) = S(\mathsf{add}(x, y))$$

How exactly does this work? We can think of 0 as being an atom, an indecomposable object, and of S as a constructor, an operation that builds more complicated objects from simpler ones. The intersection in the definition of $\mathbb{N}$ is absolutely essential, we need it to make sure that no unwarranted objects wind up in $\mathbb{N}$; we do not intend for our natural numbers to contain a beermug nor a kitchen sink. More seriously, assume we have a copy of the set of integers $\mathbb{Z}$ that is disjoint from $\mathbb{N}$ and let $\mathbb{N}' = \mathbb{N} \cup \mathbb{Z}$, so the $0 = 0_{\mathbb{N}}$ in X comes from $\mathbb{N}$ but not from $\mathbb{Z}$. The successor operation S on $\mathbb{Z}$ is defined in the obvious way. So we have 0 and successor on $\mathbb{N}'$, but our recursive definition of addition fails since $\mathsf{add}(z, y)$ is undefined for all $z \in \mathbb{Z}$: a call to $\mathsf{add}(-5, 42)$ would result in an infinite sequence of calls involving all $z < -5$, it would never terminate. Taking the intersection of all candidate sets X avoids precisely this problem.

Here is another domain where this approach works perfectly well, linked lists. These lists are inevitably represented by a little picture like the following:



The list L here is comprised of elements 5, 2 and 8, in this order. These data structures may seem fairly quaint today, but at the time of their invention in the 1950s they were a bit of a breakthrough. For instance, linked lists are essential for McCarthy's Lisp. We can describe linked lists of integers without reference to implementation details like pointers (the arrows in the picture) as follows:

- nil is a linked list, the empty list,
- for any linked list L and any integer a, $\mathsf{cons}(a, L)$ is a linked list.

Informally, this means that we can prepend integer a to an already existing list L; cons is the constructor that performs this operation. The analogy to our definition of the naturals is quite striking. We can now define other operations on linked lists in the same manner that we defined arithmetic operations on natural numbers. For example, the append operation can be defined by recursion like so:

$$\begin{aligned}\mathsf{app}(a, \mathsf{nil}) &= \mathsf{cons}(a, \mathsf{nil})\\ \mathsf{app}(a, \mathsf{cons}(b, L)) &= \mathsf{cons}(b, \mathsf{app}(a, L))\end{aligned}$$

We need to show that this append operation always terminates. To do this, we could resort to an argument about the length of the list (which, incidentally, can also be defined by recursion). In other words, we could use induction on the naturals to argue about linked lists, but there is no need for this detour: we can directly show that induction works on linked lists.

This "zero plus successor" approach applies to other structures such as trees or certain kinds of graphs. Could there be anything else, some sort of inductive structure that does not fit this pattern? The answer is a resounding Yes, as a first example we will indicate how to solve the problem of demonstrating that the so-called Ackermann function $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is total, see section 8.1.5 for more details. The Ackermann function is a classical example of an arithmetical function that is clearly computable, but cannot be defined by simple recursion. To make the definition a little more legible, we write x^+ rather than $x + 1$ to indicate where recursion takes place. The function is given by three equations, the last two of which use recursion:

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

Note the double recursion in the third equation, this is the critical part of the definition. It is far from clear from this definition that A has a value for all $x, y \in \mathbb{N}$, and that the value is uniquely determined. In fact, for general systems of recursive equations like those in the definition of Ackermann, it is undecidable whether they work as expected.

We can think of A as being given by a two-dimensional table, albeit with infinitely many rows and infinitely many columns. The question is, how do we fill in this table according to the equations? This problem arises frequently in computer science, though usually in the context of finite tables, and is called dynamic programming, a form of bottom-up recursion. The key in dynamic programming is to find the proper order in which to fill the table, so that some entries are already available when another entry depending on them is added. Informally, for the Ackermann function, we fill the table row-by-row and each row from left to right. Inspecting the equations, we can see that all the values on the right-hand side have already been entered into the table when we work on the entry for the left-hand side. More formally, we could prove this by an induction on x and sub-induction on y, each part being an ordinary induction on $\mathbb{N}$.

If we had 3 variables involved in the recursion, there would need to be a sub-sub-induction, and so forth. An elegant way to avoid these cumbersome nested inductions is to replace them by a single transfinite induction. In other words, we want to replace plain induction on the natural numbers by an analogous argument over some more complicated "inductive structure." How would we go about building such a structure? Suppose we replace every single natural number m by a copy $\mathbb{N}_m$ of the natural numbers

$$\mathbb{N}^{(2)} = \underbrace{\mathbb{N}_0, \mathbb{N}_1, \mathbb{N}_2, \ldots, \mathbb{N}_m, \ldots}_{\mathbb{N}}$$

and write $n^{(p)}$ for the copy of the natural number n in $\mathbb{N}_p$. In other words, we have flattened out the lookup table form above in row-major order.

There is a natural order on this flattened structure that extends the standard order on $\mathbb{N}$: let $m^{(p)} \prec n^{(q)}$ if $p < q$ or $p = q$ and $m < n$. This is exactly the order in which we encounter the elements of $\mathbb{N}^{(2)}$ if we traverse the copies of $\mathbb{N}$ from "left-to-rigth." In this order, $n^{+(p)}$ has the unique predecessor $n^{(p)}$, but note that none of the elements $0^{(p)}$ have predecessors, we can only get closer and closer; the whole traversal is indeed a transfinite process. We can now define a function α on $\mathbb{N}^{(2)}$ that mimics the Ackermann function as in $\alpha(n^{(p)}) = A(p, n)$ by the same traversal process. For the first copy $\mathbb{N}_0$ the first equation determines the values $\alpha(n^{(p)})$ directly. At higher levels p^+ we have $\alpha(0^{(p^+)}) = \alpha(1^{(p)})$ by the second equation, a value that is already determined since $1^{(p)}) \prec 0^{(p^+)}$. Lastly, $\alpha(n^{+(p^+)}) = \alpha(\alpha(n^{(p^+)})^{(p)})$ by the third equation, and the values on the right are again determined since $m^{(p)} \prec 0^{(p^+)} \prec n^{+(p^+)}$ for all m, and $n^{(p^+)} \prec n^{+(p^+)}$.

5.1.1 Well-Founded Relations

We will now generalize this idea of flattening out nested recursions into a single, transfinite order. In particular, we will give a formal description of the constraints we have to impose on an order to allow for induction and recursion.

Definition 5.1.1 *A strict order $\prec$ on A is a well-founded if every non-empty subset $X \subseteq A$ contains a $\prec$-minimal element. If $\prec$ is in addition a total order it is called a well-order.*

The assertion that every non-empty subset has a minimal element is also called the Least-Element Principle (LEP). Every finite order is necessarily well-founded. For assume otherwise, say, we have some strict order $\prec$ on a finite set A and $\emptyset \neq B \subseteq A$ has no minimal element. Pick $a_0 \in B$, by our assumption there is some $a_1 \in B$ such that $a_0 \succ a_1$. Continuing by induction we obtain an infinite descending chain (IDC)

$$a_0 \succ a_1 \succ a_2 \succ \ldots \succ a_n \succ \ldots$$

contradicting the finiteness of A. More importantly, the argument we just used provides a characterization of well-foundedness for all carrier sets.

Lemma 5.1.1 (AC) *Let $\prec$ be a strict order on some set A. Then $\prec$ is well-founded if, and only if, there is no infinite descending chain with respect to $\prec$.*

Proof. First assume that (x_n) is a strictly descending chain. The corresponding set $\{ x_n \mid n \geq 0 \}$ is not empty, but clearly fails to have a minimum element.

On the other hand, suppose $\prec$ fails to be well-founded, say, $\emptyset \neq X \subseteq A$ has no minimum element. Pick any element x_0 in X. Then there must be some $x_1 \prec x_0$ in X. Continuing inductively, we obtain a descending chain. □

Just to be clear, the induction in the last argument is on $\mathbb{N}$, not on A. A key question in the early development of set theory was whether every set can be well-ordered. Cantor thought this was "self-evident" because we can construct the required order in stages. More precisely, to well-order A, construct a sequence a_α in (transfinitely many) stages like so:

$$a_\alpha = \text{ pick some } x \in A - \{\, x_\beta \mid \beta < \alpha \,\}$$

To turn this into a weight-bearing construction, we need to explain precisely what the "stages" are supposed to be, see section 5.2. The second problem is the "pick some" operation: A is an abstract set and there is no clear mechanism how this choice should be made. In fact, even a rather concrete set such as the powerset of the naturals causes problems, it is entirely unclear how one should pick "the next" set of natural numbers. Our old friend, the Axiom of Choice, takes care of that issue.

To test whether some order is well-founded we can either apply the LEP test or the IDC test. The latter works well for all kinds of orders, at least at the intuitive level. For example, it is clear that the standard order on $\mathbb{N}$ is a well-order, but its extension to $\mathbb{Z}$ is not, never mind $\mathbb{Q}$ or $\mathbb{R}$. A slightly more complicated example is the order $\prec$ from the last section: there is no IDC in any one copy $\mathbb{N}_p$, so trying to build such a chain in $\mathbb{N}^{(2)}$ would force us to move into $\mathbb{N}_q$ for some $q < p$ at some point. Since $\mathbb{N}$ is well-ordered, we can do this only finitely often.

Returning to the Ackermann function from above, we can exploit LEP like so: suppose α is not defined somewhere, there is a non-empty set of counterexamples to the claim that the function is total. By LEP, let $n^{(p)}$ be the least counterexample. Then α is defined on all the ancestors of $n^{(p)}$ and that suffices to force a unique value of α for $n^{(p)}$. This argument presents the bottom-up view: we have constructed α in stages along the order $\prec$ on $\mathbb{N}^{(2)}$. An alternative perspective is to think about the recursion equations more directly: the only way a call to α could fail is that it could produce an infinite sequence of calls to smaller and smaller points in $\mathbb{N}^{(2)}$. But that produces an IDC, contradicting well-foundedness.

So far, we have based our definitions on strict orders, any relation that is reflexive even at a single point automatically has a descending chain and cannot be a well-order. Still, on occasion it can be more convenient to use reflexive orders. The well-foundedness condition that any non-empty subset have a minimal element makes sense in this context, too. We do have to modify our chain condition slightly: we insist that any non-increasing infinite sequence (a_n) is ultimately constant or ultimately stationary: $\exists\, m\, \forall\, n, n' \geq m\, (a_n = a_{n'})$.

What is the promised connection between well-founded orders and induction? Recall that $a^{\downarrow} = \{\, x \mid x \prec a \,\}$ is the set of all ancestors of a. Call a subset X of A inductive (wrto $\prec$) if

$$\forall\, a \in A\, \big(a^{\downarrow} \subseteq X \Rightarrow a \in X\big).$$

In other words, element x inherits property X from its ancestors. This is often written more explicitly as $\forall\, a \in A\, \big(\forall\, z \prec a\, (z \in X) \Rightarrow a \in X\big)$.

Those who have been bludgeoned with endless induction arguments on $\mathbb{N}$ might feel that the base case is missing, but note that all $\prec$-minimal elements of A must be in A: for any $\prec$-minimal element a the ancestor set $a^{\downarrow}$ is empty, so $a \in X$.

Theorem 5.1.1 *Let X be an inductive subset of a well-founded set A. Then $X = A$.*

Proof. Assume for the sake of a contradiction that $A \neq X$. Then $A - X$ is non-empty and therefore contains a minimal counterexample a. But then $a^{\downarrow} \subseteq X$ by minimality of a, and thus $a \in X$ by inductiveness of X. Contradiction. □

For us, the number of ancestors $a^{\downarrow}$ involved in dealing with some particular a is usually finite, but note that this is not a requirement in the last theorem. Given the importance of well-founded relations it is naturally of interest to try to analyze their properties in greater detail. We mention only one result that shows how to combine already available well-founded relations into new ones.

Lemma 5.1.2 *If ρ and σ are well-founded relations on A and B, respectively, then their direct product as well as their Cartesian product are well-founded on $A \times B$.*

The proof is left as an exercise.

5.1.2 Recursive Datatypes

The examples of the natural numbers and lists from above are just the tip of an iceberg. In both mathematics and computer science there are a great many inductive structures or inductively defined sets, also known as recursive datatypes or simply rectypes in computer science[1]. In general, to describe a rectype, we need a specification τ consisting of

- a collection At of atoms, and
- a collection Cstr of constructors.

The atoms are indecomposable from the perspective of the rectype, but objects built with the help of a constructor can be decomposed. We could think of atoms as nullary constructors, but that seems a bit more contrived. In order to avoid set-theoretic problems with collections that do not technically qualify as sets, we insist that there is an ambient set A such that $\mathsf{At} \subseteq A$ and, for any constructor $C \in \mathsf{Cstr}$, $C : A^k \to A$ for some k, the arity of the constructor. Given a rectype $\mathcal{X} = \langle A, \mathsf{At}, \mathsf{Cstr} \rangle$, we call $X \subseteq A$ $\mathcal{X}$-closed or $\mathcal{X}$-inductive if it contains all the atoms, and is closed under the constructors in the sense that $x_1, x_2, \ldots, x_k \in X$ implies $C(x_1, \ldots, x_k) \in X$ for any constructor C of arity k. The least $\mathcal{X}$-closed subset of A is

$$\bigcap \{\, X \subseteq A \mid X \ \mathcal{X}\text{-closed} \,\}$$

and we will abuse notation by also writing $\mathcal{X}$ for this set. Again, in the interest of avoiding beermugs and the like, we take the intersection of all candidates.

By slight abuse of language, we call $\mathcal{X}$ the rectype (or inductively defined set) defined by τ (we construe τ as the specification, and $\mathcal{X}$ as the implementation). If you worry where atoms and constructors fit into our set-theoretic universe, think of atoms as elements in $\mathbb{V}$, and constructors as maps $\mathbb{V}^k \to \mathbb{V}$. In the RealWorld™, we typically can constrain atoms and constructors to some smaller ambient domain such as the natural numbers, words over an alphabet, subsets of the reals, and so on. In an axiomatic setting one has to be careful to avoid any confusion between classes and sets, but we will casually ignore these difficulties and pretend that the whole universe $\mathbb{V}$ is fine as an ambient "set."

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

The examples of the natural numbers and lists from above are just the tip of an iceberg. In both mathematics and computer science there are a great many inductive structures or inductively defined sets, also known as recursive datatypes or simply rectypes in computer science[1]. In general, to describe a rectype, we need a specification τ consisting of

[1] Rectype is a neologism that we have adopted from T. Forster; it is fairly nonstandard, but it fits perfectly and should be in wider use.

[1] Rectype is a neologism that we have adopted from T. Forster; it is fairly nonstandard, but it fits perfectly and should be in wider use.

- a collection of atoms, and
- a collection of constructors.

We could think of atoms as nullary constructors, but that seems a bit more contrived. The atoms are indecomposable from the perspective of the rectype, but objects built with the help of a constructor can be decomposed. A set X of objects is τ-closed or τ-inductive if it contains all the atoms, and is closed under the constructors in the sense that $x_1, x_2, \ldots, x_k \in X$ implies $C(x_1, \ldots, x_k) \in X$ for any constructor C of arity k. In order to avoid set-theoretic problems with collections that do not technically qualify as sets, we construct inductive sets as subsets of a given ambient set A: the least τ-closed subset of A is

$$\mathcal{X} = \bigcap \{\, X \subseteq A \mid X \text{ } \tau\text{-closed} \,\}$$

Again, in the interest of avoiding beermugs[2], we take the intersection of all candidates. By slight abuse of language, we call $\mathcal{X}$ the rectype (or inductively defined set) defined by τ (we construe τ as the specification, and $\mathcal{X}$ as the implementation). If you worry where atoms and constructors fit into our set-theoretic universe, think of atoms as elements in $\mathbb{V}$, and constructors as maps $\mathbb{V}^k \to \mathbb{V}$. In the RealWorld™, we typically can constrain atoms and constructors to some smaller ambient domain such as the natural numbers, words over an alphabet, subsets of the reals, and so on. In an axiomatic setting one has to be careful to avoid any confusion between classes and sets, but we will casually ignore these difficulties and pretend that the whole universe $\mathbb{V}$ is fine as an ambient "set."

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

However, our framework is perfectly general enough to deal with infinitely many predecessors. The ability to handle arbitrarily large predecessor sets is of lesser importance in computer science, but in mathematics it is crucial. For example, Borel sets are constructed in a way that requires the use of infinitely many previously constructed sets: we start with the open sets in some topology, and then repeat the operations of complement and countable union transfinitely often. There are even echos of this sort of argument in computability theory, but for the most part we will be interested in the finitary situation.

In the world of set theory, it makes perfect sense to consider constructors with infinitely many arguments. For instance, in analysis, Borel sets are defined as a rectype: The open subsets of $\mathbb{R}$ are the atoms, the constructors are complement and countable unions. Since we focus on results relevant for computer science, we will only consider finitary rectypes, where all constructors take only finitely many arguments. If, in addition, there are only finitely many atoms and only finitely many constructors, the rectype is finitely presented.

As an example, consider the following definition of an implicational formula in the grammar style format that is rather popular in computer science.

$$\texttt{fml} ::= p \mid q \mid r \mid \bot \mid \texttt{fml} \Rightarrow \texttt{fml}$$

This defines a `fml` recursively by listing possible atoms (the constant $\bot$ and the propositional variables p, q and r) and by specifying a single constructor $\Rightarrow$ of arity 2. Thus, we have a finitely presented rectype. Of course, that's not quite right, we should allow for infinitely many propositional variables. For example, we could generate countably many propositional variables by attaching a binary string to, say, the symbol p.

$$\texttt{pvar} ::= p \mid \texttt{pvar}\, 0 \mid \texttt{pvar}\, 1$$

[2] I grew up in Munich, so I don't say this lightly.

Combining these two grammars, we get a perfectly good finitely presented rectype for implicational formulae of propositional logic.

$$\begin{aligned}\texttt{fml} &::= \texttt{pvar} \mid \bot \mid \texttt{fml} \Rightarrow \texttt{fml}\\ \texttt{pvar} &::= p \mid \texttt{pvar}\,0 \mid \texttt{pvar}\,1\end{aligned}$$

It is straightforward to extend this rectype to include other logical connectives. In a similar manner we can define simple expressions of arithmetic in prefix form as follows.

$$\texttt{exp} ::= \texttt{var} \mid \texttt{cnst} \mid +\,\texttt{exp}\,\texttt{exp} \mid *\,\texttt{exp}\,\texttt{exp}$$

where `var` and `cnst` are other rectypes standing for variables and constants, respectively.

The critical tool that makes all these rectypes interesting and eminently useful is induction. Since we are only dealing with finitary rectypes $\mathcal{X}$, we can organize their elements nicely into a hierarchy as follows according to their rank of appearance. Define $\mathcal{X}_0$ to be the atoms in the rectype, and

$$\mathcal{X}_n = \left\{ C(b_1, \ldots, b_k) \,\middle|\, b_i \in \bigcup\nolimits_{j<n} \mathcal{X}_j, C \text{ } k\text{-ary constructor} \right\}$$

If X is any τ-closed set, an easy induction shows that $\mathcal{X}_n \subseteq X$ for all n. But $\bigcup \mathcal{X}_n$ is τ-closed since all our constructors take only finitely many arguments, so we can conclude that $\mathcal{X} = \bigcup \mathcal{X}_n$. As an immediate consequence, we have the next theorem.

Theorem 5.1.2 (Structural Induction) *Suppose $\mathcal{X}$ is a finitary rectype, and $X \subseteq \mathcal{X}$ is τ-closed. Then $X = \mathcal{X}$.*

As we have seen, well-founded relations are critical for induction, so we should try to tease out the order relation connected to a rectype. We start by declaring $b_i \prec C(b_1, \ldots, b_k)$ and then take the transitive closure of this relation; by minor abuse of notation we will write $\prec$ for both. It is easy to check that $\prec$ is well-founded, so our structural induction theorem is perfectly justified. How about recursion? Having all calls to a lead to elements in the ancestor set is not quite enough: we need to make sure that compound objects are uniquely decomposable. In other words, we need not just constructors, but also destructors. This is quietly understood in the description of rectypes we gave above, but it needs to be stated clearly, in particular if one thinks about constructors as being functions. Say, we want to define an operation Θ on some rectype. The natural way to do this is to

- define Θ on all atoms, and
- explain how $\Theta(C(b_1, \ldots, b_k))$ is obtained from $\Theta(b_1)$, …, $\Theta(b_k)$.

For example, to obtain a measure of depth of a `fml` from above, we could set $\Theta(p) = \Theta(\bot) = 0$ and $\Theta(\varphi \Rightarrow \psi) = \max(\Theta(\varphi), \Theta(\psi)) + 1$. Or we could define the length of a list by $\Theta(\mathsf{nil}) = 0$, $\Theta(\mathsf{cons}(a, L)) = \Theta(L) + 1$. In both cases there are obvious destructors, say, $\mathsf{hypo}(\phi \Rightarrow \psi) = \phi$ and $\mathsf{conc}(\phi \Rightarrow \psi) = \psi$. On the other hand, consider the rectype where the atoms are the primes and multiplication is the sole constructor. Even though there is a unique prime decomposition, we have $2 \cdot (3 \cdot 5) = (2 \cdot 3) \cdot 5$, so there is no unique way to reverse multiplication. As a real world example, let's return to Borel sets of reals. We have

$$\overline{(\infty, 0)} \cap \overline{(0, \infty)} = \{0\} = \bigcap_{n>0} \overline{(-\infty, 0) \cup (1/n, \infty)}$$

and there is nothing resembling a unique decomposition into "smaller" components. This is not a problem for induction, we can still prove that every Borel set is measurable.

Definition 5.1.2 *A rectype is free if every constructor has corresponding destructors.*

More explicitly, suppose we have a k-ary constructor C. Then there are destructors $D_{C,i}$, $i \in [k]$, such that for $a = C(b_1, \ldots, b_k)$ we have $b_i = D_{C,i}(a)$. In somewhat sloppy terms, we have the following theorem.

Theorem 5.1.3 (Recursive Operators) *Let $\mathcal{X}$ be a free rectype. Then we can recursively define operations over $\mathcal{X}$.*

On occasion it is still possible to define operations on non-free rectypes, but then an additional argument is needed to justify the construction. For example, recall our primes-plus-multiplication rectype from above. We could define Θ on atoms as $\Theta(p) = 1$ when $p = 2$, 0 otherwise. On composite numbers $a = b \cdot b'$ we set $\Theta(a) = \Theta(b) + \Theta(b')$. Then Θ is the 2-adic valuation, it returns the power of 2 in the prime decomposition of the given number, regardless of how we have factored a. Again, this definition requires a separate justification, as opposed to free rectypes where no further arguments are needed.

5.2 Well-Orderings and Ordinals

5.2.1 Transfinite Reasoning

Our next step is to elaborate on the idea that a process could take transfinitely many steps. We have seen a first example in the last section where a construction of the Ackermann function taking "infinitely many repetitions of infinitely steps," or, slightly more accurately, as many steps as in $\mathbb{N}$ many copies of $\mathbb{N}$. More generally, in computability theory one naturally encounters constructions that require transfinitely many steps. For example, suppose we wish to determine whether a Turing machine $\mathcal{M}$ halts on some input x, the infamous Halting Problem. A simple solution is to run the machine for at most $\mathbb{N}$-many steps: $\mathcal{M}$ may halt after just finitely many steps, or it may keep going "forever," either way, we will find out. Now change the question to "does the machine halt on all inputs?" We can use our method for every single input, but we need to do this $\mathbb{N}$-many times to cover all possible inputs. We will come back to this idea in the chapter on computability.

As a straw-man, we will first talk about stages of a process and then make this notion more precise by implementing stages as particular sets, so-called ordinals. Taking a closer look at the Ackermann example, we can articulate the central requirements for our stages. We will use Greek letters like σ and τ to denote stages.

- 0 is a stage, when initial arrangements are made.
- There are successor stages $\sigma = \tau + 1$, when we take the next step after stage τ is finished.
- There are limit stages λ, when we collect all the work from prior stages $\tau < \lambda$.

The first few stages after 0 can be conveniently thought of as the natural numbers. In any process that requires only plain infinitely steps, this will be sufficient. If we need to continue after all of the finite stages are finished, we get to the first limit stage, which we will write as ω. Then come more successor stages $\omega + 1$, $\omega + 2$, $\ldots$, $\omega + n$, and so on. The next limit stage will be $\omega + \omega = \omega\,2$. The order in the product seems backward, but we will see in the formal treatment that $2\,\omega = \omega$, so this is actually the right expression. Next comes $\omega\,2 + 1$ and so on and so forth. In this terminology, we would need $\omega\omega = \omega^2$ stages for the Ackermann function.

So far, all we have is pretty symbols and some limited intuition about what we would like to accomplish. Before we move on to formalization, let us consider one more classical example of a transfinite process that goes much further than the Ackermann example.

∗ Cantor-Bendixson

For those of you who are familiar with analysis, here is a perfect example of a transfinite process, introduced by Cantor while exploring the limits of Fourier analysis. His work led him to consider so-called perfect sets of reals: topologically closed sets that have no isolated points. By an isolated point of a closed set $A \subseteq \mathbb{R}$ we mean a point $x \in A$ such that for some $\varepsilon > 0$ we have $A \cap B_\varepsilon(x) = \{x\}$, where $B_\varepsilon(x)$ denotes the ball or radius ε around x: $B_\varepsilon(x) = \{\, z \mid |x - z| < \varepsilon \,\}$. If a point is not isolated it is a limit point. For example, in $\mathbb{Z}$, every point is isolated, in $A = \{0\} \cup \{\, 1/n \mid n > 0 \,\}$ all points except for 0 are isolated. On the other hand, the interval $[0, 1] \subseteq \mathbb{R}$ is perfect.

It seems plausible that a perfect set can be constructed from an arbitrary closed set by systematically removing isolated points. Write

$$X' = \{\, x \in X \mid x \text{ limit point of } X \,\}.$$

This operation is referred to as the Cantor-Bendixson derivative. For example, for the last set A above we have $A' = \{0\}$ and $A'' = \emptyset$. If we replace point $1/n$ in A by the shifted version $1/n + A$, we obtain a set B such that $B' = A$.

Write $X^{(n)}$ for the nth derivative of X. Repeating the scaling/shifting construction just mentioned, it is not hard to produce a set B such that $B^{(n)} = \{b\}$, for any real number b and any $n \geq 0$. Unfortunately, given A, it may well happen that all the $A^{(n)}$ where $n \in \mathbb{N}$ fail to be perfect. How do we continue our pruning process beyond the first infinitely many stages? It is a fair guess that we should consider $\bigcap A_n$ next, so we set

$$A^{(\omega)} = \bigcap_{n<\omega} A^{(n)}$$

The weary reader might suspect at this point that $A^{(\omega)}$ is not necessarily perfect either. So we continue our pruning process with stages $\omega + 1$, $\omega + 2$, ... until, after another infinitely many steps, we get to a stage $\omega + \omega = \omega \cdot 2$, then $\omega \cdot 3$, ω^2, ω^3, ω^ω, ω^{ω^ω}, and so on. This is just wild notation so far, we have not yet explained how to represent these stages as sets, much less how to perform arithmetic on them. At any rate, one can show that for all closed sets $A \subseteq \mathbb{R}$ there is some ordinal α such that $A^{(\alpha)}$ is either perfect or empty. Since only countably many points are removed from A in the construction of $A^{(\alpha)}$, this shows that the Continuum Hypothesis holds for closed sets: they are either countable or have the same cardinality as $\mathbb{R}$.

5.2.2 Ordinals Defined

Our next task is to translate our informal notion of stage into the precise technical concept of ordinal. We need to abstract away from the precise nature of the process in question and focus only on the stages themselves. Considering the close connection between induction and well-foundedness, it is not far fetched to think of stages as forming a well-order: stages clearly are linearly ordered and we would not want to have infinite descending chains. Otherwise, the first stage in that chain could never be reached since it would depend on the second, which depends on the third, and so on. This raises the question of exactly which well-order we should pick for this purpose. Fortunately, there is a theorem in set theory that essentially makes this question moot.

Transitivity

In order to give a formal definition of ordinals, we need one more concept from set theory that is quite important in its own right: transitivity.

Definition 5.2.1 (Transitivity) *A set x is transitive if $z \in y \in x$ implies $z \in x$.*

To see why transitivity is important, imagine we have access to some set A and we want to analyze this set. Access to A means access to its elements $a \in A$, which, in our set universe, are again sets. To understand these, we need access to their elements. In a transitive set, this is a non-issue, the elements of elements also directly belong to A itself. Note that this works recursively, if we have $c \in b \in a \in A$, it follows first that $c \in b \in A$ and then $c \in A$. All the sets we need to contend with are already elements of A.

It is clear from the definition that x is transitive iff $\bigcup x \subseteq x$ iff $x \subseteq \mathfrak{P}(x)$. All the von Neumann numerals N_n are transitive, as is the set of all such numerals. The Zermelo numerals Z_n other than Z_0 fail to be transitive, but $\{Z_0, Z_1, \ldots, Z_n\}$ is transitive.

If a set fails to be transitive, we can construct the least transitive superset. This is very similar to the discussion of transitivity of relations in section 2.3, we are now focusing on the element-of relation.

Definition 5.2.2 (Transitive Set Closure) *Let A be any set. The transitive closure of A is defined to be $\mathsf{TC}(A) = \bigcap\{\, X \supseteq A \mid X \text{ transitive} \,\}$.*

We can give a more explicit definition of transitive closure by recursion along $\in$:

$$\mathsf{TC}(x) = x \cup \bigcup_{z \in x} \mathsf{TC}(z)$$

or, a bit more informally,

$$\mathsf{TC}(x) = \bigcup\{x, \bigcup x, \bigcup\bigcup x, \ldots\}$$

If you prefer a little SetTran program:

```
$U = \emptyset$
$Y = X$
forall $n < \omega$ do
    $U = U \cup Y$
    $Y = \bigcup Y$
    od
return $X$
```

There is also a recursive version, but the simple loop is less complicated.

Mostowski Collapse

The next step is to show that arbitrary well-orders can be modeled by just the element-of relation. To this end, consider some set A with a well-founded relation $\prec$. As always, we write $x^{\downarrow}$ for the set of ancestors of x. We call the order $\prec$ extensional if $x^{\downarrow} = y^{\downarrow}$ already implies $x = y$.

Theorem 5.2.1 (Mostowski Collapse) *Suppose $\prec$ is extensional and well-founded on A. Then $\langle A, \prec \rangle$ is uniquely isomorphic to $\langle B, \in \rangle$ where B is transitive.*

The idea is to construct the isomorphism directly by a form of recursion on sets. For x in A we define

$$f(x) = \{\, f(z) \mid z \prec x \,\}$$

To understand how f works, first consider the $\prec$-minimal elements of A, which must exist because of well-foundedness. If x is minimal, then $x^{\downarrow} = \emptyset$, so by extensionality there is exactly one such element a_0 and $f(a_0) = \emptyset$. Next, let z such that $z^{\downarrow} = \{a_0\}$. Again, there has to be exactly one such element a_1 and $f(a_1) = \{\emptyset\}$. And so on, and so forth.

A real justification for this construction will have to wait till we describe axioms for set theory. Suffice it to say that we can set B to be the range of f, so B is transitive and f translates $\prec$ over A into $\in$ over B. f clearly is a surjection and more work shows that it is also injective; furthermore, since B must be transitive, there is only one way to construct an isomorphism.

Mostowski's theorem motivates the following definition of ordinals as prototypical well-orders.

Definition 5.2.3 (Ordinals) *An ordinal is a set α that is transitive and is well-ordered by the element-of relation $\in$. $On = \{\, \alpha \mid \alpha \text{ ordinal} \,\}$ is the collection of all ordinals.*

We use lower case Greek letters to denote ordinals. Our definition is very concise, but rather opaque; it is far from clear that we really obtain a meaningful representation of our old informal notion of a stage. First notice that the finite von Neumann numerals that we already encountered in section 1.1.4 are ordinals according to this definition, as is ω, the collection of all the finite numerals. It is easy to check that any successor ordinal $\alpha^{+} = \alpha \cup \{\alpha\}$ is an ordinal whenever α is an ordinal. The problem is that we do not yet understand what happens with the other ordinals, the ones that fail to be successors and correspond to limit stages.

We need a few basic propositions that will show that On is well-ordered by $\in$ and that non-successor ordinals are the unions of all the smaller ones.

Lemma 5.2.1

1. *The elements of ordinals are themselves ordinals.*
2. *For any two ordinals α and β, we have $\alpha \subseteq \beta$ or $\beta \subseteq \alpha$.*
3. *$\in$ is a total strict order on On.*
4. *For any set A of ordinals, $\bigcup A$ is again an ordinal.*

We sketch some of the proofs of these claims.

For (1), suppose $x \in \alpha$. Since α is transitive we have $x \subseteq \alpha$. But $\in$ well-orders α and thus also x. For transitivity, let $z \in y \in x$. Since $y \in x \subseteq \alpha$ we have $y \in \alpha$ and hence $y \subseteq \alpha$, so $z \in \alpha$. But $\in$ is a total order on α, so $z \in x$ and x is transitive.

For (2), first assume that $\alpha \nsubseteq \beta$ so $\alpha - \beta \neq \emptyset$ and we can set $\gamma = \min \alpha - \beta$. Let $\gamma' = \alpha \cap \beta$. We claim that $\gamma = \gamma'$.

For $z \in \gamma'$ we have $z \neq \gamma$. But $\gamma \in z \in \beta$ would imply $\gamma \in \beta$, a contradiction. Hence we must have $z \in \gamma$. On the other hand, $z \in \gamma$ implies $z \in \alpha$; since $z \notin \alpha - \beta$ we must have $z \in \gamma'$. In conclusion, $\gamma' \in \alpha$.

An analogous argument shows that $\beta \nsubseteq \alpha$ implies $\gamma' \in \beta$. But if neither inclusion were to hold we would have $\gamma' \in \gamma'$, a contradiction.

□

Since we now know that $\in$ is a well-order on On, we write $\alpha < \beta$ instead of $\alpha \in \beta$. Observe that

$$\alpha = \bigcup \{\, \beta^{+} \mid \beta < \alpha \,\}$$

so, as a matter of principle, we do not have to distinguish between different types of ordinals in inductive definitions. Still, it is often clearer to spell out different cases as follows. Those ordinals λ for which $\lambda = \bigcup \lambda$ are limit ordinals. This definition includes $\lambda = 0$, but it is often more convenient to distinguish three types of ordinals: 0, successors and (proper) limit ordinals, see the example below. Observe that an ordinal $\alpha \in \mathit{On}$ is finite if, and only if, none of its predecessors is a proper limit ordinal, they are all 0 or successors.

One might wonder what can be said about intersections of sets of ordinals. Any non-empty set $A \subseteq \mathit{On}$ always has a least element; in fact, $\bigcap A$ is this particular smallest ordinal.

Well-orders are always comparable in the sense that one of them is isomorphic to an initial segment of the other. Here we think of the whole order as being an initial segment of itself (as opposed to proper initial segments). Suppose $\prec$ well-orders some set A. We can enumerate the elements of A by constructing a partial function $f : \mathit{On} \to A$ defined by

$$\begin{aligned} f(0) &= \min_{\prec}(A) \\ f(\alpha+1) &= \min_{\prec}(A - \{\, f(\nu) \mid \nu \leq \alpha \,\}) \\ f(\lambda) &= \min_{\prec}(A - \{\, f(\nu) \mid \nu < \lambda \,\}) \end{aligned}$$

or, more concisely,

$$f(\alpha) = \min_{\prec}(A - \{\, f(\nu) \mid \nu < \alpha \,\})$$

As we will see in theorem 5.3.9, the domain of f is a proper initial segment of On, so we get an order isomorphism $f : \beta \to A$. The ordinal β is uniquely determined by A and is referred to as the order type or length of the well-order. The order type of a well-order is a successor ordinal iff the order has a largest element, and a limit ordinal otherwise.

So, in a sense all well-orders reduce to initial segments of the ordinals, they are all comparable via their order types. This fact is particularly useful since we can naturally define arithmetic of ordinals—which produces a handy notation system for well-orders.

Transfinite Induction

Since the element-of relation is a well-order on On can perform induction. One little technical problem is that On is not a set but a proper class, we will simply ignore these issues. As usual, we define as subset $X \subseteq \mathit{On}$ to be inductive if $\alpha^{\downarrow} \subseteq X$ implies $\alpha \in X$.

Theorem 5.2.2 *If $X \subseteq \mathit{On}$ is inductive, then $X = \mathit{On}$.*

Likewise, we can define functions on the ordinals by recursion, a result due to von Neumann. Here is the version with an additional set parameter. Recall that $\mathbb{V}$ stands for the collection of all sets.

Theorem 5.2.3 (Ordinal Recursion) *Given a function $F(x, y)$ defined on sets, there is a unique function $f : \mathit{On} \times V \to V$ defined by*

$$f(\alpha, x) = F(f({<}\alpha, x), x).$$

Here $f({<}\alpha, x) = \{\, f(\beta, x) \mid \beta < \alpha \,\}$, we are dealing with course-of-value recursion: all the previous values of f are available in the definition of $f(\alpha, x)$. While there is only one case, in practice, it is still often convenient to distinguish between arguments 0, successor ordinals and limit ordinals. As an example, consider the two well-orders $A = \langle \mathbb{N}^+ \times \mathbb{N}^+, < \rangle$, with the usual lexicographic order, and $B = \langle \mathbb{N}^+, \prec \rangle$, defined as follows:

$$n \prec m \iff \nu_2(n) < \nu_2(b) \lor \nu_2(n) = \nu_2(b) \land n < m.$$

Here $\nu_2(x)$ is the highest power of 2 which divides x. Thus

$$1 \prec 3 \prec 5 \prec \ldots \prec 2 \prec 6 \prec 10 \prec \ldots \prec 4 \prec 12 \prec 20 \prec \ldots$$

By induction, we can define isomorphisms between A and B and their order type ω^2. Hence the two orders are isomorphic, see the exercises.

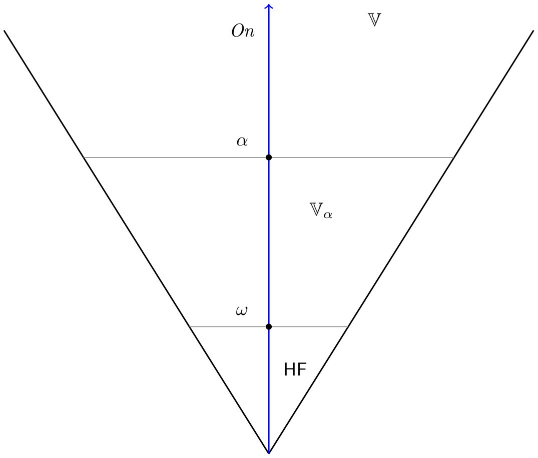
On a slightly more abstract level, we can think of the class of ordinals as being the backbone of the set-theoretic universe $\mathbb{V}$ that is built entirely from the empty set by repeated application of the powerset operation, plus union. We obtain a hierarchical structure that is often referred to as the von Neumann universe. By recursion on ordinals, we define levels $\mathbb{V}_\alpha$ of the universe $\mathbb{V}$ as follows:

$$\begin{aligned} \mathbb{V}_0 &= \emptyset \\ \mathbb{V}_{\alpha^+} &= \mathfrak{P}(\mathbb{V}_\alpha) \\ \mathbb{V}_\lambda &= \bigcup_{\alpha<\lambda} \mathbb{V}_\alpha \\ \mathbb{V} &= \bigcup_{\alpha \in On} \mathbb{V}_\alpha \end{aligned}$$

This hierarchy is cumulative, $\mathbb{V}_\alpha \subseteq \mathbb{V}_\beta$ for all $\alpha < \beta$. $\mathbb{V}_\omega$ is none other than the collection of hereditarily finite sets. Given this hierarchy, one can define the rank of a set S to be the least α such that $S \subseteq \mathbb{V}_\alpha$. We can then think of $\mathbb{V}_\alpha$ as the collection of all sets of rank less than α, which provides a more concise way of writing the recursive definition:

$$\mathbb{V}_\alpha = \bigcup_{\beta<\alpha} \mathfrak{P}(\mathbb{V}_\beta)$$

A rather feeble attempt to visualize this hierarchy follows.



Simply put, the more ordinals, the longer the spine of the universe and the more sets there are. This may sound banal, but questions surrounding the existence of very large ordinals is a major research topic in modern set theory.

Ordinal Arithmetic

There are two ways to go about defining ordinal arithmetic: we can argue about well-orders constructed from smaller ones and their attendant order types, or we can use recursion on ordinals, as just discussed. We start with the first approach since it is more intuitive and leave it to the reader to verify that the formal definitions by recursion produce the right results.

Addition corresponds to concatenating two well-orders by placing one in front of the other. More precisely, suppose A and B are well-orders of order type α and β, respectively. Let

$$C = A \times \{0\} \cup B \times \{1\}$$

be the disjoint union of A and B and order C by reverse lexicographic order: first come all the elements in A, followed by the ones in B. It is easy to see that this produces another well-order, and we think of its order type as the sum of the order types of A and B. Multiplication can be handled similarly, we use the carrier set $C = A \times B$ and reverse lexicographic order.

A direct computational definition can be modeled after the familiar Dedekind method for the naturals, we just have to make sure to handle limit ordinals appropriately

Addition:

$$\begin{aligned} \alpha + 0 &= \alpha \\ \alpha + \beta^+ &= (\alpha + \beta)^+ \\ \alpha + \lambda &= \sup\{\, \alpha + \beta \mid \beta < \lambda \,\} \end{aligned}$$

Multiplication:

$$\begin{aligned} \alpha \cdot 0 &= 0 \\ \alpha \cdot \beta^+ &= \alpha \cdot \beta + \alpha \\ \alpha \cdot \lambda &= \sup\{\, \alpha \cdot \beta \mid \beta < \lambda \,\} \end{aligned}$$

Exponentiation:

$$\begin{aligned} \alpha^0 &= 0^+ \\ \alpha^{\beta^+} &= \alpha^\beta \cdot \alpha \\ \alpha^\lambda &= \sup\{\, \alpha^\beta \mid \beta < \lambda \,\} \end{aligned}$$

One can show that addition and multiplication of ordinals are both associative, but note well that they fail to be commutative. For example, $1 + \omega = \omega \neq \omega + 1$. To see why, draw a little picture of the two well-orders, where the vertical bar indicates the place where the two componenent orders are joined.

$$\bullet \;\mid\; \bullet \;\bullet\;\bullet\;\ldots \;\neq\; \bullet\;\bullet\;\bullet\;\ldots \;\mid\; \bullet$$

Similarly for multiplication we get $2\cdot\omega = \omega \neq \omega\cdot 2 = \omega+\omega$. One can check that distributivity holds according to these definitions. Several other laws of arithmetic also carry over, e.g., $\alpha^\beta \cdot \alpha^\gamma = \alpha^{\beta+\gamma}$ and $(\alpha^\beta)^\gamma = \alpha^{\beta\cdot\gamma}$.

5.2.3 Exercises

Exercise 5.2.1 Given a function $g : X \times V \to V$ and a well-order $<$ on X, explain how to define a function $f : X \to V$ by induction along the lines of von Neumann's approach.

Exercise 5.2.2 Provide a proof for lemma 5.1.2.

Exercise 5.2.3 An ordinal is a set α that is hereditarily transitive: the set itself is transitive and all its elements are transitive.

Exercise 5.2.4 Define as set to be hereditarily finite if it is finite, its subsets are finite, their subsets are finite and so on. More precisely, x is hereditarily finite if all $z \in \mathsf{TC}(\{x\})$ are finite. Which axioms of Zermelo-Fraenkel set theory are modeled by the collection of hereditarily finite sets?

Exercise 5.2.5 Verify some of the claims in lemma 1.1.1 by translating them into propositional logic.

Exercise 5.2.6 What goes wrong if we try to weaken the hypothesis to $(a_1, \ldots, a_n) = (b_1, \ldots, b_m)$. How can one fix the problem?

Exercise 5.2.7 Find an alternative way to implement lists as functions.

Exercise 5.2.8 How many nodes does the membership-tree for the von Neumann ordinal N_n have?

Exercise 5.2.9 Show that it may happen that A_n contains isolated points, for all n.

Exercise 5.2.10 Verify that at least for finite sets this all makes perfect sense: we obtain the intuitive notion of size of a finite set.

Exercise 5.2.11 Prove that for any infinite set A the Cartesian product $A \times A$ and A are equinumerous.

Exercise 5.2.12 Construct a set A so that $A^{(\omega)}$ is perfect, but no earlier level is perfect. Repeat for $A^{(\omega+1)}$.

Exercise 5.2.13 Construct a few bijections $g : (0, 1) \to \mathbb{R}$, using whatever tools from analysis might come in handy. Construct a bijection $g : [0, 1) \to \mathbb{R}$ by hand, without using the Schröder-Bernstein theorem. Likewise, show that $card([0, 1]) = card(\mathbb{R})$ without using the theorem.

Exercise 5.2.14 Prove that the product order on $\mathbb{N}$ is indeed a well-order.

Exercise 5.2.15 Find a different rectype to define hereditarily finite sets.

Exercise 5.2.16 Find a way to code hereditarily finite sets as natural numbers. Your coding function should be computationally simple.

Exercise 5.2.17 Show that addition and multiplication of ordinals are associative.

Exercise 5.2.18 Show that multiplication of ordinals is left-distributive, but not right-distributive.

Exercise 5.2.19 Show that in ordinal arithmetic $\alpha^{\beta+\gamma} = \alpha^\beta \cdot \alpha^\gamma$.

Exercise 5.2.20 Give a justification for ordinal multiplication and exponentiation in terms of constructing well-orders from given ones.

Exercise 5.2.21 Is HF is freely generated? If not, can one adjust the definitions?

Exercise 5.2.22 Suppose $\mathcal{X}$ is a rectype τ, freely generated by X. For any map $\varphi : X \to A$, there is a unique map $\widehat{\varphi} : \mathcal{X} \to A$ that is compatible with τ.

Exercise 5.2.23 Show that $y \subseteq x$ implies $\mathsf{TC}(y) \subseteq \mathsf{TC}(x)$.

Exercise 5.2.24 Show that the recursive definition of the transitive closure operation works as advertised.

Exercise 5.2.25 Show that the von Neumann ordinals are in fact well-ordered by $\in$.

Exercise 5.2.26 Show that every element of a von Neumann ordinal is a von Neumann ordinal using as definition "transitive and $\in$-well-ordered."

Exercise 5.2.27 Show that an ordinal is finite if, and only if, it is well-ordered by $\in$ and by $\ni$.

Exercise 5.2.28 Write a program that translates rationals into surreals.

Exercise 5.2.29 Explain how $\mathbb{R}$ embeds into the surreals. Explain how *On* embeds into the surreals.

Exercise 5.2.30 Explain how to perform multiplication of surreals.

Exercise 5.2.31 Prove the Kuratowski pairing lemma 1.1.3.

Exercise 5.2.32 Discuss the Wiener pairing function $\{\{\{x\}, \emptyset\}, \{\{y\}\}\}$. and the Hausdorff pairing function $\{\{x, 1\}, \{y, 2\}\}$. How about $\{x, \{x, y\}\}$?

Exercise 5.2.33 Find another way to implement pairs as sets.

Exercise 5.2.34 Does the attempt to define pairs via $(x, y) = \{x, \{x, y\}\}$ succeed?

Exercise 5.2.35 Extend the Kuratowski pair to a map $V^{\geq 2} \to V$ by left-associativity. Is it true that $\langle a_1, \dots, a_n \rangle = \langle b_1, \dots, b_m \rangle$ implies $n = m$? Find a proof or counterexample.

Exercise 5.2.36 Explain the relationship between Foundation and the existence of descending $\in$-chains.

Exercise 5.2.37 Explain the relationship between Foundation and the existence of $\in$-minimal elements in the transitive closure of a set.

Exercise 5.2.38 Show that the constructible universe L from above can also be constructed in stages by applying the following operations to $L_\alpha \cup \{L_\alpha\}$: set difference, unordered pair, ordered pair, Cartesian product, taking the support of a binary relation, and performing a permutation of an ordered triple.

Exercise 5.2.39 Show that $(A \times B) \cup (C \cup D) \subseteq (A \cup C) \times (B \cup D)$. How about equality?

Exercise 5.2.40 Assume A, B and C are non-empty. Show that $A \times B = B \times A$ iff $A = B$. Show that $A \times (B \times C) \neq (A \times B) \times C$.

5.3 Cardinality

5.3.1 A Definition of Sorts

We now turn to the question of how to combine sets with notions of size or quantity. It was already recognized by Galileo Galilei that the standard maxim "the whole is larger than any (proper) part" fails in this context, as witnessed by the map $n \mapsto n^2$ over the naturals. The set of squares is arguably much smaller than the set of naturals, yet there is a perfect correlation between the two. It is one of Cantor's major contributions to provide a precise mathematical definition of the size of a set, and to explore the fundamental properties thereof. The size of a finite set is intuitively clear, but, in the infinite case, intuition is not a particularly reliable guide, quite the opposite. For example, it may seem natural to think that all infinite sets are of the same size, in which case there is no need for any machinery to compare them. For the time being, we refer to the size of a set as its cardinality, without attaching any precise definitions yet. In the words of G. Cantor:

> Every aggregate M has a definite "power," which we also call its "cardinal number." ...the general concept which, by means of our active faculty of thought, arises from the aggregate M when we make abstraction of the nature of its various elements m and of the order in which they are given.

Note the reference to "our active faculty of thought," not exactly what one would expect from a modern mathematical definition. For finite sets this idea is hardly contentious: our intuition tells us that $A = \{a_1, a_2, \dots, a_{n-1}, a_n\}$ should have cardinality n (at least if we tacitly assume that $a_i \neq a_j$ for $i \neq j$). The question remains how one should define cardinality more generally, including infinite sets. Cantor starts the discussion by suggesting a method to compare sizes of sets, without getting involved in issues surrounding appropriate generalizations of natural numbers.

> We say that two aggregates M and N are "equivalent" if it is possible to put them, by some law, in such a relation to one another that to every element of each one of them corresponds one and only one element of the other.

In modern parlance: there has to be a bijection between the two sets. Take the stipulation "by some law" with a grain of salt, the bijection need not have a simple description; it just has to exist in some set-theoretic sense. At any rate, here is the corresponding definition.

Definition 5.3.1 (Cardinality) *Two sets A and B have the same cardinality if there exists a bijection between them. Sets with the same cardinality are called equipotent or equinumerous. A is said to have cardinality at most the cardinality of B if there is an injection from A to B. In symbols: $A \approx B$ and $A \preccurlyeq B$.*

This explanation requires no more than the definitions 3.3.1 of injectivity and bijectivity. Note the clever maneuver here: What we really would like to do is to assign a cardinal number $|A|$ to each set A so that all questions about the size of A can be answered in terms of $|A|$. Since we don't quite know how to do this yet, we first define what it means that $|A| = |B|$ and $|A| \leq |B|$. Later we will give an interpretation of $|A|$ as a set, a suitable type of ordinal.

Not well that this approach requires a sanity check: at the very least, "same-cardinality" should be an equivalence relation. This is easy to verify: reflexivity follows from the identity function, symmetry from the fact that the inverse of a bijection is still a bijection, and transitivity is obtained from functional composition. Likewise, "at-most-same-cardinality" is a pre-order (reflexive and transitive). But it is not a partial order, since same cardinality does not imply equality of sets, far from it. Comparability holds, given sufficiently strong axioms of set theory, and in particular the Axiom of Choice. Another problem that we will address shortly is that, given our definition, it by no means clear that

$$A \preccurlyeq B \wedge B \preccurlyeq A \text{ implies } A \approx B.$$

See the discussion of the Cantor-Schröder-Bernstein theorem 5.3.5. We chose to express smaller cardinality in terms of injections, the next lemma shows that surjections work just as well.

Lemma 5.3.1 *For any sets A and B, there is an injection $f : A \to B$ if, and only if, there is a surjection $g : B \to A$.*

Proof. We may safely assume that $A \neq \emptyset$. Assume we have an injection $f : A \to B$. Pick $a_0 \in A$ and set

$$g(b) = \begin{cases} a & \text{if } f(a) = b, \\ a_0 & \text{if } b \notin \mathsf{rng}\, f. \end{cases}$$

It is ease to check that $g : B \to A$ is a surjection.

For the opposite direction assume $g : B \to A$ is a surjection. Then all the fibers $g^{-1}(a)$ are non-empty. Using Choice, we can pick an element b in each and set $f(a) = b$. □

5.3.2 Finite Sets

Finite sets are so important that is well worthwhile to take a closer look at how these definitions work out in the finite case. Recall that $n^{\downarrow} = \{0, 1, \ldots, n{-}1\}$ for any natural number n. Our formal definition of finiteness now looks like this:

Definition 5.3.2 *A set A is finite if there exists some $n \in \mathbb{N}$ a bijection $n^{\downarrow} \to A$. Otherwise, A is infinite.*

In other words, there has to be an enumeration of the form $A = \{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\}$. This sounds eminently plausible, but we still should check that this definition conforms nicely with our intuitions about finite sets. For example, subsets of finite sets should again be finite.

Lemma 5.3.2 *Let $A \subseteq B$ where B is finite. Then A is also finite.*

Proof. By definition, there is a bijection $f : n^{\downarrow} \to B$ where $n \in \mathbb{N}$. Consider the preimage of A under f: $B_0 = \{\, i < n \mid f(i) \in A \,\}$. Define a partial function g on $\mathbb{N}$ by

$$g(k) = \min\bigl(\, i \in B_0 \mid \forall\, j < k\, (g(j) \neq i)\,\bigr).$$

The domain of g is some $m \in \mathbb{N}$ and it is not hard to see that the composition $f \circ g$ is a bijection from $m^{\downarrow}$ to A. □

The recursive definition of g in the last proof is justified by theorem 1.2.2. Finite sets are special when it comes to mappings between them.

Lemma 5.3.3 *Let $f : A \to B$ be a function where A and B are two finite sets of the same cardinality. Then f is injective if, and only if, f is surjective if, and only if, f is bijective.*

Proof. By composing f with the bijections that show that both A and B have cardinality $n \in \mathbb{N}$, we may safely assume that $A = B = n^{\downarrow}$. Moreover, we can assume that n is minimal such that there is an injective function $f : n^{\downarrow} \to n^{\downarrow}$ that fails to be surjective. Note that we must have $n > 0$. Hence the restriction f' of f to $(n{-}1)^{\downarrow}$ is injective. The range of f' differs from the range of f by $b = f(n{-}1)$. Using a recursive definition, we can find a bijection $g : (n-1)^{\downarrow} \to \mathsf{rng}\, f'$; the hard part is to show that the domain of g is indeed $n{-}1$. But then $f'' = g^{-1} \circ f'$ is injective, and, by the minimality of n, also surjective. But then f is also surjective.

The opposite direction is left as an exercise. □

Finiteness is a crucial condition in the last lemma, there are lots of functions $\mathbb{N} \to \mathbb{N}$ that are either injective but not surjective, or surjective but not injective. More surprising is the fact that one could even use this property to define infinity:

Lemma 5.3.4 *A set A is infinite if, and only if, there is an injective function $f : \mathbb{N} \to A$.*

Proof. Assume A is infinite and let $\prec$ be a well-ordering of A. By induction we can construct finite injective maps $f_n : n^{\downarrow} \to A$ where f_n extends f_m for all $n > m$. The construction works for all n since we would obtain a bijection $n^{\downarrow} \leftrightarrow A$ otherwise. But then $f = \bigcup f_n$ works as required.

In the opposite direction, suppose $g : n^{\downarrow} \to A$ is a bijection. Then $g \circ f$ is an injection from $\mathbb{N}$ to $n^{\downarrow}$, a contradiction. □

Dedekind's Ketten

So far, we only have a negative characterization of infinite sets: they fail to be finite. Dedekind realized that one can use the idea of generating the natural numbers from 0 and a successor operation to give a more constructive description of infinite sets. Given a function f on a set A, and a point $a \in A$, the corresponding chain (Kette) is defined to be

$$\bigcap \{\, X \subseteq A \mid a \in X, f(X) \subseteq X \,\}$$

Thus, the chain is the least set that contains a and is closed under f. Clearly, this definition of a chain does not require the natural numbers; instead, it can be used to define them. In Dedekind's view, this means that arithmetic can be reduced to logic. At any rate, it is easy to force a chain to be infinite, we just have to make sure that f is injective and that a is not in the range of f.

Definition 5.3.3 *A set A is Dedekind-infinite if there is an injective function $f : A \to A$ whose range is a proper subset of A.*

In other words, a Dedekind-infinite set has the same cardinality as a proper subset of itself. This is similar to but different from the traditional definition of infinity. One advantage of Dedekind's definition is that it makes no reference to $\mathbb{N}$, so there is no need to construct the naturals first. Given the Axiom of Choice, the two definitions are equivalent.

Theorem 5.3.1 (Dedekind) *A set is infinite if, and only if, it is Dedekind-infinite.*

Proof. Let A be Dedekind-infinite and f the corresponding function. The proof is based on finding a subset of A that is equinumerous with $\mathbb{N}$. To this end, note that $\mathsf{rng}\, f^n - \mathsf{rng}\, f^{n+1} \neq \emptyset$ for all $n \geq 0$. For example, suppose $\mathsf{rng}\, f = \mathsf{rng}\, f^2$. Then, for any $x \in A$, there is a $x' \in A$ such that $f(x) = f(f(x'))$. But then by injectivity $x = f(x')$, contradicting the assumption that $\mathsf{rng}\, f \subsetneq A$. By (AC) we can select $a_n \in \mathsf{rng}\, f^n - \mathsf{rng}\, f^{n+1}$. Then $\mathbb{N}$ and $\{ a_n \mid n \geq 0 \}$ are equinumerous.

The opposite direction is easy. □

5.3.3 Countable Sets

Finite sets are arguably the most important type of set in computer science, closely followed by infinite sets that have the same cardinality as the natural numbers.

Definition 5.3.4 (Countability) *A set A is denumerable if there is a bijection $f : \mathbb{N} \to A$. A set A is countable if it is denumerable or finite, and uncountable otherwise.*

Thus, a non-empty set is countable iff there is a surjection from $\mathbb{N}$ to the set. Informally, a set is denumerable if it can be listed just like $\mathbb{N}$:

$$a_0, a_1, a_2, \ldots, a_n, a_{n+1}, \ldots$$

Note that we assume this enumeration to be repetition-free, each element of A appears exactly once, though the exact order is irrelevant. It is a good exercise to modify an enumeration with repetitions to one that is repetition-free, assuming the set is infinite. If the elements a_i are in addition ordered (assuming there is some total order on A), then we refer to the map $\mathbb{N} \to A$ as the principal function for A, aka Hauptfunktion. The same ideas apply to finite sets, we just have to replace $\mathbb{N}$ by $n^{\downarrow}$.

A word of warning: from the perspective of set theory, there is little difference between an arbitrary enumeration and the principal function, we can just modify the former to obtain the latter. But if one takes into account additional properties of these maps, things become more complicated: there are cases where the principal function fails to be computable, yet A is denumerable via a computable function.

It is natural to ask about the size of various numerical sets such as $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$. The integers are easy to deal with, we can simply alternate positive and negative integers to show that the whole set is denumerable:

$$0, -1, 1, -2, 2, -3, 3, \ldots, -n, n, \ldots$$

Somewhat less informally, we can define a bijection $f : \mathbb{N} \to \mathbb{Z}$ by $f(2n) = n$ and $f(2n+1) = -(n+1)$.

For rational numbers the situation is less clear. By using the same trick as for $\mathbb{Z}$, we may safely consider only non-negative fractions of the form $n/(m+1)$ where $n, m \in \mathbb{N}$. Since both n and m range over the natural numbers, one might think that there are "infinitely times infinitely" many fractions, more than there are natural numbers. More formally, we have to determine the cardinality of $\mathbb{N} \times \mathbb{N}$. Using the enumeration method this is rather intuitive:

$$(0,0)(0,1)(1,0)(0,2)(1,1)(2,0)(0,3)(1,2)(2,1)(3,0)(4,0)\ldots$$

In the computation part of these notes we will show in great detail that there are computationally simple bijection between $\mathbb{N}$ and $\mathbb{N}^2$, see section 8.1.3.

Theorem 5.3.2 (Cantor) *$\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$ have the same cardinality.*

If we allow enumerations that contain repetitions, we also obtain all non-empty finite sets this way. Being enumerable with repetitions is a property that is inherited by subsets.

Lemma 5.3.5 *Let $f : \mathbb{N} \to A$ be a surjection and $B \subseteq A$. Then B is countable.*

Proof. Here is a convoluted recursive definition:

$$g(n) = f\left(\min\bigl(j \in \mathbb{N} \mid f(j) \in B \land \forall\, i\ < n(f(j) \neq g(i))\bigr)\right)$$

If B is finite, the domain of g is an initial segment of $\mathbb{N}$, otherwise it is all of $\mathbb{N}$. It is easy to check that g is a bijection. □

For $A = B$ this means: if there is a surjection $\mathbb{N} \to A$, there already is a bijection: Just redefine the function so it does not hit the same element in A twice.

Theorem 5.3.3 *(AC) A countably union of countable sets is countable.*

Proof. Suppose $(A_i)_{i\in\mathbb{N}}$ is a countable family of countable sets and let $A = \bigcup A_i$. By Choice, we can fix a surjection $f_k : \mathbb{N} \to A_k$ for all k. It follows that $funcF\mathbb{N} \times \mathbb{N}A$, $F(k, x) = f_k(x)$ is a surjection. Done the Cantor's theorem.

□

Words are Countable

As is shown in section 8.1.3 there is a computationally simple bijection $\mathbb{N}^\star \leftrightarrow \mathbb{N}$. By the last lemma, the collection of all sequences of natural numbers $a_1, a_2, \ldots, a_n$ such that $0 \leq a_i < a$ for some fixed bound a is also countable. Hence the collection of all words over a finite alphabet is countable.

A more constructive way to establish this result is to arrange the words into a sequence by sorting them in length-lex order, see section 2.2.1. E.g., for alphabet $\{a, b, c\}$ we get:

$$\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \ldots$$

Hence we have the following theorem:

Theorem 5.3.4 *There are only countably many words over a finite alphabet.*

It is important to note that the standard lexicographic order does not work here (unless the alphabet contains only a single letter): there is an infinite descending chain

$$b > ab > aab > aaab > \ldots > a^n b > a^{n+1} b > \ldots$$

so we do not get a sequence of order type $\mathbb{N}$ from the lexicographic order.

Countability of words provides a proof for the fact that only countably many functions can be computable. To see this, note that any conceivable computable function could be implemented in a standard programming language, say, C. But every C program is just a special string of ASCII characters. Since there are only countably many ASCII strings, there are only countably many programs by the lemma, and hence only countably many computable functions.

5.3.4 Three Theorems

Recall that we can compare the cardinality of two sets without having to define cardinal numbers: we can use injective and bijective functions for the comparison. The definition really requires a sanity check, we need to make sure that $X \preccurlyeq Y$ and $Y \preccurlyeq X$ implies $X \approx Y$, in keeping with our intuition about comparing sizes.

Theorem 5.3.5 (Schröder-Bernstein) *Suppose $f : X \to Y$ and $g : Y \to X$ are injective. Then X and Y have the same cardinality.*

Proof. It suffices to establish the following, apparently weaker claim, already known to Cantor to be equivalent to the full result.

Claim: Suppose $C \subseteq B \subseteq A$ and $A \approx C$. Then $A \approx B$.

To see this, set $A = X$, $B = g(Y)$ and $C = g(f(X))$. Then certainly $X \approx A \approx C$, and $B \approx Y$. From the claim it follows that $A \approx B$, and, by transitivity, $X \approx Y$.

To establish the claim, suppose we have a bijection $h : A \longleftrightarrow C$. We need to concoct a new bijection $H : A \longleftrightarrow B$. To this end, let $D = \{\, h^n(x) \mid x \in A - B, n \geq 0 \,\}$. In other words, D is the union of all orbits of points in $A - B$ when we think of them as sets rather than sequences. Define a function H on A as follows:

$$H(x) = \begin{cases} h(x) & \text{if } x \in D, \\ x & \text{otherwise.} \end{cases}$$

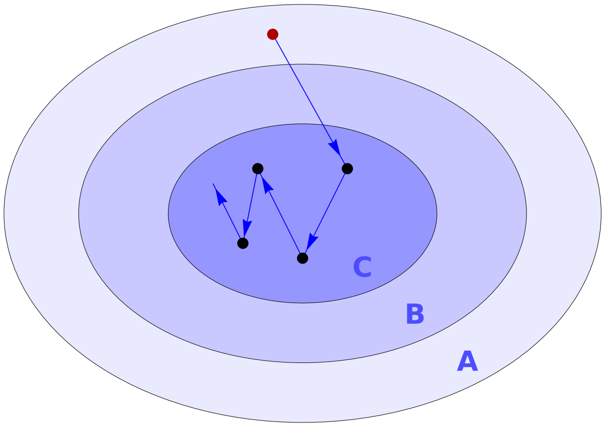
Note that $A - B \subseteq D$, so the range of H is certainly contained in B. So consider an arbitrary $x \in B$. If $x \in D$, then for some z we have $h(z) = x$, so x is in the range of H.

For injectivity suppose $H(x_1) = H(x_2)$. If both x_1 and x_2 are in D, or both are not in D, it follows that $x_1 = x_2$. So suppose $x_1 \in D$ but $x_2 \notin D$. Then $H(x_1) \in D$, but $H(x_2) = x_2 \notin D$, contradiction.

□

There are lots of little details in this proof that should be checked carefully, it is far from obvious that this construction really works. Cantor himself conjectured this result, but was unable to provide a proof. Dedekind, in typical form, had a more elegant proof than Schröder and Bernstein, and about 10 years earlier. For another proof, due to J. Kňig, see the exercises.

Intuitively, one can motivate the construction as follows. Think of the elements of A as places, and put a pebble on each place. We have to move the pebbles in such a way that they all wind up on places in B. Moreover, each place in B must be occupied by exactly one pebble after all the moving around has finished. If $A = B$ this is easy: do nothing. Otherwise, we have to move all pebbles on places $x \in A - B$. The only hope is to move this pebble x to some place $h(x)$, displacing the pebble there, which displaces another, and so on. The argument above shows that this strategy works.



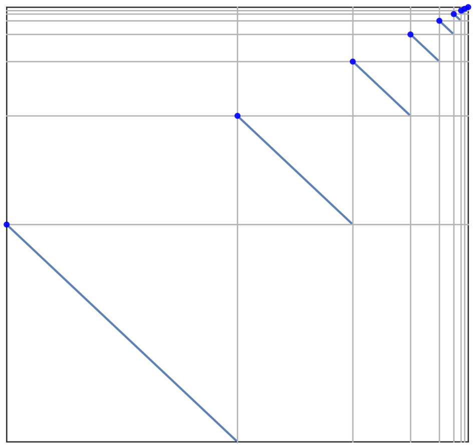
Theorem 5.3.5 does not just legitimize our indirect definition of cardinality, it also helps to pin down the cardinality of various sets. For example, consider the problem of showing that the open interval $(0,1) \subseteq \mathbb{R}$ has the same cardinality as all of $\mathbb{R}$. We can establish a bijection explicitly by

$$f : (0,1) \to \mathbb{R} \qquad f(x) = \tan \pi(x - 1/2).$$

Now change the problem by one point: show that the half-open interval $[0,1) \subseteq \mathbb{R}$ and $\mathbb{R}$ are equinumerous. Using Schröder-Bernstein this is quite straightforward by establishing two injections:

$$f^{-1} : \mathbb{R} \to (0,1) \subseteq [0,1) \qquad \mathsf{id} : [0,1) \to \mathbb{R}$$

It is often much harder to establish an explicit bijection. Here is one between $[0,1]$ and $(0,1]$.



Diagonalization and Uncountability

Here are two important results by Cantor that show that the world of cardinalities is much more complicated than what one might expect. The first theorem shows that there are at least two levels of infinity, and that they play a rôle in calculus: the reals are uncountable. The second theorem shows that in fact there are infinitely many levels of infinity

$$|\mathbb{N}| < |\mathfrak{P}(\mathbb{N})| < |\mathfrak{P}^2(\mathbb{N})| < \ldots < |\mathfrak{P}^{42}(\mathbb{N})| < \ldots$$

associated with the powerset construction. The ellipsis may suggest that the ordertype of these cardinalities is just ω, but reality is far worse: there are as many cardinals as there are ordinals. Luckily, only a few of them appear to be of major relevance (unless you are working in set theory, of course).

Theorem 5.3.6 (Cantor) *The set of real numbers is uncountable.*

From a computability point of view, uncountability of the reals is a major obstacle: the structure of the reals is by necessity non-computable, placing natural restrictions on our ability to compute with real numbers. Note that in the discussion of methods such as the simplex algorithm for linear programming it is often convenient to pretend that the computation takes place over the reals, and then deal with issues of round off errors and the like later. Also note that the theorem leaves open the possibility that higher-dimensional Euclidean spaces like $\mathbb{R}^2$ might be of even higher cardinality. It took Cantor some time to realize that all these spaces have the same cardinality as $\mathbb{R}$, a result that clashes with one's geometric intuition about functions between, say, $[0,1]$ and $[0,1]^2$, see section 5.3.5.

Surprisingly, the method Cantor introduced to establish his uncountability result is arguably even more important than the theorem itself: it is known as diagonalization and is a key technique in computability theory and complexity theory, see chapter 8. Here is an easy warm-up exercise: show that the number of binary sequences of length n is larger than n. This claim can be established directly by counting, but ordinary counting does not work for infinite sets; we need a different approach. Instead, let us assume there are only n many binary sequences $s_1, \ldots, s_n$ of length n. We can now construct a new binary sequence t, again of length n, by setting

$$t(i) = 1 - s_i(i).$$

Then t differs from all the $s_1, \ldots, s_n$ in at least one bit. Hence $t \neq s_i$ for all $i = 1, \ldots, n$ and we can conclude that there must be more than n such sequences. We can think of this argument as flipping each bit along the diagonal of a square table. The resulting sequence cannot be a row in the table.

$$\begin{array}{ccccc}
s_0(0) & s_0(1) & s_0(2) & \ldots & s_0(n-1) \\
s_1(0) & s_1(1) & s_1(2) & \ldots & s_1(n-1) \\
s_2(0) & s_2(1) & s_2(2) & \ldots & s_2(n-1) \\
 & \vdots & & & \vdots \\
s_{n-1}(0) & s_{n-1}(1) & s_{n-1}(2) & \ldots & s_{n-1}(n-1)
\end{array}$$

It is really quite surprising that we can construct a sequence not in the table without even looking at all the entries. In fact, we only consider a fraction of $1/n$ of the table elements. At any rate, this approach is clearly overblown for finite binary sequences, but it has the great advantage of generalizing immediately to infinite sequences. Recall that $\mathbf{2} = \{0, 1\} \subseteq \mathbb{N}$.

Theorem 5.3.7 (Cantor) *There are uncountably many infinite binary sequences: the set* $\mathbb{N} \to \mathbf{2}$ *of all infinite binary sequences is uncountable.*

Proof. Suppose for the sake of a contradiction that $(s_i)_{i \in \mathbb{N}}$ is a enumeration of all sequences $\mathbb{N} \to \mathbf{2}$. Define a new binary sequence $t : \mathbb{N} \to \mathbf{2}$ by setting

$$t(i) = 1 - s_i(i).$$

Clearly $t \neq s_i$ for all i, contradiction. □

It follows that $\mathfrak{P}(\mathbb{N})$ is also uncountable: we can think of a subset of $\mathbb{N}$ as a map $f : \mathbb{N} \to \mathbf{2}$: this is just the standard representation of a set as a characteristic function.

Proof. (of theorem 5.3.6) As an easy corollary to the last theorem, the set S of all infinite sequences of numbers 2 and 7 is also uncountable. Now consider the evaluation map

$$f(x) = \sum_{i \geq 0} x_i 10^{-i-1}$$

where $x \in \{2,7\}^\omega$. The map takes values in $\mathbb{R}$, and in fact into the interval $[0, 7/9]$. But f is injective, so $\mathbb{R}$ must be uncountable. □

Our choice of digits may seem strange, more natural would be to use all decimal digits $\{0, 1, \ldots, 9\}$ (the last theorem easily generalizes from binary to all infinite sequences over finite sets). But then there is a problem with the evaluation map: $f(1000\ldots) = f(0999\ldots)$ and the map is no longer injective. One can fix this problem by analyzing the sequences that cause difficulties, but the preceding argument is easier.

As already mentioned, Euclidean spaces $\mathbb{R}^d$ all have the same cardinality as $\mathbb{R}$. So the question arises whether there are even larger sets.

Theorem 5.3.8 (Cantor) *For any set A, the cardinality of the powerset $\mathfrak{P}(A)$ is greater than the cardinality of A.*

Proof. We need to show that the cardinality of $\mathfrak{P}(A)$ is strictly greater than the cardinality of A. The map $a \mapsto \{a\}$ is an injection from A to $\mathfrak{P}(A)$, so $A \preccurlyeq \mathfrak{P}(A)$. To show that the cardinality of the powerset is strictly larger, assume that there is a surjection $f : A \to \mathfrak{P}(A)$. Think of a as a "name" for $f(a)$. Define a set

$$B = \{\, a \in A \mid a \notin f(a) \,\} \subseteq A.$$

Since f is surjective, we must have $B = f(b)$ for some $b \in A$. But then $b \in B$ implies $b \notin B$, and conversely; contradiction. □

Cantor's observation that there are many different cardinalities provides an opportunity to establish highly non-constructive existence results. Suppose we have two sets $A \subseteq B$, and we would like to show that there is some $x \in A - B$. To this end, it suffices to show that $|A| < |B|$. For example, one can show this way that there is a non-computable function $\mathbb{N} \to \mathbb{N}$ without exhibiting such a function. These arguments, while absolutely correct, tend to be somewhat irritating: we would much prefer to have an explicit example in hand. For instance, we want a concrete example of a function $f : \mathbb{N} \to \mathbb{N}$ that has a clear definition but is provably not computable. We will show how to handle this in chapter 8.

Comments

Note that Cantor's construction is very similar to Russell's paradox, it is surprising that Cantor never made the transition.

Russell's set:	$R = \{\, x \mid x \notin x \,\}$
Cantor's set:	$B = \{\, a \in A \mid a \notin f(a) \,\}$

The existence of R is contradictory and ruined Frege's axioms. But there is nothing paradoxical about B, it just shows that f cannot be surjective.

There is another, astonishing fact relating to diagonalization: it is nowadays a key technique in computability theory and complexity theory. For example, the undecidability of the infamous Halting Problem is established by diagonalization, see the chapter 8. This is quite amazing since diagonalization appears to have absolutely nothing to do with computability. In fact, when diagonalization was introduced by Cantor in 1891, no concept of computability even existed; it would take another 40 years for that development.

5.3.5 Cardinals

Let's return to the issue of measuring the sizes of sets. We have a solution of sorts based on the existence of injections from one set to another, but that is really a bit of a cop out. Following Frege's maxim that every concept should have an associated extension, we would like to assign an actual number object to a set that measures size. This requires a concrete notion of a cardinal number. Considering the natural numbers, it is a fair guess is that cardinals should be ordinals of sorts. Of course, ordinals in general do not work in this capacity, for example, there are many ways we can well-order $\mathbb{N}$ producing a variety of order types, yet the cardinality of all these orders is the same as the cardinality of $\mathbb{N}$. To get just some impression of how complicated these countable ordinals can be, note that we can certainly reach $\omega + \omega$, $\omega \cdot \omega$ and ω^ω. Here is one more example of a countable ordinal that may seem absurdly large. Call an ordinal α an ε-number if $\alpha = \omega^\alpha$. Thus, an ε-number is a fixed point of the map $\beta \mapsto \omega^\beta$. At first glance, it may not be clear that ε-numbers exist at all, it surely is quite difficult to visualize the well-orders that would be associated with these ordinals. Still, we can construct an ε-number by a limit process: let $\alpha_0 = 1$ and set $\alpha_{n+1} = \omega^{\alpha_n}$. Then the limit of $\alpha = \bigcup \alpha_n$ is the least ε-number and commonly referred to as ε_0 (epsilon naught). Informally,

$$\varepsilon_0 = \omega^{\omega^{\omega^{\cdot^{\cdot^{\cdot}}}}}$$

ε_0 is countable and there is a close connection between ε_0 and induction in Peano arithmetic, see section 7.6.4. As a matter of fact, ε_0 is actually quite tiny as far as countable ordinals go. Really.

Returning to our old function-based approach to cardinality, we might try the following definition.

Definition 5.3.5 (Cardinals) *An ordinal κ is a cardinal if there is no injection $\kappa \to \alpha$ for any $\alpha < \kappa$. The cardinality of a set A is the uniquely determined cardinal κ such that there is a bijection $\kappa \leftrightarrow A$. In symbols: $\kappa = |A|$.*

By the venerated Pigeon Hole Principle, all finite ordinals, as is ω, the first infinite cardinal. So we can attach a number to finite and countably infinite sets. Are there any other cardinals, though? This question was resolved over a century ago.

Theorem 5.3.9 (Hartog 1905) *For any set A there is an ordinal α such that there is no injection $\alpha \to A$.*

Proof. Any injection $\alpha \to A$ determines a well-order on some subset of A. Let W be the set of order types of all these well-orders. Since W is well-ordered by $\in$, it has some order type β. But then there cannot be an injection $\beta \to A$. □

In particular, when $A = \mathbb{N}$, there is some ordinal α that has no injection to $\mathbb{N}$. In other words, there is an uncountable ordinal. But then there also is a least uncountable ordinal, another one larger than that, and so on. Informally, the cardinalities of the sequence of ordinals tend to be constant over long stretches, and then jump when the length of a well-order reaches the next cardinal. Following Cantor, it is customary to use different symbols for cardinals, even if we already have a symbol for the corresponding ordinal. In particular ω, in its capacity as a cardinal, is written

$$\aleph_0 \qquad \text{aleph naught}$$

so that $\aleph_0 = |\omega| = |\omega^2| = |\omega^\omega| = |\varepsilon_0|$. So now we have three notations for the natural numbers: $\mathbb{N}$ in the context of arithmetic, ω as an ordinal, and $\aleph_0$ as a cardinal. This distinction is actually quite useful since it emphasizes different concepts that, it so happens, are all represented by the same set. And, this is just the start, we actually have an ever increasing sequence

$$\aleph_0, \aleph_1, \ldots \aleph_\omega, \ldots \aleph_{\omega+\omega}, \ldots \aleph_{\omega^2}, \ldots \aleph_{\varepsilon_0}, \ldots, \aleph_{\aleph_1}, \ldots$$

If you find this vertigo-inducing, you are not alone. This never-ending stream of alephs is important in axiomatic set theory, but only a few of these cardinals are prominent outside of set theory. The second infinite cardinal, $\aleph_1$, is the least uncountable ordinal. One can show that

$$\aleph_1 \leq |\mathbb{R}|$$

but equality, Cantor's famous Continuum Hypothesis, cannot be settled in the framework of standard set theory. So it is safe to assume that $\aleph_1 = |\mathbb{R}|$ or that $\aleph_1 < |\mathbb{R}|$. Assuming the Continuum Hypothesis would seem to make the analysis of subsets of the reals a bit easier, since there are just three cases to consider: finite, countably infinite, or the same cardinality as $\mathbb{R}$. Unlike with the Axiom of Choice, though, neither option seems to be particularly natural or consequential, so the Continuum Hypothesis has not been adopted as a standard axiom.

Cantor referred to the natural numbers as the first number class, and to countable ordinals as the second number class. The second number class is particularly important in the theory of computation. One interesting property of this class is that all its elements can be approximated by sequences of length ω.

Definition 5.3.6 *Define the cofinality of a limit ordinal α to be the length of a shortest sequence that is cofinal in $\{\, \beta \in On \mid \beta < \alpha \,\}$. Ordinal α is regular if it has cofinality α.*

A set $A \subseteq \alpha$ is cofinal in α if $\bigcup A = \alpha$. Likewise, a sequence $(\beta_n)_{n<\lambda}$ is cofinal in α if $\{\, \beta_n \mid n < \omega \,\}$ is so cofinal. This is particularly interesting when the sequence is strictly increasing, we are approximating α from below; We are trying to minimize λ. Verify that $\omega + \omega$, ω^2, ε_0 and the like all have cofinality ω. Beware, there are much larger cardinals whose cofinality is again ω, for example $\aleph_\omega$ works. In order to escape from cofinality ω, we have to increase the cardinality.

Lemma 5.3.6 *All limit ordinals in the second number class have cofinality ω. But the cardinal $\aleph_1$ is regular.*

Cardinal Arithmetic

Recall that the motivation for ordinal arithmetic is the construction of compound well-orders constructed from smaller ones. An analogous approach can be used to justify the arithmetic of cardinals. For instance, for addition we again consider the disjoint union of two sets:

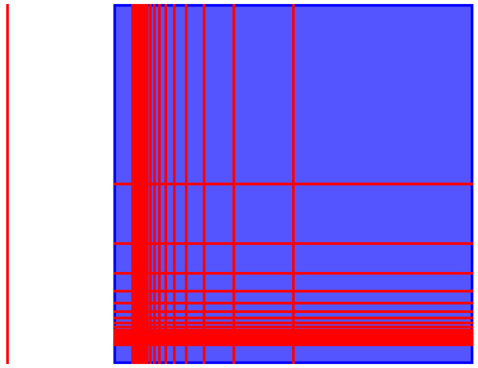
$$C = A \times \{0\} \cup B \times \{1\}$$

The cardinality of C is the sum of the cardinalities of A and B. In the same way, the cardinality of $A \times B$ is their product. From our prior experience in constructing bijections, we conclude that $\aleph_0 + \aleph_0 = \aleph_0 \cdot \aleph_0 = \aleph_0$. We won't give a detailed definition of the arithmetic operations here. One can show that addition and multiplication of cardinals are associative and commutative operations. The important point to remember is that arithmetic of cardinals is different from ordinal arithmetic. Some aspects of it appear much simpler.

Lemma 5.3.7 *Let λ and κ be two non-zero cardinals, at least one of them infinite. Then*

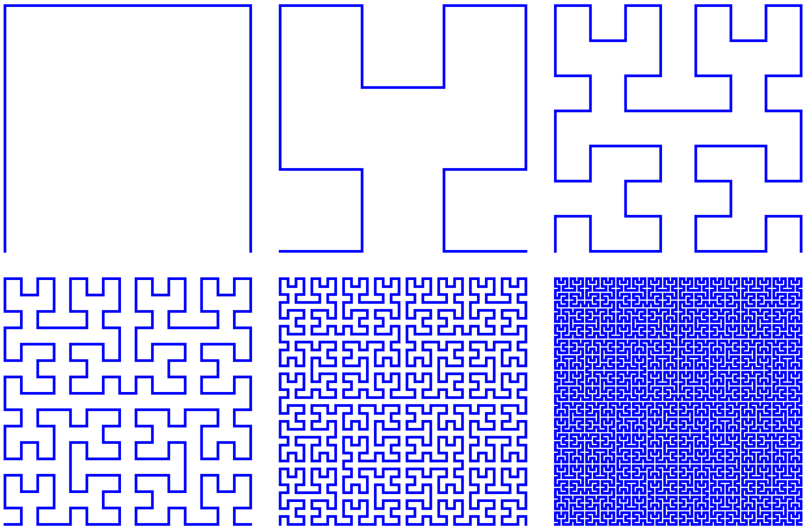
$$\lambda + \kappa = \lambda \cdot \kappa = \max(\lambda, \kappa).$$

Note that the lemma implies in particular that any infinite set A is equinumerous with $A \times A$. This fact may seem rather unimpressive, but when applied to analysis it met with fierce resistance in Cantor's time. For example, the unit interval $[0, 1] \subseteq \mathbb{R}$ has the same cardinality as the unit square $[0, 1]^2 \subseteq \mathbb{R}^2$.



This is rather counter-intuitive; the unit interval can be embedded into the unit square in many ways, uncountably many in fact. Indeed, there are no "nice" bijections[1]. On the other hand, they don't have to be pathological either. One possible approach to defining such a bijection is to design a sequence of curves that fill the whole unit square in the limit. The one in figure **??** is due to Hilbert and is continuous.

[1] Cantor himself wrote in a letter to Dedekind: Je le vois, mais je ne le crois pas (I see it, but I can't believe it).



Strictly speaking, the curve is obtained in as a limiting process and requires a bit of argument to show that one really obtains a bijection $[0,1] \leftrightarrow [0,1]^2$. Hilbert's 1881 paper is only two pages long, tough.

Alas, when exponentiation enters the picture, cardinal arithmetic becomes rather complicated. Cantor's diagonal argument shows that $2^\kappa > \kappa$ for all cardinals κ. But since $|\mathbb{R}| = |\mathfrak{P}(\mathbb{N})| = 2^{\aleph_0}$ we cannot even say if $2^{\aleph_0} > \aleph_1$, a truly astonishing failure. A few other basic inequalities between cardinals are given in the next lemma.

Lemma 5.3.8 *Let $\lambda_1 \leq \kappa_1$ and $\lambda_2 \leq \kappa_2$ be cardinals. Then*

$$\lambda_1 + \lambda_2 \leq \kappa_1 + \kappa_2 \text{ and } \lambda_1 \cdot \lambda_2 \leq \kappa_1 \cdot \kappa_2$$

Unless $\lambda_1 = \lambda_2 = \kappa_1 = 0 < \kappa_2$ we also have

$$\lambda_1^{\lambda_2} \leq \kappa_1^{\kappa_2}.$$

There are also infinitary version of the arithmetic operations on cardinals. Here is one famous result.

Theorem 5.3.10 (Köenig's Lemma, AC) *Let I be an arbitrary index set and assume cardinals $\lambda_i < \kappa_i$ for all $i \in I$. Then*

$$\sum_i \lambda_i < \prod_i \kappa_i.$$

The important feature of this result is the strict inequality. The result is quite obvious as long as all the cardinals are natural numbers, but it holds in general. The result requires the Axiom of Choice and is in fact equivalent to it. König's theorem has the following consequence with respect to cofinality.

Lemma 5.3.9 *Let $\aleph_0 \leq \kappa$ and $2 \leq \lambda$. Then*

1. $\kappa < \kappa^{\operatorname{cof}\kappa}$
2. $\kappa < \operatorname{cof} \lambda^\kappa$

Proof. Let $\alpha_\nu < \kappa$, $\nu < \mathrm{cof}\kappa$, be a sequence of ordinals with limit κ. By Köenig

$$\kappa = \sum \alpha_\nu < \prod \kappa = \kappa^{\mathrm{cof}\kappa}$$

For the second part, set $\mu = \lambda^\kappa$ and assume $\mathrm{cof}\mu \leq \kappa$. Then by part (1)

$$\mu < \mu^{\mathrm{cof}\mu} \leq \mu^\kappa = (\lambda^\kappa)^\kappa = \lambda^\kappa = \mu$$

contradiction. □

One of the reason the last lemma is interesting is that it shows $\aleph_\omega \neq 2^{\aleph_0}$. Solovay has proven that this constraint (and analogous ones obtained in a similar manner) are the only obstructions to choosing a value for $\mathfrak{c}$, anything else can be realized in suitable models for ZFC.

5.3.6 Exercises

Exercise 5.3.1 Prove the missing direction of lemma 5.3.3.

Exercise 5.3.2 Consider the positive naturals $\mathbb{N}_+ = \{1, 2, 3, \ldots\}$. Assume the weak associativity property $x+(y+1) = (x+y)+1$. Use induction to show that $x+(y+a) = (x+y)+a$ for all a.

Exercise 5.3.3 Show that a countable union of countable sets is countable.

Exercise 5.3.4 Consider two injections $f : A \to B$ and $g : B \to A$ and define the map $\phi : \mathfrak{P}(A) \to \mathfrak{P}(A)$ by

$$\phi(X) = A - g(B - f(X)).$$

Show that ϕ is monotone, and give a proof of Schroeder-Bernstein using the least fixed point of ϕ.

Exercise 5.3.5 Assume A and B are uncountable. Show that $A \cup B$ is again uncountable.

Exercise 5.3.6 Show that the set of irrational numbers is uncountable. Show that the set of transcendental numbers is uncountable.

Exercise 5.3.7 Give a proof of the uncountability of $\mathbb{R}$ using infinite sequences of all decimal digits $\{0, 1, \ldots, 9\}$.

Six

Axiomatic Set Theory

So far, our approach to set theory has been intuitive, based on just two fairly plausible assumptions: Extensionality and Comprehension. The gentle reader will have noticed occasional ominous remarks about the need to be careful in the application of Comprehension. For example, the universe of all sets, $\mathbb{V} = \{\, x \mid x = x \,\}$ cannot itself be a set. In this section we will explain in greater detail why problems arise in the first place and how to address them, by axiomatizing set theory and reasoning from the axioms, rather than just intuition—however tempting the latter may be.

6.1 Axiomatic Set Theory

6.1.1 Frege's Axioms

Extensionality and Comprehension are the backbone of work by Gottlob Frege in the late 1800s[1]. Frege's system was really a system of higher-order logic, but we can interpret his ideas nicely in set theory. Frege assumes that every concept P is associated with an "extension," essentially the set of all objects with property P. This idea leads to our two basic principles:

- Extensionality

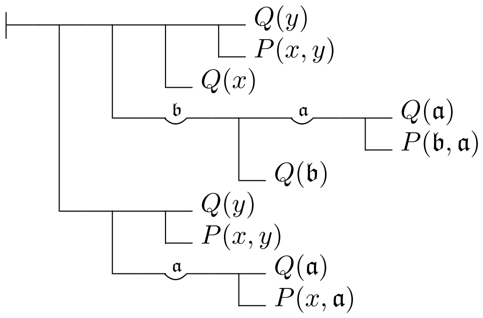
$$\forall\, z\, (z \in x \iff z \in y) \Rightarrow x = y$$

- Comprehension
  For any property $P(z)$:

$$\exists\, x\, \forall\, z\, \big(z \in x \iff P(z)\big)$$

For modern standards, the notion of a property is still a bit vague, we can make it more precise by fixing a particular system of formal logic such as first-order logic, and then replacing $P(z)$ by an actual formula with free variable z. In this setting, there is not a single comprehension axiom but a whole family of them, one for each choice of the formula (a so-called axiom schema). Frege was also adamant about specifying the logical structure of his system in great detail, a topic we will return to in the section on logic. Incidentally, Frege also invented a two-dimensianal notation system for the formulae in his system.

[1] G. Cantor, the driving force behind the development of set theory, was never much interested in formalization or axiomatization.



Considering the state of typesetting technology in the late 19th century, it is astounding that Frege's books ever went into print. Nothing is known about the mental health of the typesetter. Apart from the notation, Frege's system is strikingly elegant and surprisingly powerful. Unfortunately, it suffers from a major defect: it is inconsistent.

6.1.2 Russell's Antinomy

This fatal flaw in Frege's system was pointed out by Bertrand Russell in a letter in 1902, just when Frege was getting ready to publish the second volume of his *Grundgesetze*. To make things worse, Russell's argument is infuriatingly simple. In modern notation it comes down to constructing the following set:

$$R = \{\, z \mid z \notin z \,\}$$

On the face of it, R may not make a whole lot of sense, but we can still ask the basic membership question: is $R \in R$? Alas, we get a contradiction either way: if $R \in R$, then, by the very definition of R, $R \notin R$. On the other hand, if $R \notin R$, then, again by the definition, $R \in R$. Either way, we have a contradiction. As a consequence, the system that Frege so painstakingly constructed turned out to be inconsistent and had to be abandoned. Here is a closer look at Russell's antinomy. For any set A, define

$$A_{-} = \{\, z \in A \mid z \notin z \,\}$$

An argument similar to the one for R shows that $A_{-} \notin A$, we have a uniform method to construct a set that is not an element of a given set. There are other ways to obtain a similar effect. For example, Cantor showed that the cardinality of $\mathfrak{P}(A)$ is strictly larger than the cardinality of A; hence, $\mathfrak{P}(A) - A$ must be non-empty. Finding a non-element in itself is unobjectionable, but a problem arises when we assume the existence of a universal **set** V of all sets: then $V_{-} \in V$ by universality, contradicting $V_{-} \notin V$ by construction. We can avoid this problem by declaring the universe $\mathbb{V}$ to be a non-set, a proper class. This makes our ontology a bit more complicated, but turns out to be a very minor irritant.

At this point one might panic and conclude that we are forced to discard the whole system. Given the usefulness of Frege's system one would do well to check exactly how serious this problem really is. After all, many sets can be constructed by Frege's axioms that are completely natural and unsuspicious. Frege personally took Russell's objection very seriously and mentioned Russell's result in an appendix to his book. He also presented a supposed remedy, which Russell originally accepted, but later both he and Frege realized that the remedy did not solve the problem either. Still, something good might come of this flaw. Certainly some physicists are undaunted by similar issues.

> How wonderful that we have met with a paradox.
> Now we have some hope of making progress.
>
> Niels Bohr

In addition, unless the reader decides to become a logician (an unlikely event after having been exposed to this text), chances are quite good that he or she will never encounter inconsistent set constructions. In fact, Wittgenstein suggested that it might be perfectly reasonable to employ an inconsistent system:

> If you based something on this system, I don't see that it would necessarily be detrimental if there were a contradiction in it, as long as this contradiction is just not used as a thoroughfare or circus, thus allowing to derive any statement whatsoever . . . The only point would be: how to avoid going through the contradiction unawares.
>
> Ludwig Wittgenstein

Fair enough, but it would make sense to explore alternatives to just ignoring inconsistencies. Russell, for example, suggested a method in 1908 to stratify sets into layers of increasing complexity (type theory), but his approach never caught on in the mathematics community, though it later had enormous influence in theoretical computer science. Ironically, Zermelo published his first axiomatization of set theory in the same year, which axiomatization, with an extension by Fraenkel, has become the standard foundational framework. At any rate, modern proof assistants tend to be organized around ideas from type theory rather than set theory, so there is a chance that the ground will shift in the future as these tools become more powerful and more generally accepted. Still, in the interest of keeping our framework as spartan as possible, we will introduce a different solution that has become the *lingua franca* in mathematics in the next section.

To be sure, Russell's antinomy is not the only one to contend with. Another well-known antinomy is due to Burali-Forti and is based on the observation that the class of ordinals, On, is well-ordered by the element-of relation. If On were a set, it's order type would be some ordinal α. But $\alpha \in On$, so it cannot possibly be the order type of On.

6.1.3 Zermelo-Fraenkel Set Theory

Here is a glimpse at a system of axioms that (apparently) avoids inconsistencies and that has become the de facto standard in mathematical reasoning: Zermelo-Fraenkel set theory, usually augmented by the Axiom of Choice. The notion of a de facto standard has to be taken with a grain of salt: a great many practicing mathematicians will reflexively refer to this system, without necessarily being familiar with any of the technical details. Still, as before with Frege's system, there is ample evidence that the axioms are perfectly adequate to express most of mathematics in a clean and precise way. Perhaps surprisingly, most of the axioms are fairly easy to accept intuitively.

We will keep Extensionality axiom since it describes a fundamental property of sets. Instead of unlimited comprehension, we select a number of modest set-existence axioms: axioms that only guarantee the existence of sets with certain narrow properties exist, a much more guarded approach. E.g., there is an axiom that says "the empty set exists," there is an axiom for unordered pairs $\{x, y\}$, for union, and so on. Needless to say, this is a bit more tedious than Frege's approach, but it has the great advantage to get us around Russell's paradox. It also makes it much more tempting to assume that this system is indeed consistent.

Here are several fairly uncontentious axioms describing simple set constructions.

Empty Set	$\forall\, z\, (z \notin \emptyset)$
Unordered Pair	$\exists\, u\, \forall\, z\, (z \in u \Leftrightarrow z = x \lor z = y)$
Union	$\exists\, u\, \forall\, z\, (z \in u \Leftrightarrow \exists\, y\, (y \in x \land z \in y))$
Comprehension	$\exists\, u\, \forall\, z\, (z \in u \Leftrightarrow z \in x \land P(z))$
Power Set	$\exists\, u\, \forall\, z\, (z \in u \Leftrightarrow z \subseteq x)$
Infinity	$\exists\, u\, (\emptyset \in u \land \forall\, z\, (z \in u \Rightarrow z \cup \{z\} \in u))$

This version of Comprehension is also referred to as Separation (the Aussonderungsaxiom): select a subset of an already given set according to some criterion P. Note that Comprehension is actually not a single axiom, but a so-called schema: we think of $P(z)$ as a formula with free variable z and there is one axiom for each choice of this formula. The axiom for the empty set is redundant, but having notation for the empty set makes it a bit easier to state other axioms. The system we have so far is essentially Zermelo set theory (Z) from 1908, and this system suffices for quite a few arguments. Alas, it fails with respect to certain constructions that we take for granted. For example, define $A_0 = \mathbb{N}$, $A_{n+1} = \mathfrak{P}(A_n)$; then (Z) cannot prove the existence of the set $\{A_0, A_1, A_2, \ldots\}$. To address this problem, Fraenkel and Skolem proposed an additional Axiom of Replacement, which deals with applying a function to a set and is a bit more complicated to state. A binary predicate Q is called a functional property if

$$\forall\, w, u, v\ \ (Q(w,u) \land Q(w,v) \Rightarrow u = v)$$

For any functional property Q we have the following axiom:

Replacement Axiom	$\exists\, u\, \forall\, z\, (z \in u \Leftrightarrow \exists\, w\, (w \in x \land Q(w,z)))$

The replacement axiom says in effect that we can apply a function to a set, and get back another set. Loosely speaking, $\{\, f(w) \mid w \in x \,\}$ is a set.

Another problem is induction with respect to $\in$, a method that we have used in many places. If we think of sets as being constructed in layers, it is natural that we would not want to allow a situation where

$$x_0 \in x_1 \in \ldots \in x_k \in x_0$$

Nor would we want infinite descending chains with respect to $\in$. To rule out this kind of possibility, von Neumann proposed an Axiom of Foundation:

Foundation	$x = \emptyset \lor \exists\, u \in x\, (u \cap x = \emptyset)$

As stated, this axiom may seem slightly cryptic. To see why it makes sense, again think of the universe of all sets as being constructed in stages. The elements of x must have appeared at certain stages prior to x. Now consider an element u that appears at the earliest such stage: by necessity, this u will have no elements also appearing in x, exactly as the axiom demands. This opens the door to induction along $\in$: if property $\varphi(x)$ is inherited by x from all $y \in x$, then φ holds universally.

Theorem 6.1.1 *Assume all axioms other than Foundation. Then Foundation is equivalent to the $\in$-induction principle.*

Proof. Assume Foundation, and let's ignore the parameters $\boldsymbol{z}$. Suppose $\varphi(x)$ fails and define

$$u = \{\, y \in \mathsf{TC}(\{x\}) \mid \neg\varphi(y) \,\}$$

Then u is not empty and must have an $\in$-minimal element y. The elements of y are in $\mathsf{TC}(\{x\}) - u$, so $\varphi(z)$ holds for all $z \in y$. Contradiction.

For the opposite direction, let us call a set x *regular* if

$$\forall\, y\, (x \in y) \Rightarrow \exists\, z \in y\, (z \cap y = \emptyset)$$

So Foundation means that every set is regular. Now assume that $x \in z$ has only regular elements. Then x could be a minimal member of z, otherwise $x \cap z \neq \emptyset$. But then any $y \in x \cap z$ is regular, and z still has a minimal element. □

Note that this axiomatic approach to set theory makes no attempt whatsoever to define ontologically what a set is or to define the membership relation. Rather, we pin down a number of basic properties of the objects in the universe and the binary relation on them. For example, one could interpret sets as a kind of pointed directed graph, where the root represents the set, and the membership relation is expressed in terms of directed edges. So, axioms may appear trivial if one thinks about the interpretation as "collections," but that's in the eye of the beholder.

Also note that some axioms that have turned out to be quite useful in mathematical practice are not quite so obvious and have indeed produced quite a bit of controversy. The most important one is the famous Axiom of Choice (AC). It is indispensable for many arguments in mathematics, but, as we will see, it also has some nearly absurd consequences.

Suppose we have a collection $(A_i)_{i \in I}$ of non-empty sets that are pairwise disjoint: $i \neq j$ implies $A_i \cap A_j = \emptyset$. We would like to construct a choice set C for A_i: a set that selects exactly one element from each A_i:

$$C \cap A_i = \{a_i\} \quad \text{for all } i \in I.$$

As an indication of the difficulties involved, consider the case when the A_i are sets of reals. We would need a method to select a particular real from an arbitrary set. Concretely, think of the reals as given by Dedekind cuts, section 4.3. There is no obvious way to pick one Dedekind cut out of a whole collection of them. The cardinality of A_i is not the key issue here, even when all these sets have just two elements it is not clear how to select one of them. Russell famously explained the problem by pointing out that we can easily make a selection given and infinite set of shoes, but we cannot when confronted with a set of socks[2].

At any rate, the existence of a choice set is guaranteed by the Axiom of Choice, The need for this axiom is perhaps surprising since one might believe that Comprehension could be used to extract a suitable subset of $\bigcup A_i$. However, we are dealing with abstract sets and there is no systematic way to define a suitable element a_i uniformly for all $i \in I$. It has been become clear in the last century that some form of Choice is very natural in many mathematical constructions; the axiom is often used without further mention.

Definition 6.1.1 *Zermelo-Fraenkel set theory* (ZF) *is defined by the axioms listed above.* (ZFC) *is* (ZF) *plus the Axiom of Choice.*

In fact, it turns out that against the backdrop of set theory, a great many cherished mathematical principles are equivalent to Choice: one can be proven from the other.

[2] One has to assume that socks were made differently in Russell's days.

Theorem 6.1.2 *The following assertions are all equivalent to the Axiom of Choice.*[3]

- *The Well-Ordering Principle: every set can be well-ordered.*
- *Zorn's Lemma: every partial order in which every chain has an upper bound contains a maximal element.*
- *Every vector space has a basis.*

One can think of the chains in Zorn's Lemma as totally ordered subsets of the given poset. If all such chains are bounded from above, then the whole order must already have a maximal element. This may sound a bit contrived, but it is well aligned with the flow of many arguments in mathematics. For example, consider some vector space V. Let $\mathcal{I}$ be the collection of all independent subsets of V, ordered under set inclusion. Any chain $C \subseteq \mathcal{I}$ has $\bigcup C$ as an upper bound. Hence there is a maximal independent set, which can be seen to be a basis. The lemma was popularized in particular by Bourbaki and more or less summarizes the extent in which logical tools were used in their monumental effort. We won't worry about foundational issues and simply assume either and all of the above.

Note that there is an interesting idea hiding in these equivalences: instead of asking what logical axioms one needs to prove such-and-such mathematical theorem, we can reverse the question: what logical axioms can be extracted from a theorem, given some reasonable and weak background theory. This area is referred to as reverse mathematics and has produced a number of surprising results (lots of math can be neatly organized into just a handful of logical theories).

If the Axiom of Choice is indeed so useful, and even makes intuitive sense, why not just adopt it as a plain axiom and not make a big fuss about it? Obviously, we would not want to have to deal with linear algebra without having a basis for every vector space. Because Choice also has a dark side, a few extremely bizarre consequences. Here is a theorem whose proof requires the Axiom of Choice that seems to make an entirely absurd claim. Note that the theorem is usually referred to as a paradox; but it is a simply a correct albeit unexpected theorem of (ZFC).

Theorem 6.1.3 (Banach-Tarski Paradox) *A unit sphere can be decomposed into finitely many pieces that can be reassembled to form a sphere of radius 2.*

A more dramatic way of expressing this would be to say that a golf ball can be cut up into pieces that can be reassembled to form a sphere the size of the sun. We are not going to explain exactly what is meant by decompose and reassemble in this context, but the operations are perfectly reasonable. The caveat here is that the pieces produced by the decomposition process are very strange and do not possess a simple geometric structure. In particular we cannot assign qualities such as "volume" to them and refute the claim by pointing out that volume is a property that remains invariant under the operations of decomposition and reassembly. By contrast, see the polygonal decomposition result in section 2.2.

Classes and Sets

There is another technical difficulty with the Zermelo-Fraenkel approach: some perfectly reasonable collections such as the collection of all sets, all ordinals, all groups, all total orders, and so, are too large to be classified as "sets." There are ways to deal with these problems systematically: one introduces the notion of a second type of collection, a class that is supposed to accommodate these overly large "sets." In particular Gödel-Bernays-von

[3] "The Axiom of Choice is obviously true, the well-ordering principle obviously false, and who can tell about Zorn's lemma?" Jerry Bona

Neumann set theory is an often used system. In a nutshell, one allows for classes whose elements are sets, but classes cannot be nested themselves. Formation is unrestricted

$$X = \{\, x \mid \varphi(x) \,\}$$

but x is required to range over sets whereas X may be a class (and φ cannot quantify over class variables). Some classes are sets, but there are others such as the universe $V = \{\, x \mid x = x \,\}$ that are proper classes (not equal to any set). As a consequence, we are not allowed to form the object $\{V\}$. Russell's "set" $\{\, x \mid x \notin x \,\}$ is another proper class, so no contradiction ensues.

Von Neumann suggested an ingenious axiom to determine which classes are proper: a class C is proper if, and only if, there is a bijection between V and C. This single axiom encapsulates comprehension, replacement and choice all at once.

6.1.4 * The Constructible Universe

Imposing order on the collection of all sets V in terms of a few well-chosen and succinct axioms turns out to be rather more difficult than one might expect. For example, there is no easy answer to Cantor's old question about the size of the powerset of $\mathbb{N}$, short of simply picking a number by fiat. Difficult, but not impossible as demonstrated by Gödel in 1938. Gödel showed how to construct a model of (ZF) in a manner similar to von Neumann's hierarchical V, but with one critical difference: von Neumann applies the power set operator to obtain the next level, Gödel only allows sets that are first-order definable at the current level. As a consequence, every set in Gödel's universe has an explicitly description in terms of a first-order formula (albeit with parameters). This can be used for example to show that Choice holds in this universe.

For any set A define

$$\mathsf{FO}(A) = \{\, \{\, x \subseteq A \mid \langle A, \in \rangle \models \phi(x, \boldsymbol{p}) \,\} \mid \phi \text{ first-order}, \boldsymbol{p} \in A \,\}$$

to be the collection of all subsets of A that are definable by a first-order formula over the structure $\langle A, \in \rangle$, with parameters from A.

Definition 6.1.2 (Constructible Universe) *By induction on ordinals, define $L_0 = \emptyset$ and*

$$L_\beta = \bigcup_{\alpha < \beta} \mathsf{FO}(L_\alpha)$$

Lastly, set $\mathbb{L} = \bigcup L_\alpha$ where α ranges over all ordinals.

Distinguishing between different types of ordinals as usual, this means that

$$\begin{aligned} L_0 &= \emptyset \\ L_{\alpha+1} &= \mathsf{FO}(L_\alpha) \\ L_\lambda &= \bigcup_{\alpha < \lambda} L_\alpha \end{aligned}$$

Thus L is a large transitive class that contains in particular all ordinals, but may be slimmer than V. The assumption that $\mathbb{L}$ already forms the whole universe is usually written as $\mathbb{V} = \mathbb{L}$ and can be adopted as an additional axiom. The first ω levels are not particularly interesting: we simply construct all hereditarily finite sets, $L_\omega = \mathsf{HF}$. But $L_{\omega+1}$ already contains all arithmetical sets, and all hyperarithmetical sets appear at level ω_1^{CK}. It is a good exercise to verify that the usual (ZF) axioms hold in $\mathbb{L}$. Moreover, Choice and the

Generalized Continuum Hypothesis also hold true in $\mathbb{L}$. Hence, neither Choice nor GCH can be refuted in (ZF).

The definition involving first-order definability may seem somewhat cryptic: we have a good understanding of what happens at level ω, but it is far from clear what subsets of L_β will be definable for some arbitrary ordinal β. There is an alternative definition that avoids logical tools and builds the requisite sets directly, using only set theoretic operations: set difference, unordered pair, ordered pair, Cartesian product, taking the support of a binary relation, and performing a permutation of an ordered triple. Applying all these operations to $L_\alpha \cup \{L_\alpha\}$ will produce the same constructible universe as the original definition from above.

It is a good exercise to verify that the usual (ZF) axioms hold in $\mathbb{L}$. Moreover, Choice and the Generalized Continuum Hypothesis also hold true in $\mathbb{L}$. Hence, neither Choice nor GCH can be refuted in (ZF) .

Exercises

Exercise 6.1.1 Prove the Kuratowski pairing lemma 1.1.3.

Exercise 6.1.2 Discuss the Wiener pairing function $\{\{\{x\}, \emptyset\}, \{\{y\}\}\}$. and the Hausdorff pairing function $\{\{x, 1\}, \{y, 2\}\}$. How about $\{x, \{x, y\}\}$?

Exercise 6.1.3 Find another way to implement pairs as sets.

Exercise 6.1.4 Does the attempt to define pairs via $(x, y) = \{x, \{x, y\}\}$ succeed?

Exercise 6.1.5 Extend the Kuratowski pair to a map $V^{\geq 2} \to V$ by left-associativity. Is it true that $\langle a_1, \ldots, a_n \rangle = \langle b_1, \ldots, b_m \rangle$ implies $n = m$? Find a proof or counterexample.

Exercise 6.1.6 Explain the relationship between Foundation and the existence of descending $\in$-chains.

Exercise 6.1.7 Explain the relationship between Foundation and the existence of $\in$-minimal elements in the transitive closure of a set.

Exercise 6.1.8 Show that the constructible universe L from above can also be constructed in stages by applying the following operations to $L_\alpha \cup \{L_\alpha\}$: set difference, unordered pair, ordered pair, Cartesian product, taking the support of a binary relation, and performing a permutation of an ordered triple.

Exercise 6.1.9 Show that $(A \times B) \cup (C \cup D) \subseteq (A \cup C) \times (B \cup D)$. How about equality?

Exercise 6.1.10 Assume A, B and C are non-empty. Show that $A \times B = B \times A$ iff $A = B$. Show that $A \times (B \times C) \neq (A \times B) \times C$.

Seven

Logic

Logic is the science of reasoning. For our purposes, we are only interested in mathematical logic, the part that pertains to reasoning in mathematics and computer science. It is fortunate that in these areas reasoning is fairly well understood, to the point of being capable of mechanization and automation, at least in part.

Statements such as "17 is a prime number and odd" or "assertion A implies assertion B" or "upon completion of the algorithm, the stack S is empty" are the building blocks of many arguments in mathematics and computer science. For the most part, they are used in the customary, informal fashion. At least to the expert, their meaning as well as the proper way to manipulate them is clear and requires no further discussion. This approach has proven enormously successful as witnessed by the fact that most theorems established by informal proofs have withstood more probing, formal investigation. In fact, a member of the Italian school of algebraic geometry is supposed to have quipped:

> Intuition is the aristocratic way of discovery, rigour the plebeian way.
>
> Federigo Enriques

Is there any reason, then, to join the plebs? First off, a better understanding of the formal aspects of reasoning is generally helpful to organize arguments, even when they are presented in an informal setting. At the very least, it helps to avoid inaccuracies and errors. There is a persistent myth that mathematical arguments are entirely impervious to error, in particular when they are put forth by well-known practitioners. Here is a quote from a letter by C.G.J. Jacobi to A. von Humboldt that should dampen expectations a bit.

> If Gauss says he has proved something, it seems very probable to me; if Cauchy says so, it is about as likely as not; if Dirichlet says so, it is certain.
>
> C.G.J. Jacobi

The 19th century saw developments that forced the issue, in particular the discovery of "pathological" objects such as continuous functions that a nowhere differentiable, or the realization that the unit interval has the same cardinality as the unit square. Russell's set is another object that causes concern. The response to these difficulties was the systematic development of formal logic by Frege, Peano, Hilbert and others; to a lesser degree, the fundamental work of Bourbaki also contributed to establishing better and more reliable foundations. From a computational perspective, there is another compelling reason to pay close attention to logic: as early as 1947, Alan Turing foresaw a need for symbolic logic arising from the increased use of computers. In an address to London Mathematical Society he made a prescient prediction:

> I expect that digital computing machines will eventually stimulate a considerable interest in symbolic logic . . . The language in which one communicates with these machines . . . forms a sort of symbolic logic.
>
> Alan Turing

It has turned out that not just communication with computers often involves logic, reasoning about their properties and capabilities also depends very much on formal logic. A now classical example for this claim are correctness proofs of software systems. Given the complexity and size of these systems, their proper performance can no longer be established by careful testing and inspection. Instead, one has to formalize the properties of the system and the specifications it is supposed to satisfy. One can then bring methods from symbolic logic to bear and show that the system indeed works as required. Moreover, a detailed understanding of the computational aspects of logic is necessary to construct these correctness proofs: the arguments are typically so large and involved that they have to be handled by a computer.

Our focus in this chapter will be the basic ideas, in particular the syntax and semantics of first-order logic. As it turns out, a careful discussion of a logical system involves a great many fiddly details. This will be appreciated most clearly by anyone willing to take on the Implementation Challenge: build a proof checker or a theorem prover, not in theory but in actual code. We will stay out of the weeds.

7.1 True and False

7.1.1 Boolean Functions

Much of the material in the following chapters is motivated by a simple challenge: determine whether a particular assertion is true or false. For example, we know that the assertion "there are infinitely many primes" is true, "all primes are odd" is false, and the truth value of "there are infinitely many prime twins" is currently an open problem (there is much evidence that the statement will turn out to be true, but there is no proof). To avoid confusion, we introduce special symbols for the truth values "true" and "false" and write tt for true and ff for false.

Definition 7.1.1 *The set of Boolean values is* $\mathbb{B} = \{\mathsf{ff}, \mathsf{tt}\}$. *A Boolean function is any map* $\mathbb{B}^n \to \mathbb{B}$ *where* $n \geq 0$.

Occasionally we will also consider vector-valued functions $\mathbb{B}^n \to \mathbb{B}^m$, in particular in connection with circuits. Note that there are 2^{2^n} such functions, so we will only be able to handle small values of n:

n	1	2	3	4	5	6	7	8
2^{2^n}	4	16	256	65536	4.3×10^9	1.8×10^{19}	3.4×10^{38}	1.2×10^{77}

For example, the number of Boolean functions on 20 variables is $2^{2^{20}} = 6.7 \times 10^{315652}$, a stupefyingly large number. Though $\mathbb{B}^{20} \to \mathbb{B}$ is finite, it might as well be infinite. Still, let us try to develop a little taxonomy: what kind of Boolean functions are there for small values of n?

The case $n = 0$ only yields 2 functions: the constants ff and tt. For a single argument, we get 4 functions, the 2 constants, plus

- the identity H_{id}, defined by $H_{\mathrm{id}}(x) = x$, and
- negation H_{not}, defined by $H_{\mathrm{not}}(\mathsf{ff}) = \mathsf{tt}$ and $H_{\mathrm{not}}(\mathsf{tt}) = \mathsf{ff}$.

The case $n = 2$ is much more interesting: there are 16 binary Boolean functions. However, several of them depend on only one or none of the arguments. A function that depends on all its arguments is called essential. Inessential functions can be reduced essential ones by projections and composition. More precisely, a n-ary function f depends on its ith argument if

$$f(x_1, \dots, x_{i-1}, a, x_{i+1}, \dots, x_n) \neq f(x_1, \dots, x_{i-1}, b, x_{i+1}, \dots, x_n)$$

for some $a, b, x_1, \dots, x_n$. There are 10 essential binary Boolean functions. The following examples correspond naturally to conjunction, disjunction, material implication, bi-implication, exclusive or and not-and, and are particularly interesting. Informally, these correspond to logical "and," "or," "implies," and so on, which correspondence accounts for the symbols we have chosen for these functions:

x	y	$H_{\mathrm{and}}(x,y)$	$H_{\mathrm{or}}(x,y)$	$H_{\mathrm{imp}}(x,y)$	$H_{\mathrm{equ}}(x,y)$	$H_{\mathrm{xor}}(x,y)$	$H_{\mathrm{nand}}(x,y)$
ff	ff	ff	ff	tt	tt	ff	tt
ff	tt	ff	tt	tt	ff	tt	tt
tt	ff	ff	tt	ff	ff	tt	tt
tt	tt	tt	tt	tt	tt	ff	ff

For example, the assertion $H_{\mathrm{imp}}(H_{\mathrm{and}}(x, y), z) = \mathsf{tt}$ expresses the fact that in the presence of both x and y we may infer z. In such logical arguments the number of atomic propositions is usually small, perhaps a dozen or so. But in circuit design and in some algorithmic applications one has to contend with Boolean functions of hundreds or even many thousands of variables. It is thus of interest to be able to decompose such complicated functions into simpler ones.

Notational Convention: Before we continue, let us adjust our notation slightly. Examples like the table above show that the use of tt and ff for "true" and "false," while formally correct, leads to some amount of visual clutter. From now on, following a convention in some programming languages, we will abuse notation and often write 1 for tt and 0 for ff. Correspondingly, we also refer to maps $\mathbf{2}^k \to \mathbf{2}$ as Boolean functions. No confusion will arise from this, and legibility will be improved. If need be, we can always revert to our formal notation.

While there are numerous Boolean functions, they can all be obtained from just a few well-chosen ones by functional composition.

Theorem 7.1.1 *All Boolean functions can be generated from* 0*,* H_{not} *and* H_{or}*.*

Proof. These functions are nullary, unary and binary, respectively. The first step is to generalize disjunction to an arbitrary number of arguments. By induction set

$$\begin{aligned} H() &= 0 \\ H(x_1, \dots, x_n) &= H_{\mathrm{or}}(x_1, H(x_2, \dots, x_n)) \end{aligned}$$

Thus $H(x_1, \dots, x_n)$ yields 1 iff at least one of the arguments is 1. For simplicity, we write H_{or} for the multiadic version as well. Likewise set

$$H_{\mathrm{and}}(x_1, \dots, x_n) = H_{\mathrm{not}}(H_{\mathrm{or}}(H_{\mathrm{not}}(x_1), \dots, H_{\mathrm{not}}(x_n)))$$

so that $H_{\text{and}}(x_1, \ldots, x_n)$ is 1 iff all of the arguments are 1.

Now consider an arbitrary Boolean function $f : \mathbf{2}^n \to \mathbf{2}$. Let $T = \{\, \boldsymbol{a} \in \mathbf{2}^n \mid f(\boldsymbol{a}) = 1 \,\}$ be the collection of all inputs that evaluate to 1. For any $\boldsymbol{a} \in T$, define z_i to be x_i if $a_i = 1$ and $z_i = H_{\text{not}}(x_i)$ otherwise. Define $f_{\boldsymbol{a}}$ to be $H_{\text{and}}(z_1, \ldots, z_n)$, so that $f_{\boldsymbol{a}}(\boldsymbol{x}) = 1$ exactly when $\boldsymbol{x} = \boldsymbol{a}$. But then

$$f(\boldsymbol{x}) = H_{\text{or}}(f_{\boldsymbol{a}_1}(\boldsymbol{x}), \ldots, f_{\boldsymbol{a}_m}(\boldsymbol{x}))$$

where m is the cardinality of T. □

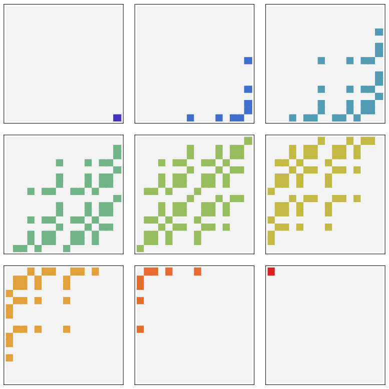
The last theorem is often expressed as follows: the set $\{H_{\text{not}}, H_{\text{or}}\}$ is adequate or forms a basis for all (non-nullary) Boolean functions. Another such basis is $\{H_{\text{not}}, H_{\text{and}}\}$. The function H_{nand} by itself is also a basis. The reason we added the constant 0 is simple convenience, otherwise we would have to distinguish between constant Boolean functions and all the others.

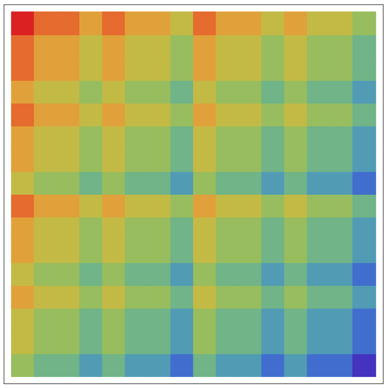
As we already mentioned, for large n, the space $\mathbf{2}^n \to \mathbf{2}$ is huge, and we have little understanding of all these functions. However, some n-ary Boolean functions have a clear combinatorial meaning and are relatively easy to analyze.

Definition 7.1.2 (Threshold Functions) *A threshold function* thr^n_m, $0 \le m \le n$, *is an n-ary Boolean function defined by*

$$\text{thr}^n_m(\boldsymbol{x}) = \begin{cases} 1 & \textit{if } \#(\, i \bigm| x_i = 1) \ge m, \\ 0 & \textit{otherwise.} \end{cases}$$

Note that lots of Boolean functions can be defined in terms of threshold functions: thr^n_0 is constant 1, thr^n_1 is n-ary disjunction, thr^n_n is n-ary conjunction, and $H_{\text{and}}(\text{thr}^n_k(\boldsymbol{x}), H_{\text{not}}(\text{thr}^n_{k+1}(\boldsymbol{x})))$ is the counting function: "exactly k out of n." Here is an image of the nine counting functions for $n = 8$ on the left, and their overlay on the right. It is a good exercise to figure out what this picture means. A hint: the individual matrices have size 16 × 16.





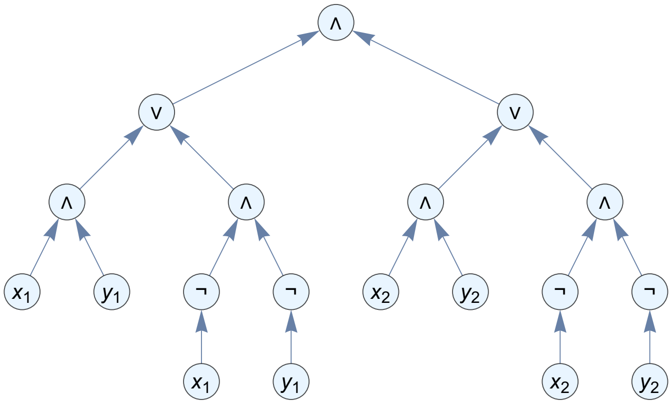
7.1.2 Boolean Circuits

Bases for Boolean functions are important since they allow us to decompose complicated functions into simpler components. A particularly fruitful way to think about these decompositions is to interpret them as a kind of abstract circuit. A circuit has special input nodes at which nodes the arguments of the function are provided. The input then propagates through several layers of gates and finally reaches the output nodes. The gates are just Boolean functions, typically chosen from a particular basis. We only consider feedback-free circuits.

As a simple example consider the equality function $f : \mathbf{2}^{2n} \to \mathbf{2}$ defined by

$$E_n(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} 1 & \text{if } x_i = y_i \text{ for all } i, \\ 0 & \text{otherwise.} \end{cases}$$

We will use the Boolean functions H_{not}, H_{or}, H_{and} but write the standard logical connective symbols $\{\neg, \vee, \wedge\}$ for clarity. The following circuit implements E_2.



To give a more precise definition of a circuit, fix some family of Boolean functions $\mathcal{B}$. Suppose we have directed acyclic graph $G = \langle V, E \rangle$ together with a labeling function λ. Indegree 0 nodes are called input nodes and are labeled by Boolean variables or constants. All other nodes are called gates and are labeled by functions in $\mathcal{B}$. For this to make sense, the indegree of any gate x must be the same as the arity of $f(x)$ (also, we quietly assume that the predecessors are ordered from left to right). The outdegree 0 nodes are called output nodes. We may refer to the edges of a circuit as wires.

It should be intuitively clear how to evaluate a circuit: we provide Boolean values for all the input nodes and propagate them upwards towards the output nodes, level by level, by evaluating the Boolean functions at the gates. This method depends critically on our assumption that G is acyclic. So we have an evaluation map

$$\mathsf{eval}(G, .) : \mathbf{2}^n \longrightarrow \mathbf{2}^m$$

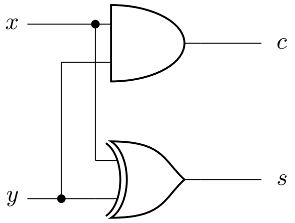
In fact, given any reasonable representation of the circuit, evaluation can be handled in linear time, see the exercises.

We can now say that circuit G realizes the Boolean function $f : \mathbf{2}^n \to \mathbf{2}^m$ if $\mathsf{eval}(G, \boldsymbol{x}) = f(\boldsymbol{x})$ for all $\boldsymbol{x} \in \mathbf{2}^n$. A key problem in circuit design is to find small circuits that realize a given Boolean function. This turns out to be rather difficult in a strict technical sense, see the chapter on complexity theory.

A slightly more complicated example is the construction of an addition circuit for binary numbers; we confine ourselves to a 1-bit adder with carry. First, a half-adder the realizes the function given by

x	y	s	c'
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

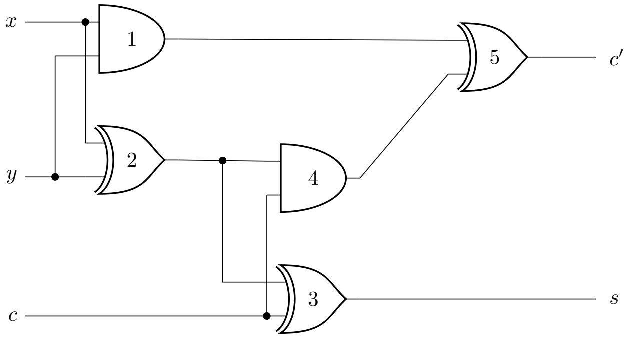
Using the standard engineering symbols for and-gates (top) and xor-gates (bottom) we get the following circuit diagram.



In the second step we combine two half-adders to get a circuit the realizes the following Boolean function $\mathbf{2}^3 \to \mathbf{2}^2$.

x	y	c	s	c'
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Note that we allow outdegrees larger than one, so the output of a sub-circuit can be provided as input to several parts of the circuit.

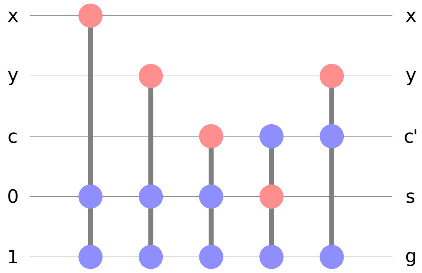


The visual representation makes it easy to understand the workings of the circuit. We will see in a while, though, that finding a good algebraic model can help in the design of small circuits.

While we are talking about one-bit adders, here is another solution to this problem, using the so-called Fredkin gate, a gate with 3 inputs and 3 outputs. The first input x is a control bit, and y and z are swapped provided that $x = 1$; otherwise they are simply copied to the output.

x	y	z	x'	y'	z'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	1	1	1

What is remarkable about this gate is its reversibility: given the outputs, we can reconstruct the inputs. This is of interest since reversible gates carry no intrinsic thermodynamic cost, at least in principle. We can build a one-bit adder out of 5 Fredkin gates as follows.



Data flow from left to right, and the red points indicate the control bit.

7.1.3 Exercises

Exercise 7.1.1 Find all binary Boolean functions that depend on both arguments and try to associate some intuitive meaning to them.

Exercise 7.1.2 Does $\{H_{\text{not}}, H_{\text{xor}}\}$ form a basis? Does $\{H_{\text{not}}, H_{\text{imp}}\}$ form a basis? Does $\{H_{\text{and}}, H_{\text{or}}\}$ form a basis?

Exercise 7.1.3 Discuss some plausible implementations of circuits and the corresponding evaluation algorithm.

Exercise 7.1.4 Show that the number of essential n-ary Boolean functions is $\sum_{i \leq n} (-1)^{n-i} \binom{n}{i} 2^{2^i}$.

7.2 Propositional Logic

We will now introduce our first logical system propositional logic, also known as sentential logic of zero-order logic. The main idea behind propositional logic is assume that we are given the truth values of a number of atomic assumptions, and we are only concerned with the task of determining the truth value of compound assertions that are built from the atomic ones. Specifically, we use logical connectives that correspond informally to connectives "not, " "and," "or," "if-then" and so forth in ordinary language. We postpone until later the question of how one can determine the truth values of the atomic propositions in the first place.

7.2.1 Language and Semantics

The first step is to fix a formal language for logical formulae, in which language we can express assertions built from our connectives. We want a sufficiently high level of detail so that someone could build a compiler that tests whether an arbitrary expression is well-formed according to our definition. As is often the case, what is good for computers is not so great for humans, so later on we will follow the standard practice of using a rather loose syntactic description, essentially just an affirmation that we are dealing with a nicely structured rectype and need not worry about pesky details. Moreover, we should be able to develop algorithms that manipulate these expressions and perform various tasks. For example, we would like to be able to determine whether a given formula is valid. As we will see, this seemingly simple effort runs into significant algorithmic difficulties, at least if one is concerned about efficiency. The central problem is that in order to apply propositional logic to a practical problem, such as checking a train schedule for correctness, produces huge formulae, and even the best algorithms mail fail to generate results within an acceptable time frame. This is in contrast to more traditional applications to formal reasoning. For example, recall the *modus ponens*: whenever we have established both "A" and "if A then B", then we may also conclude that "B". The corresponding propositional formula is tiny and can easily be shown to be valid by brute-force computation.

The Language of Propositional Logic

We need to explain what we mean by "formula of propositional logic." We want this first description to be detailed enough so someone could write a parser for these formulae. To this end we need to keep track of the arities of connectives. The most popular ones are displayed in the next table:

symbol	purpose	arity
$\top$	true	0
$\bot$	false	0
$\neg$	not, negation	1
$\wedge$	and, conjunction	2
$\vee$	or, disjunction	2
$\oplus$	exclusive or	2
$\Rightarrow$	conditional (implies)	2
$\Leftrightarrow$	biconditional (equivalent)	2
ite	if-then-else	3

To define propositional formulae, we select some of these connectives in an alphabet Σ and define a collection Var of propositional variables, typically written p, q, x, y, and the like. As usual we allow various decorations to lighten notation, so we can write x', $\widehat{x}$, x_i and so on.

Definition 7.2.1 (Propositional Formulae) *Given Σ and a collection* Var *of variables, we define the rectype of propositional formulae* $\mathsf{Pfml} = \mathsf{Pfml}(\mathsf{Var}, \Sigma)$ *as follows. For simplicity, assume $\Sigma = \{\bot, \neg, \vee, \wedge\}$.*

$$\texttt{Pfml} ::= \mathsf{Var} \mid \bot \mid \neg\,\texttt{Pfml} \mid \vee\,\texttt{Pfml}\;\texttt{Pfml} \mid \wedge\,\texttt{Pfml}\;\texttt{Pfml}$$

In the last three cases, we refer to $\neg$, $\lor$ and $\land$ as the principal connective. From the rectype perspective, the elements of Var and $\bot$ are the atoms, and $\neg$ is a constructor of arity 1, $\lor$ and $\land$ are constructors of arity 2. In practice, arities 1,2 and 3are far and away the most important, but there are other connectives of larger arities that come in handy on occasion, e.g. in the context of counting operations. The set Var of variables could be made more precise as another rectype. For instance, we could allow all expressions of the form `x10011`, a letter `x` followed by an arbitrary number of binary digits. We will ignore these details.

It is useful to organize propositional formulae into a small taxonomy:

- Variables and nullary connectives are atomic formula.
- A literal is a variable p or its negation $\neg p$.
- A compound formula is a formula that is not atomic.

Observe that we can decompose every compound formula by peeling off the principal connective and then working on the subformulae.

Unfortunately, we are currently far from any notation system a human would feel comfortable with, just think about a formula like

$$\Rightarrow \lor x \lor yz \Rightarrow \land \neg x \neg yz$$

We will relax our notation considerably without going to the trouble of giving a precise definition of the relaxed syntax; after all, we are not interested in building a real parser. The most important change to assure readability is to use infix notation for binary connectives, so we write $(p \land q)$ instead of $\land pq$. The price for infix notation is that we need parentheses for grouping. The last implication now looks like

$$\big((x \lor (y \lor z)) \Rightarrow (((\neg x) \land (\neg y)) \Rightarrow z)\big)$$

Much better, but now there are too many parentheses. In order to cut back on those, we adopt precedence rules along the lines of

$$\neg \succ \land \succ \lor \succ \Rightarrow \succ \Leftrightarrow$$

So negation binds the strongest and biimplication the weakest. In addition, one usually removes the outermost parentheses and assumes that binary connectives associate to the right, so that $x \land y \land z$ is just short for $x \land (y \land z)$. Now our implication looks like so

$$x \lor y \lor z \Rightarrow \neg x \land \neg y \Rightarrow z$$

in perfect keeping with traditional notation.

The right-association rule bears some further comment. Disjunctions and conjunctions naturally generalize to an arbitrary number of arguments, much the way addition and multiplication generalize in arithmetic. There one uses summation and product operators of arbitrary finite arity (and sometimes even of infinite arity), we can do the same here by writing

$$\bigwedge_{i=1}^{n} p_i \qquad\qquad \bigvee_{i=1}^{n} p_i$$

for conjunctions and disjunctions, respectively. We could also introduce an n-ary symbol $\bigwedge_n$ in our language, but we prefer our simpler alphabet. Our operators make sense for any number n of arguments as long as $n \geq 2$, but how about one or zero arguments? A moment's thought will reveal that we should think of $\bigwedge_{i=1}^{1} p_i$ as p_1 and of $\bigwedge_{i=1}^{0} p_i$ as $\top$.

We notice in passing that it is quite a bit harder to write a parser that can deal with our relaxed notation, rather than the original rigid prefix form. It is one of the big success stories of computer science that building such a compiler can be handled fairly easily by standard techniquesSome authors avoid the rectype approach and instead think of a formula as *well-formed formula*, a particular kind of string, and go to great lengths to establish some unique parseability lemma. Again, this is the right angle for someone trying to build a parser, but for us a formula is an element of a rectype and well-formed by definition.

Our choice of base alphabet Σ is not uncommon, but there are many alternatives. The reason for this is that we are usually interested in formulae "up to equivalence," an idea explained below, so we could use different connectives altogether. As an example, it is well-known that we could retain just $\bot$ and the binary $\Rightarrow$. We can then introduce all other standard connectives as definitional extensions: we add the appropriate symbols to our language and then define "abbreviations" as in

$$\begin{aligned}
\top &\coloneqq \bot \Rightarrow \bot \\
\neg\varphi &\coloneqq \varphi \Rightarrow \bot \\
\varphi \lor \psi &\coloneqq (\varphi \Rightarrow \bot) \Rightarrow \psi \\
\varphi \land \psi &\coloneqq (\varphi \Rightarrow (\psi \Rightarrow \bot)) \Rightarrow \bot
\end{aligned}$$

Here are some examples of propositional formulae in simplified infix notation. We retain some parens that could be eliminated according to our rules, for the sole purpose of increasing legibility. Of course, this is largely a matter of taste.

$$\begin{array}{ll}
p \land q \Rightarrow p & (p \Rightarrow \bot) \Rightarrow \neg p \\
p \land q \Rightarrow q \land p & p \Rightarrow (q \Rightarrow p) \\
p \Rightarrow q \Rightarrow p \land q & \neg(p \land q) \Rightarrow (\neg p \lor \neg q) \\
p \Rightarrow p \lor q & p \land (p \Rightarrow q) \Rightarrow q \\
\bot \Rightarrow p & (p \land q \Rightarrow r) \Rightarrow (\neg p \lor q) \Rightarrow (p \Rightarrow r)
\end{array}$$

In all these formulae, the principal connective is an implication. Intuitively, they describe valid forms of inference: for example, the fourth formula in the second column corresponds to the old *modus ponens*, if p holds and we know that from p we can infer q. It is a good exercise to extract the intuitive meaning of all these formulae.

Semantics of Propositional Logic

So far we only have syntax, we also need to pin down the meaning of a formula. Since propositional variables are per se meaningless, we have to replace them by truth values before we can talk about the truth of a formula.

Definition 7.2.2 *A truth assignment or truth valuation over variables* Var *is a map* $\mathsf{Var} \to \mathbf{2}$ *that associates a truth value with each variable.*

For any given formula φ we only need a finite truth assignment σ that is defined for all the variables occurring in φ. However, it is often more convenient to assume that the truth assignment is actually defined on all of Var, and thus applies to any formula in Pfml.

Definition 7.2.3 *Assume σ is a truth assignment. Define the corresponding truth valua-*

tion $[\![\varphi]\!]_\sigma$ *by induction on the build-up of formulae in* Pfml *as follows.*

$$\begin{aligned}
[\![\bot]\!]_\sigma &= 0 & [\![\varphi \lor \psi]\!]_\sigma &= H_{or}([\![\varphi]\!]_\sigma, [\![\psi]\!]_\sigma) \\
[\![\top]\!]_\sigma &= 1 & [\![\varphi \land \psi]\!]_\sigma &= H_{and}([\![\varphi]\!]_\sigma, [\![\psi]\!]_\sigma) \\
[\![p]\!]_\sigma &= \sigma(p) & [\![\varphi \Rightarrow \psi]\!]_\sigma &= H_{imp}([\![\varphi]\!]_\sigma, [\![\psi]\!]_\sigma) \\
[\![\neg\varphi]\!]_\sigma &= H_{not}([\![\varphi]\!]_\sigma) & [\![\varphi \Leftrightarrow \psi]\!]_\sigma &= H_{equ}([\![\varphi]\!]_\sigma, [\![\psi]\!]_\sigma)
\end{aligned}$$

This is a classical example of a recursive definition in a rectype. Any formula φ over n variables naturally induces a Boolean function $\widehat{\varphi} : \mathbf{2}^n \to \mathbf{2}$: identity a suitable truth assignment σ with a vector in $\mathbf{2}^n$, and set $\widehat{\varphi}(\sigma) = [\![\varphi]\!]_\sigma$. Note that, say, $\bot$ and $p \land \neg p$ produce two different Boolean functions, the first nullary, but the second is unary. If the base alphabet of logical connectives is sufficiently rich, the corresponding collection of these functions $\widehat{\varphi}$ contains all Boolean functions.

Our definitions are well-aligned with our intuition, perhaps with one notable exception: implication, the if-then connective. In the formula $\varphi \Rightarrow \psi$, φ is the hypothesis or premise; ψ is the conclusion or consequent. If one understands implication to mean that the hypothesis is somehow causally responsible for the conclusion, then our definition of truth of an implication is indeed not quite on target. In particular, if the hypothesis is false, the whole implication is true regardless of the conclusion: this is the principle of explosion (*ex falso quodlibet*), $[\![\bot \Rightarrow \varphi]\!]_\sigma = 1$, regardless of σ and φ. It is important to bear in mind that our definition is purely extensional, it depends only on the truth values of the two subformulae rather than their "meaning." A particular striking example is $(p \Rightarrow q) \lor (q \Rightarrow p)$, a formula that is always true, clashing head-on with our tendency to interpret implication as causation.

Another minor source of confusion are the variants that can be associated with an implication: the converse $\psi \Rightarrow \varphi$, the inverse $\neg\varphi \Rightarrow \neg\psi$ and the contrapositive $\neg\psi \Rightarrow \neg\varphi$. Note that $[\![\varphi \Rightarrow \psi]\!]_\sigma = [\![\neg\psi \Rightarrow \neg\varphi]\!]_\sigma$ so that the implication and its contrapositive always have the same truth value. There is no such connection between the implication and its converse or inverse.

In most reasonable implementations of $\mathsf{Pfml}(V)$, the truth value $[\![\varphi]\!]_\sigma$ can be computed in time linear in the size of φ. Computing the truth value of a propositional formula is essentially the same problem as evaluating a Boolean circuit, see section 7.1.2. Naturally, we are particularly interested in formulae that evaluate to true.

Definition 7.2.4 *Let σ be a truth assignment and φ a propositional formula. σ satisfies or models φ if φ is true under σ: $[\![\varphi]\!]_\sigma = 1$. A truth assignment satisfies a set of formulae Γ if it satisfies each formula in the set. In symbols:*

$$\sigma \models \varphi \qquad \text{and} \qquad \sigma \models \Gamma$$

Given a truth valuation σ, the truth set of σ is the collection Γ of formulae φ such that $\sigma \models \varphi$.

Incidentally, the turnstile notation is lifted from Frege's Begriffsschrift, nothing else has survived[1], see section 6 for a terrible example.

A truth set is by nature a semantic concept, but, in the case of propositional logic, it can be characterized in a purely syntactical fashion.

[1] Frege was ahead of his times, his groundbreaking work was poorly received. No doubt his horrible notation system had a major role to play in this rejection.

Lemma 7.2.1 *Let Γ be a set of formulae containing $\top$. Then Γ is a truth set if, and only if, the following hold:*

- $\varphi \in \Gamma$ *iff* $\neg\varphi \notin \Gamma$.
- $\varphi \wedge \psi \in \Gamma$ *iff* $\varphi \in \Gamma$ *and* $\psi \in \Gamma$.
- $\varphi \vee \psi \in \Gamma$ *iff* $\varphi \in \Gamma$ *or* $\psi \in \Gamma$.
- $\varphi \Rightarrow \psi \in \Gamma$ *iff* $\varphi \notin \Gamma$ *or* $\psi \in \Gamma$.

Proof. The direction left-to-right follows immediately from the definition of a truth valuation. For the opposite direction, note that we can recover σ from Γ: define $\sigma(p) = 1$ if $p \in \Gamma$ and $\sigma(p) = 0$ otherwise. By induction on the build-up of φ one can then show that $\sigma \models \varphi$ iff $\varphi \in \Gamma$. □

The next important step is to model the notion that an assertion follows from a collection of other assertions. In our setting, we are interested in having all truth assignments render a formula true as long as they render each formula in a given set true.

Definition 7.2.5 *Let Γ be a set of formulae and φ a propositional formula. φ is a semantic consequence of Γ if every truth assignment σ that satisfies Γ also satisfies φ: $\sigma \models \Gamma$ implies $\sigma \models \varphi$. In symbols: $\Gamma \models \varphi$.*

Here is a simple example of semantic consequence, an analogue of the famous modus ponens rule:

$$\varphi, \varphi \Rightarrow \psi \models \psi.$$

Implications do indeed model consequence naturally as can be seen from

$$\Gamma, \varphi \models \psi \quad \text{implies} \quad \Gamma \models \varphi \Rightarrow \psi.$$

As a last example, from $\Gamma, \varphi \models \bot$ it follows that $\Gamma \models \neg\varphi$.

7.2.2 Tautologies, Contradictions and Contingencies

We now have a clear definition of what we mean by a "true formula" of propositional logic, a formula that evaluates to true under all truth assignments. Here is some related terminology.

Definition 7.2.6 *A formula φ is a tautology if it is true under all truth assignments. In symbols: $\models \varphi$. A formula φ is a contradiction if $\sigma \models \varphi$ for no assignment σ. It is a contingency (or satisfiable) if $\sigma \models \varphi$ for some assignment σ*

Proposition 7.2.1 *For any formula φ we have:*

- *φ is a tautology if, and only if, $\neg\varphi$ is not satisfiable.*
- *φ is a contradiction if, and only if, φ is not satisfiable.*

It is clear that $\bot$ and $p \wedge \neg p$ are contradictions, $\top$ and $p \vee \neg p$ are tautologies. But how about a slightly more complicated formula such as

$$\varphi = (p \wedge q \Rightarrow r) \wedge (\neg p \vee q) \;\; \Rightarrow \;\; p \Rightarrow r$$

We are looking for a method, a decision algorithms, that, on input a propositional formula, will determine whether the formula is a tautology or whether it is satisfiable. The obvious brute-force approach is to substitute all possible values for p, q, and r, and check the value of the whole formula for each. To obtain the truth value of the whole formula in a systematic way it is natural to first consider appropriate sub-formulae. This method can be nicely visualized in a table, a so-called truth table, an idea discovered multiple times.

p	q	r	$(p \wedge q) \Rightarrow r$	$\neg p \vee q$	$p \Rightarrow r$	φ
0	0	0	1	1	1	1
0	0	1	1	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	0	0	1
1	0	1	1	0	1	1
1	1	0	0	1	0	1
1	1	1	1	1	1	1

So, φ is a tautology.

The truth table method is straightforward to implement: we already know how to evaluate a formula under a given truth assignment in linear time. If n is the number of variables in the formula, we only need to repeat this evaluation up to 2^n times. Alas, since many applications require thousands of variables we will need to look for alternative methods. As we will see in chapter **??**, there are good reasons to believe that the efficiency of the truth table method is difficult to surpass significantly in any tautology testing algorithm. In particular it seems unlikely that the general problem can be handled in polynomial time. The hardness of tautology testing is shared with checking for satisfiability. Still, there are practical algorithms that work quite well for many real world inputs, even when thousands of variables are involved, see 7.3.

7.2.3 Semantic Consequence and Equivalence

If the truth value of two formulae is the same under each truth assignment then they are in a sense interchangeable, though syntactically they are of course distinct. For example, as far as truth values go, there is no difference between $p \wedge q$ and $q \wedge p$.

Definition 7.2.7 *Two formulae φ and ψ are equivalent if for any truth assignment σ: $[\![\varphi]\!]_\sigma = [\![\psi]\!]_\sigma$. In symbols: $\varphi \equiv \psi$.*

Note that $\varphi \equiv \psi$ is not a propositional formula. Rather, equivalence is a binary relation on propositional formulae, but it can be expressed in terms of tautologies.

Proposition 7.2.2 *φ and ψ are equivalent if, and only if, $\varphi \Leftrightarrow \psi$ is a tautology.*

Proof. To see this, observe that $[\![\varphi \Leftrightarrow \psi]\!]_\sigma = 1$ iff $[\![\varphi]\!]_\sigma = [\![\psi]\!]_\sigma$. □

There are a number of basic equivalences that show that logical "and" is an associative operation modulo equivalence, that $\top$ is a one-element with respect to this operation, and so on. As we will see later, modulo equivalence the propositional formulae form a Boolean algebra with respect to $\wedge$, $\vee$ and $\neg$, see **??**.

Lemma 7.2.2 *The following equivalences hold for all formulae φ, ψ and ρ.*

$$\begin{aligned}
\varphi \land \bot &\equiv \bot & \varphi \lor \top &\equiv \top \\
\varphi \land \top &\equiv \varphi & \varphi \lor \bot &\equiv \varphi \\
\varphi \land \varphi &\equiv \varphi & \varphi \lor \varphi &\equiv \varphi \\
\varphi \land \psi &\equiv \psi \land \varphi & \varphi \lor \psi &\equiv \psi \lor \varphi \\
\varphi \land (\psi \land \rho) &\equiv (\varphi \land \psi) \land \rho & \varphi \lor (\psi \lor \rho) &\equiv (\varphi \lor \psi) \lor \rho \\
\varphi \land (\psi \lor \rho) &\equiv (\varphi \land \psi) \lor (\varphi \land \rho) & \varphi \lor (\psi \land \rho) &\equiv (\varphi \lor \psi) \land (\varphi \lor)
\end{aligned}$$

Negation produces

$$\begin{aligned}
\neg\neg\varphi &\equiv \varphi \\
\varphi \land \psi &\equiv \neg(\neg\varphi \lor \neg\psi)
\end{aligned}$$

Since $\varphi \Rightarrow \psi$ is equivalent to $\neg\varphi \lor \psi$ and $\neg(\varphi \land \neg\psi)$, we could have omitted implication from our primitive connectives and instead introduced it as a defined operation. The following equivalences involving implications are often useful.

Lemma 7.2.3 *The following equivalences hold for all formulae φ, ψ and ρ.*

$$\begin{aligned}
\varphi \Rightarrow (\psi \land \rho) &\equiv (\varphi \Rightarrow \psi) \land (\varphi \Rightarrow \psi) & \varphi \Rightarrow (\psi \lor \rho) &\equiv (\varphi \Rightarrow \psi) \lor (\varphi \Rightarrow \psi) \\
(\psi \land \rho) \Rightarrow \varphi &\equiv (\psi \Rightarrow \rho) \lor (\rho \Rightarrow \varphi) & (\psi \lor \rho) \Rightarrow \varphi &\equiv (\psi \Rightarrow \rho) \land (\rho \Rightarrow \varphi)
\end{aligned}$$

Note that the last two equivalences are a bit counterintuitive, the connectives switch.

It is clear that semantic equivalence is indeed an equivalence relation. But a much stronger property holds: semantic equivalence is a congruence with respect to the logical connectives. This means that

$$\varphi \equiv \varphi', \psi \equiv \psi' \quad \text{implies} \quad \varphi \land \psi \equiv \varphi' \land \psi'$$

and likewise for the other connectives. This has the important consequence that we can replace any subformula in a propositional formula by an equivalent one and obtain an equivalent formula. More precisely, write $\varphi[\alpha]$ to indicate that α is a subformula of φ. Then $\varphi[\beta]$ denotes the result of the substitution replacing α by β in φ.

Lemma 7.2.4 $\alpha \equiv \beta$ *implies* $\varphi[\alpha] \equiv \varphi[\beta]$

There is a striking symmetry in the equivalences involving $\land$ and $\lor$ above: interchanging $\land$ and $\lor$, as well as $\bot$ and $\top$ seems to produce new equivalences. To see why, define the dual φ^{op} for any propositional formula φ by induction on the build-up as follows.

$$\begin{aligned}
\top^{\mathrm{op}} &= \bot & \bot^{\mathrm{op}} &= \top \\
p^{\mathrm{op}} &= p & (\neg\varphi)^{\mathrm{op}} &= \neg\varphi^{\mathrm{op}} \\
(\varphi \land \psi)^{\mathrm{op}} &= \varphi^{\mathrm{op}} \lor \psi^{\mathrm{op}} & (\varphi \lor \psi)^{\mathrm{op}} &= \varphi^{\mathrm{op}} \land \psi^{\mathrm{op}}
\end{aligned}$$

The dual φ^{op} can be computed in linear time. Abusing notation slightly, we write σ^{op} for the truth assignment obtained from σ by flipping all the values.

Proposition 7.2.3 $[\![\varphi^{\mathrm{op}}]\!]_\sigma = 1 - [\![\varphi]\!]_{\sigma^{\mathrm{op}}}$

Proof is by straightforward induction on the build-up of φ. It follows that we can dualize any semantic equivalence:

$$\alpha \equiv \beta \quad \text{implies} \quad \alpha^{\mathrm{op}} \equiv \beta^{\mathrm{op}}$$

In some sense, this cuts our work in half: once we have the truth value of α, we also have the truth value of α^{op}.

7.2.4 Positive and Negative Parts

Are there any reasonable methods to test whether a formula is a tautology by hand? As we have mentioned, testing whether a formula is a tautology is difficult in a certain technical sense, so we should not hold much hope, but perhaps we can find a decent method at leas for formulae of modest size. The idea in the positive/negative part method is to try to find subformulae ψ of the main formula φ so that, if ψ is satisfied, φ itself must also be satisfied; alternatively, if ψ is not satisfied, then φ is satisfied. For example, if $\varphi = \psi_1 \vee \psi_2$, then it suffices to satisfy one of the ψ_i. On the other hand, for $\varphi = \psi_1 \Rightarrow \psi_2$ to be true, it suffices to make ψ_1 false, or ψ_2 true. We need to systematically identify appropriate subformulae. To this end, think of traversing the parse tree of φ starting at the root and labeling the nodes (i.e., the subformulae). We use labels *positive* and *negative.*

- The root is positive.
- At any $\neg$-node, the sign switches.
- If a $\vee$-node is positive, both of its children are positive.
- If a $\Rightarrow$-node is positive, its left child is negative, and its right child is positive.
- If a $\wedge$-node is negative, both of its children are negative.
- If a $\Rightarrow$-node is negative and its right child is $\bot$, its left child is positive.

A subformula φ of ψ is a positive/negative part if it is labeled positive/negative. Here is a little example where this method works very well:

$$\varphi = (p \wedge \neg q) \Rightarrow (p \vee \neg s)$$

Here the positive parts, other than φ itself, are q, $p \vee \neg q$, p and $\neg s$; the negative parts are $p \wedge \neg q$, p, $\neg q$ and s. The variable p occurs as a negative and a positive part, and so φ is a tautology. The method also works to show that $p \Rightarrow q \Rightarrow p$ is a tautology.

Unfortunately, things do not always work out so well. Take

$$\varphi = \neg(\neg(p \Rightarrow q) \wedge (q \vee r))$$

The positive parts, other than φ itself, are $p \Rightarrow q$ and q_1, where the subscript indicates the first occurrence of q. The negative parts are $\neg(p \Rightarrow q) \wedge (q \vee r)$, $\neg(p \Rightarrow q)$, $q \vee r$, and p. Note that q_2 and r are neither positive nor negative parts.

We need to verify that these definitions are sound.

Lemma 7.2.5 *Suppose some truth assignment σ satisfies a positive part of φ or does not satisfy a negative part. Then σ satisfies the whole formula.*

Proof. Write $\varphi = \Phi[\psi]$ to indicate that ψ is a subformula of φ. More precisely, think of Φ as being a parse tree with a special leaf X, and $\Phi[\psi]$ as the result of replacing X by ψ. To indicate whether we have a positive or negative part, we write $\Phi^+[\psi]$ and $\Phi^-[\psi]$, respectively.

The proof is by induction on Φ. If $\Phi^+ = X$ we have $\varphi = \Phi^+[\psi] = \psi$ and the claim holds. The other cases where X appears positively are $\Phi^+ = X \vee \theta$, $\Phi^+[\theta \vee X]$, $\Phi^+[\theta \Rightarrow X]$. For a negative occurrence of X we have $\Phi^+[\neg X]$ and $\Phi^+[X \Rightarrow \theta]$. Similarly, X appears positively in $\Phi^-[X \Rightarrow \bot]$ and $\Phi^-[\neg X]$; negatively in are $\Phi^- = X \vee \theta$ and $\Phi^-[\theta \vee X]$.

In either case, the standard bottom-up evaluation shows that the whole formula is satisfied as long as the assumptions of the theorem hold. □

To deal with the problem of our labeling algorithm not reaching the leaves of the parse tree (variables or $\bot$), call a positive part blocking if it is a variable, $\bot$ or of the form $\varphi \wedge \psi$; a negative part is blocking if it is a variable, $\bot$ or of the form $\varphi \lor \psi$ or $\varphi \Rightarrow \psi$, where $\psi \neq \bot$. We can still derive information about the truth value of the whole formula, given the truth values of all the blocking parts.

Lemma 7.2.6 *Let φ be a formula and σ a truth assignment that fails to satisfy any blocking positive part of φ, but does satisfy all blocking negative parts. Then σ satisfies not positive part of φ, and satisfies all negative parts of φ.*

The proof is in the exercises. A formula is admissible if it contains no positive part of the form $\varphi \land \psi$, nor a negative part of the form $\varphi \lor \psi$ or $\varphi \Rightarrow \psi$ where $\psi \neq \bot$. In other words, every subformula is labeled, as in the first example above. For admissible formulae we can give a simple syntactical characterization of tautologies, a characterization that can easily be tested in linear time.

Theorem 7.2.1 *An admissible formula is a tautology if, and only if, it has a negative part $\bot$, or it has some variable p as both a positive and negative part.*

Proof. The implication from right to left is an easy consequence of lemma 7.2.5.

For the direction from left to right, we construct a non-satisfying truth assignment for any admissible formula φ that is not of the form in the theorem. Since φ is admissible, every blocking positive/negative-part of φ must be a propositional variable or $\bot$. In particular, all blocking negative parts must be variables. But since φ is not both positive/negative, none of these variables can be a positive part. Now define $\sigma(p) = 1$ iff p is a negative part. Done by lemma 7.2.6. □

Of course, there are lots of inadmissible formulae that are tautologies. For instance, Peirce's Law

$$((p \Rightarrow q) \Rightarrow p) \Rightarrow p$$

is a tautology, but our analysis fails completely. The conclusion p is a positive part, the premise $(p \Rightarrow q) \Rightarrow p$ is a negative part, and that is where it ends.

The question arises whether inadmissible tautologies also have some sort of syntactical characterization. Indeed, they do, but this time we cannot derive any computational advantage from this characterization, there are exponentially many cases to check.

Theorem 7.2.2 *Let φ be an inadmissible formula.*

Let $\varphi = \Phi^+[\alpha \land \beta]$. Then φ is a tautology if, and only if, both $\Phi[\alpha]$ and $\Phi[\beta]$ are tautologies.

Let $\varphi = \Phi^-[\alpha \lor \beta]$. Then φ is a tautology if, and only if, both $\Phi[\alpha]$ and $\Phi[\beta]$ are tautologies.

Let $\varphi = \Phi^-[\alpha \Rightarrow \beta]$. Then φ is a tautology if, and only if, both $\Phi[\alpha \Rightarrow \bot]$ and $\Phi[\beta]$ are tautologies.

Proof. First assume that both $\Phi^+[\alpha]$ and $\Phi^+[\beta]$ are tautologies. Since $[\![\alpha \land \beta]\!]_\sigma$ is always equal to $[\![\alpha]\!]_\sigma$ or $[\![\beta]\!]_\sigma$, we must have $[\![\Phi^+[\alpha \land \beta]]\!]_\sigma = [\![\Phi^+[\alpha]]\!]_\sigma$ or $[\![\Phi^+[\alpha \land \beta]]\!]_\sigma = [\![\Phi^+[\beta]]\!]_\sigma$. But the φ is also a tautology.

For the opposite direction assume wlog that $\Phi^+[\alpha]$ fails to be a tautology and choose σ such that $[\![\Phi^+[\alpha]]\!]_\sigma = 0$. But then $[\![\alpha]\!]_\sigma = 0$ and also $[\![\alpha \land \beta]\!]_\sigma = 0$. Hence $[\![\Phi^+[]\alpha \land \beta]\!]_\sigma]$ and we are done.

The other two parts are entirely similar. □

Exercises

Exercise 7.2.1 Show that the following formulae are all tautologies.

$p \Rightarrow p$	identity
$p \Rightarrow q \Rightarrow p$	introduction
$(p \Rightarrow q \Rightarrow r) \Rightarrow (q \Rightarrow p \Rightarrow r)$	interchange
$(p \Rightarrow q) \Rightarrow (q \Rightarrow r) \Rightarrow p \Rightarrow r$	cut
$(p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r$	Frege cut
$((p \Rightarrow q) \Rightarrow p) \Rightarrow p$	Pierce

Exercise 7.2.2 Show that all the implications from the previous exercise are tautologies using the positive/negative part machinery.

Exercise 7.2.3 Give a precise definition of α being a subformula of φ. Extend your definition to positive/negative parts.

Exercise 7.2.4 Prove the equivalence substitution lemma 7.2.4.

Exercise 7.2.5 Prove lemma 7.2.3.

Exercise 7.2.6 Using P/N-parts, determine which of the following formulae are tautologies:

$$p \Rightarrow (q \Rightarrow p)$$
$$(p \Rightarrow q) \Rightarrow (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$$

Exercise 7.2.7 Give a proof for lemma 7.2.6. Hint: use induction on the length of a P-part (or N-part).

Exercise 7.2.8 Give a proof for the remaining two parts of theorem 7.2.2.

7.3 Satisfiability

Over the last few decades, Satisfiability, the logical problem of determining whether a propositional formula has a satisfying truth assignment, has become one of the key problems in the theory of algorithms and deserves a separate discussion. On the one hand, it is known that the problem is computationally hard in a strict technical sense, and is thus unlikely to admit a polynomial time solution[2]. On the other hand, state-of-the-art implementations can routinely handle large formulae with tens of thousands of variables and consisting of millions of symbols.

7.3.1 Normal Forms

Negation Normal Form

We can use the equivalence substitution lemma 7.2.4 to transform propositional formulae into equivalent ones that have a special syntactical structure, a so-called normal form.

[2] Some eminent researchers like D. Knuth or L. Levin would vociferously disagree with this judgment.

The first one pushes negation all the way inside, until only propositional variables appear negated. Recall that a literal is a variable or a negated variable. A formula is in negation normal form (NNF) if it only contains negations as part of a literal.

Theorem 7.3.1 *For every formula, there is an equivalent formula in negation normal form. Moreover, the conversion can be effected in linear time.*

Proof. To see this, first eliminate all implications and biconditionals. Then push all negation signs inward, until they occur only next to a variable using the following rules for this transformation:

$$\begin{aligned} \neg(\varphi \wedge \psi) &\rightsquigarrow \neg\varphi \vee \neg\psi \\ \neg(\varphi \vee \psi) &\rightsquigarrow \neg\varphi \wedge \neg\psi \\ \neg\neg\varphi &\rightsquigarrow \varphi \end{aligned}$$

It is easy to check that these rules produce an equivalent formula and we are done by lemma 7.2.4. □

For example, we have the following transformation to NNF:

$$\begin{aligned} &\neg(p \wedge (q \vee \neg r) \wedge s \wedge \neg t) \\ &\neg p \vee \neg(q \vee \neg r) \vee \neg s \vee t \\ &\neg p \vee (\neg q \wedge r) \vee \neg s \vee t \end{aligned}$$

But note that the answer is not unique, here are some other NNFs for the same formula:

$$\begin{aligned} &\neg p \vee \neg s \vee t \vee (\neg q \wedge r) \\ &(\neg p \vee \neg s \vee t \vee \neg q) \wedge (\neg p \vee \neg s \vee t \vee r) \end{aligned}$$

Conversion to NNF is an example of applying a rewrite system: replace the LHS by the RHS of a substitution rule, over and over, until a fixed point is reached. Note that this requires pattern matching: we have to find the handles in the given expression. The effect of the conversion rules is easy to visualize if we think of the formulae as given by their parse trees: we push negations down to the leaves. Conversion to NNF is often a useful preprocessing step. However, for many algorithms we need more restrictive kinds of normal forms.

Disjunctive Normal Form

Definition 7.3.1 *A minterm is a conjunction of literals and a maxterm is a disjunction of literals.*

Given some fixed variables $x_1, x_2, \ldots, x_n$, a full minterm is a minterm of length n that involves either x_i or $\neg x_i$ for each i.

Every truth assignment σ over $x_1, x_2, \ldots, x_n$ can be associated with a corresponding full minterm $z_1 \ldots z_n$: $z_i = x_i$ if $[\![x_i]\!]_\sigma = 1$ and $z_i = \neg x_i$ otherwise. A truth table can then be construed as a listing of all possible 2^n full minterms, along with the corresponding truth values of the formula $\varphi(x_1, \ldots, x_n)$. By forming a disjunction of the minterms for which the formula is true, we get a normal form representation of the formula.

Definition 7.3.2 *A formula is in Disjunctive Normal Form (DNF) if it is a disjunction of minterms (conjunctions of literals).*

In other words, a DNF formula is a “sum of products” and looks like so:

$$(x_{11} \wedge x_{12} \wedge \ldots \wedge x_{1n_1}) \vee (x_{21} \wedge \ldots \wedge x_{2n_2}) \vee \ldots \vee (x_{m1} \wedge \ldots \wedge x_{mn_m})$$

where each x_{ij} is a literal. In short we write $\bigvee_i \bigwedge_j x_{ij}$. If you think of the formula as a circuit DNF means that there are two layers: an OR gate on top, AND gates below (assuming unbounded fan-in and disregarding negation).

Theorem 7.3.2 *For every formula, there is an equivalent formula in DNF.*

Proof. We may assume that the formula is already in NNF. Then we use the rewrite rules

$$\begin{aligned} \varphi \wedge (\psi_1 \vee \psi_2) &\leadsto (\varphi \wedge \psi_1) \vee (\varphi \wedge \psi_2) \\ (\psi_1 \vee \psi_2) \wedge \varphi &\leadsto (\psi_1 \wedge \varphi) \vee (\psi_2 \wedge \varphi) \end{aligned}$$

By the equivalent substitution lemma 7.2.4 we obtain an equivalent formula that is in DNF. □

Here is an example of conversion. First we generate NNF.

$$\begin{aligned} &\neg(p \vee (q \wedge \neg r) \vee s \vee \neg t \\ &\neg p \wedge \neg(q \wedge \neg r) \wedge \neg s \wedge t \\ &\neg p \wedge (\neg q \vee r) \wedge \neg s \wedge t \end{aligned}$$

Reorder, and then distribute $\wedge$ over $\vee$.

$$\begin{aligned} &\neg p \wedge \neg s \wedge t \wedge (\neg q \vee r) \\ &(\neg p \wedge \neg s \wedge t \wedge \neg q) \vee (\neg p \wedge \neg s \wedge t \wedge r) \end{aligned}$$

The rewrite rules for NNF and DNF may seem quite similar, but note that in DNF conversion we created a second copy of φ (though in an actual algorithm we can avoid duplication by sharing subexpressions; nonetheless the ultimate output will be large). More precisely, the size of the formula in NNF is linear in the size of the input, but the size of the formula in DNF is not polynomially bounded by the size of the input. Thus, even if we have a perfect linear time implementation of the rewrite process, we still wind up with an exponential algorithm.

The formula resulting from extracting full minterms from a truth table has $O(2^n)$ conjunctions of n literals each. Unfortunately, this approach may not produce optimal results: for $p \vee q$, a formula already in DNF, we get

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

So brute force application of this method yields 3 full minterms: $p \wedge \neg q$, $\neg p \wedge q$ and $p \wedge q$. Clearly, we need some simplification process. More about this later in the discussion of resolution. This is essentially the same method we used to get the expressions for sum and carry in the 2-bit adder. In a Boolean algebra, one talks about sums of products of literals instead of DNF. Hence, any Boolean expression can be written in sum-of-products form.

Conjunctive Normal Form (CNF)

We now introduce the dual of disjunctive normal form.

Definition 7.3.3 *A formula is in Conjunctive Normal Form (CNF) if it is a conjunction of maxterms (disjunctions of literals). It is customary to refer to these disjunctions as clauses and write them in set notation* $\{x_1, x_2, \ldots, x_k\}$.

By standard abuse of notation, we can write a formula in CNF as a product-of-sums:

$$\bigwedge_i \bigvee_j x_{ij}.$$

Theorem 7.3.3 *For every formula, there is an equivalent formula in CNF.*

Proof. Again start with NNF, but now use the rules

$$\begin{aligned} \varphi \lor (\psi_1 \land \psi_2) &\rightsquigarrow (\varphi \lor \psi_1) \land (\varphi \lor \psi_2) \\ (\psi_1 \land \psi_2) \lor \varphi &\rightsquigarrow (\psi_1 \lor \varphi) \land (\psi_2 \lor \varphi) \end{aligned}$$

□

The main interest in these normal forms comes from the fact that certain decision algorithms require the given input formula to be given in this particular form. For example, the Davis-Putnam satisfiability testing algorithm requires input in CNF. But note that conversion may be computationally expensive, even if we already have some other normal form. For example, the formula

$$\varphi = (p_{10} \land p_{11}) \lor (p_{20} \land p_{21}) \lor \ldots (p_{n0} \land p_{n1})$$

is in DNF, but conversion to CNF using our rewrite rule produces the exponentially larger formula

$$\varphi \equiv \bigwedge_{f:[n]\to\mathbf{2}} \bigvee_{i\in[n]} p_{if(i)}$$

This may seem like a serious obstacle towards practical CNF-based algorithms. However, there is a beautiful solution to the problem, we just have to be careful with the conversion process. Instead of preserving the underlying set of propositional variables, which may appear to be the obvious approach, we extend it by a new variable q_ψ for each subformula ψ of ϕ. For each propositional variable p we let $q_p = p$ and introduce a clause $\{p\}$. Otherwise we introduce clauses as follows:

$$\begin{aligned} q_{\neg\psi} &: \{q_\psi, q_{\neg\psi}\}, \{\neg q_\psi, \neg q_{\neg\psi}\} \\ q_{\psi\lor\varphi} &: \{\neg q_\psi, q_{\psi\lor\varphi}\}, \{\neg q_\varphi, q_{\psi\lor\varphi}\}, \{\neg q_{\psi\lor\varphi}, q_\varphi, q_\psi\} \\ q_{\psi\land\varphi} &: \{q_\psi, \neg q_{\psi\land\varphi}\}, \{q_\varphi, \neg q_{\psi\land\varphi}\}, \{\neg q_\psi, \neg q_\varphi, q_{\psi\land\varphi}\} \end{aligned}$$

The intended meaning of q_ψ is pinned down by these clauses, e.g. $q_{\psi\lor\varphi} \equiv q_\psi \lor q_\varphi$. This transformation is due to G. S. Tseitin.

Theorem 7.3.4 *The set C of clauses above is satisfiable if, and only if, its Tseitin transform ϕ is so satisfiable. Moreover, C can be constructed in time linear in the size of ϕ.*

Proof. From left to right, suppose that $\sigma \models C$. An easy induction shows that for any subformula ψ we have $[\![\psi]\!]_\sigma = [\![q_\psi]\!]_\sigma$. Hence $[\![\phi]\!]_\sigma = [\![q_\phi]\!]_\sigma = 1$ since $\{q_\phi\}$ is a clause in C.

For the opposite direction assume that $\sigma \models \phi$. Define a new valuation τ by $\tau(q_\psi) = [\![\psi]\!]_\sigma$ for all subformulae ψ. It is easy to check that $\tau \models C$. □

As an example of this transformation, consider again the formula

$$\varphi = (p_{10} \land p_{11}) \lor (p_{20} \land p_{21}) \lor \ldots \lor (p_{n0} \land p_{n1})$$

Set $B_k = (p_{k0} \land p_{k1})$ and $A_k = B_k \lor B_{k+1} \lor \ldots \lor B_n$ for $k = 1, \ldots, n$. Thus, $\varphi = A_1$ and all the subformulae other than variables are of the form A_k or B_k. The clauses in the Tseitin form of φ are as follows (we ignore the variables):

- q_{A_k}: $\{q_{B_k}, \neg q_{B_k \land A_{k-1}}\}, \{q_{A_{k-1}}, \neg q_{B_k \land A_{k-1}}\}, \{\neg q_{B_k}, \neg q_{A_{k-1}}, q_{B_k \land A_{k-1}}\}$
- q_{B_k}: $\{\neg p_{k0}, q_{B_k}\}, \{\neg p_{k1}, q_{B_k}\}, \{\neg q_{B_k}, p_{k1}, p_{k0}\}$

It should be noted that formulae in CNF and DNF are not particularly useful for a human being when it comes to understanding the meaning of a formula (NNF is not quite as bad). But that is not their purpose: they provide a handle for specialized algorithms, for instance, to test validity or satisfiability.

7.3.2 Satisfiability Testing

How could we go about trying to ascertain satisfiability of a formula, say, in conjunctive normal form?

There is an old, but surprisingly powerful satisfiability testing algorithm due to Davis and Putnam, originally published in 1960. A variant was first implemented by Davis, Longman and Loveland in 1962 on an IBM 704. Modern versions of the DPLL algorithm are still widely used today. Input for the algorithm is required to be in CNF.

SAT Solvers

Algorithms that check whether a propositional formula is satisfiable are often referred to as SAT solvers. By Tseitin's transformation, we may as well assume that the formula is given in CNF. In fact, we may safely assume that no clause contains a literal and its negation.

Davis/Putnam Algorithm

Suppose a formula φ is given in CNF and we are trying to determine satisfiability. Say, the formula might be written as

$$\{x, \overline{y}, u\}, \{\overline{x}, y, u\}, \{x, \overline{u}\}$$

A clause is a unit clause if it contains just one literal. Note that an empty clause corresponds to $\bot$: there are no literals that one could try to set true. Unit clauses are easy to deal with: The only way to satisfy a single literal x is by setting $\sigma(x) = 1$. Note that once we decide $\sigma(x) = 1$, we can perform

- Unit Subsumption: delete all clauses containing x, and
- Unit Resolution: remove $\overline{x}$ from all remaining clauses.

This process is called unit clause elimination. The crucial point is: a CNF formula φ containing unit clause $\{x\}$ is satisfiable iff there is an assignment σ setting x to true, and satisfying φ' obtained from φ by UCE.

A pure literal in a CNF formula is a literal that occurs only directly, but not negated. So the formula may contain x but not $\overline{x}$ or conversely only $\overline{x}$ but not x. In this case we can simply set $\sigma(z) = 1$ and remove all the occurrences of the literal. To do this efficiently, we should keep a counter for the total number of occurrences of both x and $\overline{x}$ for each variable.

- Perform unit clause elimination until no unit clauses are left. Then perform pure literal elimination, call the result ψ.
- If an empty clause has appeared, return false.
- If all clauses have been eliminated, return true.
- Splitting: otherwise, pick one of the remaining literals, x. Recursively test **both**

  $$\psi, \{x\} \qquad \text{and} \qquad \psi, \{\overline{x}\}$$

  for satisfiability. Return true if at least one of the branches returns true, false otherwise.

In a sense, DPLL is dangerously close to brute-force search. The algorithm still succeeds beautifully in real world applications since it systematically prunes irrelevant parts of the search tree.

Here is an example that implement a CNF formula as a list of lists of literals. After three unit clause elimination steps (no pure literal elimination) and one split on d we get an empty clause list, and thus the answer Yes.

$((a,b,c),(a,!b),(a,!c),(c,b),(!a,d,e),(!b))$	unit clause $!b$
$((a,c),(a,!c),(c),(!a,d,e))$	unit clause c
$((a),(!a,d,e))$	unit clause a
$((d,e))$	split d
$()$	satisfiable

Note that this algorithm implicitly also solves the search problem: we only need to keep track of the assignments made to literals. For the example, the corresponding assignment is $\sigma(b) = 0$, $\sigma(c) = \sigma(a) = \sigma(d) = 1$. The choice for e does not matter.

Lemma 7.3.1 *The Davis/Putnam algorithm is correct: it returns true if, and only if, the input formula is satisfiable.*

Proof. Suppose φ is in CNF and has a unit clause $\{x\}$. Then φ is satisfiable iff there is a satisfying truth assignment σ such that $\sigma(x) = 1$. But then $\sigma(C) = 1$ for any clause containing x, so Unit Subsumption does not affect satisfiability. Also, $\sigma(C) = \sigma(C')$ for any clause containing $\overline{x}$, where C' denotes the removal of literal $\overline{x}$. Hence Unit Resolution does not affect satisfiability either.

Suppose z is a pure literal. If σ satisfies φ then σ' also satisfies φ where

$$\sigma(u) = \begin{cases} \sigma(u) & \text{if } u = z, \\ 1 & \text{otherwise.} \end{cases}$$

Let x be any literal in φ. Then trivially $\varphi \equiv (x \wedge \varphi[1/x]) \vee (\neg x \wedge \varphi[0/x])$. But splitting checks exactly the two formulae on the right for satisfiability; hence φ is satisfiable if, and only if, one of the two branches returns true. Termination is obvious. □

Note that in the splitting step there usually there are many choices for x. This provides an opportunity to use clever heuristics to speed things up. One plausible strategy is to pick the most frequently occurring literal. Still, there is no guarantee that exponential running time can be avoided. In practice, though, Davis/Putnam is usually quite fast. It is not entirely understood why formulae that appear in real-world problems tend to produce only polynomial running time when tackled by Davis/Putnam.

7.3.3 Refutation

A carefully designed implementation of the Davis/Putnam algorithm is a practical and powerful method to tackle fairly large instances of Satisfiability, dealing with thousands of variables. The main idea in Davis/Putnam is to search for a satisfying truth-assignment in a way that often circumvents the potentially exponential blow-up. How about the opposite approach: trying to show there cannot be satisfying truth-assignment? The procedure should be usually fast, but on occasion may blow-up exponentially. As with Davis/Putnam, we assume the formula is given in CNF and think of Γ as a set of clauses. Let x be a variable and define

- Γ_x^+: the clauses of Γ that contain x positively,
- Γ_x^-: the clauses of Γ that contain x negatively, and
- Γ_x^*: the clauses of Γ that are free of x.

So $\Gamma = \Gamma_x^+ \cup \Gamma_x^- \cup \Gamma_x^*$. This partition gives rise to a trie data structure that is also useful in the implementation of Davis/Putnam: it helps to speed up the fundamental operations in this algorithm.

Proposition 7.3.1 *If Γ_x^+ or Γ_x^- is empty, one can replace Γ by Γ_x^*.*

More precisely, in this case Γ is a contradiction if, and only if, Γ_x^* is a contradiction. This is very much the same idea as pure literals elimination in DPLL. Since Γ_x^* is smaller than Γ (unless x does not appear at all) this step makes the problem a little easier. Of course, we get stuck with this simplification when all variables have positive and negative occurrences. Suppose x is a variable that appears in clause C and appears negated in clause C':

$$C = \{x, y_1, \ldots, y_k\} \qquad C' = \{\overline{x}, z_1, \ldots, z_l\}$$

Then we can introduce an new clause, a resolvent of C and C' by

$$D = \{y_1, \ldots, y_k, z_1, \ldots, z_l\}$$

Proposition 7.3.2 $\{C, C'\} \equiv \{C, C', D\}$

We write $\mathsf{Res}(C, C') = D$, but note that there may be several resolvents. The formula $\varphi = \{\{x, \overline{y}, \overline{z}\}, \{\overline{x}, \overline{y}\}, \{\overline{y}, z\}, \{y\}\}$ admits the following steps:

$$\begin{aligned} \mathsf{Res}(\{x, \overline{y}, \overline{z}\}, \{\overline{x}, \overline{y}\}) &= \{\overline{y}, \overline{z}\} \\ \mathsf{Res}(\{\overline{y}, z\}, \{y\}) &= \{z\} \\ \mathsf{Res}(\{\overline{y}, \overline{z}\}, \{z\}) &= \{\overline{y}\} \\ \mathsf{Res}(\{y\}, \{\overline{y}\}) &= \emptyset \end{aligned}$$

Note that we have used intermediate results in later steps. This process is called resolution.

The last resolvent is the empty clause. It is customary in the context of resolution methods to write the empty clause thus: □. Yes, this collides with our end-of-proof symbol. As we will see shortly, □ is a resolvent of Γ if, and only if, the formula is a contradiction. More precisely, given a collection of clauses Γ, let $\mathsf{Res}(\Gamma)$ be the collection of all resolvents of clauses in Γ plus Γ itself. Set

$$\mathsf{Res}^\star(\Gamma) = \bigcup_n \mathsf{Res}^n(\Gamma)$$

so $\mathsf{Res}^\star(\Gamma)$ is the least fixed point of the resolvent operator applied to Γ. We will show that

Lemma 7.3.2 *Γ is a contradiction if, and only if, $\Box \in \mathsf{Res}^\star(\Gamma)$.*

One often speaks of a resolution proof for the un-satisfiability of Γ as a directed, acyclic graph G whose nodes are clauses. The following degree conditions hold:

- The clauses of Γ have indegree 0.
- Each other node has indegree 2 and corresponds to a resolvent of the two predecessors.
- There is one node with outdegree 0 corresponding to the empty clause.

The graph is in general not a tree, just a DAG, since nodes may have outdegrees larger than 1 (a single clause can be used together with several others to produce resolvents). Other than that, you can think of G as a tree with the clauses of Γ at the leaves, and $\Box$ at the root. There are two issues we have to address:

- Correctness: if a formula has $\Box$ as a resolvent then it is indeed a contradiction.
- Completeness: if a formula is a contradiction then it has $\Box$ as a resolvent.

Note that for a practical algorithm the last condition is actually a bit too weak: there are many possible ways to construct a resolution proof, since we do not know ahead of time which method will succeed we need some kind of robustness: it should not matter too much which clauses we resolve first. On the upside, note that we do not necessarily have to compute all of $\mathsf{Res}^\star(\Gamma)$: if $\Box$ pops up we can immediately terminate.

Lemma 7.3.3 *For any truth-assignment σ and clauses $C = (x, \boldsymbol{y})$ and $C' = (\overline{x}, \boldsymbol{y})$ we have*

$$\sigma(C) = \sigma(C') = 1 \quad \textit{implies} \quad \sigma(\mathsf{Res}(C, C')) = 1$$

Proof. If $\sigma(y_i) = 1$ for some i we are done, so suppose $\sigma(y_i) = 0$ for all i. Since σ satisfies C we must have $\sigma(x) = 1$. But then $\sigma(\overline{x}) = 0$ and thus $\sigma(z_i) = 1$ for some i. Hence σ satisfies $\mathsf{Res}(C, C')$. □

It follows by induction that if σ satisfies Γ it satisfies all resolvents of Γ. Hence resolution is correct.

Theorem 7.3.5 *Resolution is complete.*

Proof. By induction on the number n of variables. We have to show that if Γ is a contradiction then $\Box \in \mathsf{Res}^\star(\Gamma)$.

$n = 1$: Then $\Gamma = \{x\}, \{\overline{x}\}$. In one resolution step we obtain $\Box$. Done.

Assume $n > 1$ and let x be a variable. Let Γ_0 and Γ_1 be obtained by performing unit clause elimination for $\{x\}$ and $\{\overline{x}\}$. Note that both Γ_0 and Γ_1 must be contradictions. Hence by IH $\Box \in \mathsf{Res}^\star(\Gamma_i)$. Now the crucial step: by repeating the "same" resolution proof with Γ rather than Γ_i, $i = 0, 1$, we get $\Box \in \mathsf{Res}^\star(\Gamma)$ if this proof does not use any of the mutilated clauses. Otherwise, if mutilated clauses are used in both cases, we must have

- $\{x\} \in \mathsf{Res}^\star(\Gamma)$ from Γ_1, and
- $\{\overline{x}\} \in \mathsf{Res}^\star(\Gamma)$ from Γ_0.

Hence $\Box \in \mathsf{Res}^\star(\Gamma)$. □

It is clear that we would like to keep the number of resolvents introduced in the resolution process small. Let's say that clause ψ subsumes clause φ if $\psi \subseteq \varphi$: ψ is at least as hard to satisfy as φ. We keep a collection of "used" clauses U which is originally empty. The algorithm ends when C is empty.

- Pick a clause ψ in C and move it to U.
- Add all resolvents of ψ and U to C except that:
  - Tautology elimination: delete all tautologies.
  - Forward subsumption: delete all resolvents that are subsumed by a clause.
  - Backward subsumption: delete all clauses that are subsumed by a resolvent.

So how large is a resolution proof, even one that uses the subsumption mechanism? What is a problem that is particularly difficult for resolution? There is a Boolean formula $\mathsf{EO}_k(x_1,\dots,x_k)$ of size $\Theta(k^2)$ such that σ satisfies $\mathsf{EO}_k(x_1,\dots,x_k)$ iff σ makes exactly one of the variables $x_1,\dots,x_k$ true.

$$\mathsf{EO}_k(x_1,\dots,x_k) = (x_1 \vee x_2 \ldots \vee x_k) \wedge \bigwedge_{1\le i<j\le k} \neg(x_i \wedge x_j).$$

Using these formulae we can express the Dirichletsche Schubfachprinzip, aka as Pigeonhole Principle, in propositional logic.

Lemma 7.3.4 *There is no injective function from $[n+1]$ to $[n]$.*

This sounds hopelessly obvious, but the pigeonhole principle is in fact an important combinatorial principle that is used in many places; the standard proof is by induction on n. We can translate the pigeonhole principle into a Boolean formula as follows: the idea is that variable x_{ij} is true iff pigeon i sits in hole j. So we have variables x_{ij} where $1 \le i \le m$ and $1 \le j \le n$. Let

$$\Gamma_{mn} = \bigwedge_{i\le m} \mathsf{EO}_n(x_{i1}, x_{i2}, \dots, x_{in})$$

Then Γ_{mn} is satisfiable if, and only if, $m \le n$. In particular we ought to be able to use resolution to show that $\Gamma_{n+1,n}$ is a contradiction. By completeness there must be a resolution proof showing that $\Gamma_{n+1,n}$ is a contradiction. As it turns out no such proof can be short.

Theorem 7.3.6 *Every resolution proof for the contradiction $\Gamma_{n+1,n}$ has exponential length.*

The proof of the theorem is too lengthy to be included here.

Horn Formulae

One might wonder if there is perhaps a special class of formulae where a resolution type approach is always fast. We can think of a clause $\{\overline{x}_1, \overline{x}_2, \dots, \overline{x}_r, y_1, y_2, \dots, y_s\}$ as an implication: $x_1 \wedge x_2 \wedge \ldots \wedge x_r \Rightarrow y_1 \vee y_2 \vee \ldots \vee y_s$. When $s = 1$ these implication are particularly simple.

Definition 7.3.4 *A formula is a Horn formula if it is in CNF and every clause contains at most one unnegated variable.*

For example, $\varphi = \{\overline{x}, \overline{y}, z\}, \{\overline{y}, \overline{z}\}, \{x\}$ can be written equivalently as $\varphi = x \wedge y \Rightarrow z, y \wedge z \Rightarrow \bot, x$. In other words, a Horn formula has only Horn clauses, and a Horn clause is essentially an implication of the form

$$C = x_1 \wedge x_2 \wedge \ldots \wedge x_n \Rightarrow y$$

where we allow $y = \bot$. We also allow single unnegated variables (if you like: $\top \Rightarrow x$). Note that if we have unit clauses x_i then a resolvent of these and C will be y. This gives rise to the following satisfiability testing algorithm.

- Mark all variables x in unit clauses $\{x\}$.
- If there is a clause $x_1 \wedge x_2 \wedge \ldots \wedge x_n \Rightarrow y$ such that all the x_i are marked, mark y. Repeat till a fixed point is reached.
- If $\perp$ is ever marked, return `No`.
- Otherwise, return `Yes`.

You can also think of this as a graph exploration algorithm: node y is marked only if all predecessor nodes x_i are already marked (careful though, y can be the RHS of several clauses.) The algorithm is linear in the size of Γ. We can read off a satisfying truth-assignment if the formula is satisfiable:

$$\sigma(x) = \begin{cases} 1 & x \text{ is marked,} \\ 0 & \text{otherwise.} \end{cases}$$

Then $\sigma(\Gamma) = 1$. Moreover, $\tau(\Gamma) = 1$ implies that $\forall\, x\, (\tau(x) \leq \sigma(x))$so σ is the "smallest" satisfying truth-assignment. Logic programming languages such as PROLOG rely heavily on Horn clauses since the more general case is prohibitively complex.

Exercises

Exercise 7.3.1 Discuss the usefulness, or lack thereof, of the positive/negative part method for formulae in CNF and DNF.

Exercise 7.3.2 Suppose we have a resolution proof for some contradiction Γ. For any truth-assignment σ there is a uniquely determined path from a clause of Γ to the root $\Box$ such that for any clause C along that path we have: $\sigma(C) = 0$.

Exercise 7.3.3 Prove that the method in theorem 7.3.2 really produces DNF. What is the complexity of this algorithm?

Exercise 7.3.4 Construct an example where conversion to DNF causes exponential blow-up.

Exercise 7.3.5 Construct a formula Γ_n that is satisfiable if, and only if, the $n \times n$ chess-board has the property that a knight can reach all squares by a sequence of admissible moves. What would your formula look like in CNF and DNF?

Exercise 7.3.6 Construct a formula Γ_n that is satisfiable if, and only if, the $n \times n$ chess-board admits a knight's tour: a sequence of admissible moves that touches each square exactly once. Again, what would your formula look like in CNF and DNF?

Exercise 7.3.7 What is the running time of the algorithm that converts to CNF?

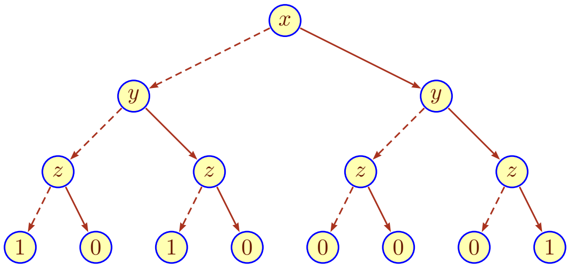
Exercise 7.3.8 Show how to convert directly between DNF and CNF.

Exercise 7.3.9 How hard is it to check whether a formula in CNF is a tautology? How hard is it to check whether a formula in DNF is a tautology? How about checking whether a formula in DNF (or CNF) is a contradiction?

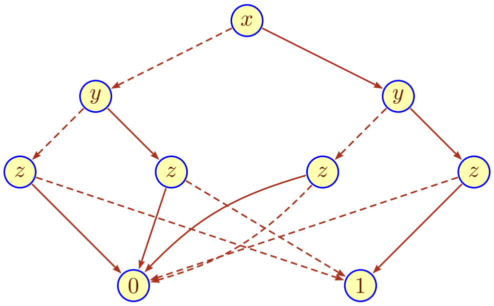
7.3.4 Binary Decision Diagrams

State-of-the-art satisfiability testing algorithms all use clever logical structure to avoid exponential blow-up, the underlying data structures are relatively straightforward. The question arises whether satisfiability could also be tested with a method that instead relies heavily on a suitable data structure.

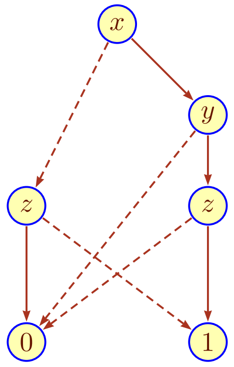
To see what such a structure might look like, consider the ternary Boolean function $f = (x + \overline{z})(\overline{x} + y)(\overline{x} + \overline{y} + z)$. We can represent f naturally as a complete binary tree:



The nodes are labeled by the Boolean variables, and the edges indicate the truth assignment for these variables. The leaves are labeled with the values for f under the corresponding assignment. We can reduce the size of the tree by sharing leaves with the same label:



By continuing this merging process, and removing nodes whose out-edges lead to the same target, we ultimately wind up with



Fix a set $\mathsf{Var} = \{x_1, x_2, \ldots, x_n\}$ of n Boolean variables.

Definition 7.3.5 *A binary decision diagram (BDD) (over* Var*) is a rooted, directed acyclic graph with two terminal nodes (out-degree* 0*) and interior nodes of out-degree 2. The interior nodes are labeled in* Var*.*

We write $\mathsf{var}(u)$ for the node labels. The two successors of an interior node are traditionally referred to as $\mathsf{lo}(u)$ and $\mathsf{hi}(v)$, corresponding to the false and true branches. The terminal nodes are labeled by constants 0 and 1, indicating values false and true.

It is desirable to access the variables in the same order when traversing the graph from root to leaves. To this end, fix an ordering $x_1 < x_2 < \ldots < x_n$ on Var. For simplicity assume that $x_i < 0, 1$.

Definition 7.3.6 *A BDD is ordered (OBDD) if the label sequence along any path is ordered.*

Thus $\mathsf{var}(u) < \mathsf{var}(\mathsf{lo}(u)), \mathsf{var}(\mathsf{hi}(u))$ for any interior node u in a OBDD.

Definition 7.3.7 *A OBDD is reduced (ROBDD) if it satisfies the following two conditions:*

- *Uniqueness: there are no nodes* $u \neq v$ *such that* $\mathsf{var}(u) = \mathsf{var}(v), \mathsf{lo}(u) = \mathsf{lo}(v), \mathsf{hi}(u) = \mathsf{hi}(v)$
- *Non-Redundancy: for all nodes* u*:* $\mathsf{lo}(u) \neq \mathsf{hi}(u)$

The uniqueness condition corresponds to shared subexpressions: we could merge u and v. Non-redundancy corresponds to taking shortcuts: we can skip ahead to the next test. Since ROBDDs are by far the most important type, we simply refer to them as BDD.

One can think of BDDs as a special kind of normal form, if-then-else normal form (INF): the only allowed operations are if-then-else and constants. Moreover, tests are performed only on variables (not compound expressions). By if-then-else we mean the ternary Boolean function

$$\mathsf{ite}(x, y_1, y_0) = x\, y_1 + \overline{x}\, y_0$$

In the corresponding INF, the variables are always ordered in the sense that

$$\mathsf{ite}(x, \mathsf{ite}(y, f_1, f_2), \mathsf{ite}(z, f_3, f_4)) \text{ implies } x < y, z$$

It is easy to express the standard Boolean operations in terms of if-then-else:

$$\begin{aligned}
\neg x &= \mathsf{ite}(x, 0, 1)\\
x \wedge y &= \mathsf{ite}(x, y, 0)\\
x \vee y &= \mathsf{ite}(x, 1, y)\\
x \Rightarrow y &= \mathsf{ite}(x, y, 1)\\
x \oplus y &= \mathsf{ite}(x, \mathsf{ite}(y, 0, 1), y)
\end{aligned}$$

Boole-Shannon Expansion

We need one more idea: unfolding a Boolean function into simpler ones by fixing the value of one of the variables. Suppose f is a Boolean function with variable x. Define the cofactors of f by

$$\begin{aligned}
f_x(\boldsymbol{u}, x, \boldsymbol{v}) &= f(\boldsymbol{u}, 1, \boldsymbol{v})\\
f_{\overline{x}}(\boldsymbol{u}, x, \boldsymbol{v}) &= f(\boldsymbol{u}, 0, \boldsymbol{v})
\end{aligned}$$

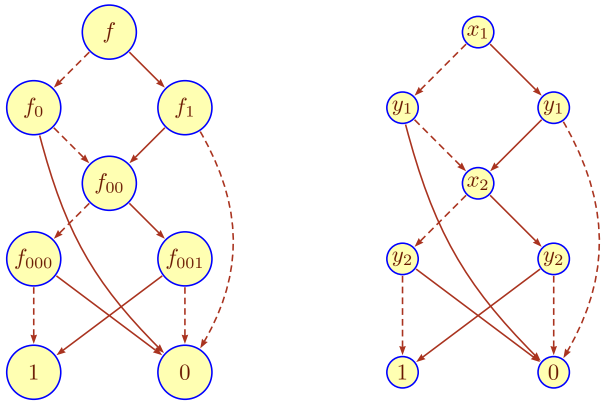
Note that f_x does not depend on x. Also $f_{xy} = f_{yx}$, so the order in which we perform the decomposition does not matter. Clearly,

$$f = \overline{x}\, f_{\overline{x}} + x\, f_x = \mathsf{ite}(x, f_x, f_{\overline{x}}).$$

For example, for $f \equiv (x_1 = y_1) \wedge (x_2 = y_2)$, and sharing common subexpressions, we get

$$\begin{aligned}
f &= \mathsf{ite}(x_1, f_1, f_0)\\
f_0 &= \mathsf{ite}(y_1, 0, f_{00})\\
f_1 &= \mathsf{ite}(y_1, f_{00}, 0)\\
f_{00} &= \mathsf{ite}(x_2, f_{001}, f_{000})\\
f_{000} &= \mathsf{ite}(y_2, 0, 1)\\
f_{001} &= \mathsf{ite}(y_2, 1, 0)
\end{aligned}$$

corresponding to the BDD, labeled once by cofactors, and once by variables:



By a straightforward induction we can associate any BDD with root u with a Boolean function F_u:

$$\begin{aligned}
&F_0 = \mathbf{0} \qquad F_1 = \mathbf{1}\\
&F_u = \mathsf{ite}(\mathsf{var}(u), F_{\mathsf{lo}(u)}, F_{\mathsf{hi}(u)})
\end{aligned}$$

If the BDDs under consideration are also ordered and reduced, we get a useful representation.

Theorem 7.3.7 (Canonical Form Theorem) *For every Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ there is exactly one ROBDD u such that $f = F_u$.*

Proof. The claim is clear for $n = 0$, so consider a function $f : \mathbb{B}^n \to \mathbb{B}$, written $f(x, \boldsymbol{y})$.

By IH we may assume that the BDDs for $f(0, \boldsymbol{y})$ and $f(1, \boldsymbol{y})$ are unique, say, with roots u and v, respectively. If $f(0, \boldsymbol{y}) = f(1, \boldsymbol{y})$ this is also the BDD for f. Otherwise introduce a new node w and set

$$\lambda(w) = x \qquad \mathsf{lo}(w) = u \qquad \mathsf{hi}(w) = v$$

The resulting BDD represents f and is unique. □

Constructing a BDD from an exponential size decision diagram as above is of little interest, one needs to construct the graph directly from the cofactors. Of course, there are a number of natural operations on BDDs.

- Reduce: Turn a decision tree into a BDD.
- Apply: Given two BDDs u and v and a Boolean operation $\diamond$, determine the BDD for $F_u \diamond F_v$, and likewise for negation.
- Restrict: Given a BDD u and a variable x, determine the BDD for $F_u[a/x]$.

Again, the purpose of Reduce is not to convert a raw decision tree, but to clean up after other operations. All Apply operations are based on the fact that Boolean operations coexist peacefully with cofactors:

Proposition 7.3.3 $\overline{f}_x = \overline{f_x}$, $(f + g)_x = f_x + g_x$, $(fg)_x = f_x g_x$

Negation deserves a special mention: the BDD for $\overline{f}$ differs from the BDD for f only in the terminal nodes, which are swapped. This can be exploited to make complementation constant time: we add a "negation bit" to the pointers leading to $\mathsf{lo}(v)$ and $\mathsf{hi}(v)$. In this setting, one can get away with a single terminal node, say, $\mathbf{1}$. Since all paths end at $\mathbf{1}$, evaluation comes down to counting the number of complement edges (modulo 2). However, a little care is needed to preserve canonicity. We have

$$f = xf_x + \overline{x}f_{\overline{x}} = \overline{(x\overline{f}_x + \overline{x}\overline{f}_{\overline{x}})}$$

so there are two ways to represent f. We adopt the convention that the *then* branch is chosen to be a plain pointer, without a complement bit.

For binary operations, one can use top-down recursion. To apply a Boolean operation to F_u and F_v then requires (expected) time $O(|u||v|)$, and is often faster.

Once we have constructed a BDD for a Boolean function f it is trivial to check if f is a tautology: the test is for $f = \mathbf{1}$ and is constant time. Likewise, it is trivial to check if f is satisfiable: we need $f \neq \mathbf{0}$. In fact, we can count satisfying truth assignments in $O(|u|)$: they correspond to the total number of paths (including potentially phantom variables) from the root to terminal 1. We can even construct a BDD s for all satisfying truth assignments in time $O(n|s|)$ where $|s| = O(2^{|u|})$.

Alas, there is one fundamentals problem with this approach: BDDs may have size exponential in the number of variables of the function. Just like Shannon's bound on the size of circuits, this follows from basic information theoretic considerations. Concretely, it turns out that addition of k-bit natural numbers can be expressed nicely in a linear size BDD. Alas, multiplication causes problems.

Lemma 7.3.5 *BDDs for multiplication require exponential size.*

Another major issue is the underlying variable ordering. Consider the pairwise equality function

$$f \equiv (x_1 = y_1)(x_2 = y_2) \dots (x_k = y_k)$$

on $2k$ variables. It is easy to see that the ordering $x_1, y_1, x_2, y_2, \dots, y_k$ leads to a linear size BDD, whereas the ordering $x_1, x_2, \dots, x_k, y_1, y_2, \dots, y_k$ leads to exponential blow-up. Unfortunately, the problem of determining an optimal variable ordering for a given function is computationally hard.

Exercises

Exercise 7.3.10 Compute the Tseitin transform of $\varphi \equiv \bigwedge_{f:[n]\to\mathbf{2}} \bigvee_{i\in[n]} p_{if(i)}$. By contrast, show that there is no small CNF for φ: the 2^n disjunctions of length n must all appear.

Exercise 7.3.11 For symmetric Boolean functions variable ordering does not matter. Determine the size of the BDD for the following functions: conjunctions, disjunctions, parity (xor), threshold functions.

Exercise 7.3.12 Implement a SAT solver along the lines of DPLL in C or C++.

Exercise 7.3.13 Implement a refutation solver in C or C++. How does your program compare to the SAT solver from the last exercise?

Exercise 7.3.14 Implement binary decision diagrams in C or C++. How does your program compare to the SAT solver from a previous exercise?

7.4 Deduction Systems

7.4.1 Deduction Systems for Propositional Logic

We cannot determine efficiently whether a given formula is a tautology, at least not without some major upheaval but perhaps we can still generate all tautologies in some systematic way? Our goal is to try to understand reasoning, so why not model an essential aspect of reasoning: conclusions being derived from premise. There are two components to such a system:

- Axioms: Unchallenged assumptions that we simply believe to be true, at least in some particular context or other.
- Rules of Inference: Rules that allow us to form assertions from other ones.

The key point here is that we rely only on the syntax of the formulae, not their behavior under truth valuations. Of course, in the context of propositional logic, the axioms should all be tautologies, and the rules of inference should represent methods of deduction that preserve tautologies. There are many ways one can go about this.

A Hilbert-Style System

We start a type of deduction system first championed by D. Hilbert. Hilbert-style systems have one major advantage: they have a particularly simple structure and are somewhat easier to understand than alternative approaches. A proof in a Hilbert-style system is just a sequence of formulae and it is straightforward to check whether a given sequence indeed constitutes a a valid proof. This type of proof is easy to formalize and is used, for example, in Gödel's famous incompleteness theorem (albeit in the context of predicate logic rather than propositional logic). The substantial draw-back of a Hilbert-style system is that the proofs bear almost no resemblance to an actual mathematical proof, the type that can be found in journals and proceedings. Instead, there is a mind-numbingly long sequence of small, mechanical steps that have little meaning taken individually. The guiding intuition behind the argument is typically lost in an ocean of combinatorial details that are difficult for humans to deal with. Still, the idea that proofs themselves can be construed as mathematical objects, and can be studied using mathematical methods, is absolutely essential. In particular in the presence of modern digital computers some manipulations on proofs can be automated, and one should expect that computer-assisted and even computer-generated proofs will play an important rôle in the future.

For simplicity, the following system $\mathcal{H}$ uses only two connectives: $\neg$ and $\lor$, other connectives could be defined from these using some of the equivalences from above. The system has the advantage that the axioms are exceedingly simple, but there are several rules of inference to contend with.

Logical Axioms:

$$\begin{array}{ll} \text{tertium non datur} & \neg\varphi \lor \varphi \end{array}$$

Rules of Inference:

$$\begin{array}{ll} \text{expansion} & \dfrac{\varphi}{\psi \lor \varphi} \\ \text{contraction} & \dfrac{\varphi \lor \varphi}{\varphi} \\ \text{associativity} & \dfrac{\varphi \lor (\psi \lor \chi)}{(\varphi \lor \psi) \lor \chi} \\ \text{cut rule} & \dfrac{\varphi \lor \psi \qquad \neg\varphi \lor \chi}{\psi \lor \chi} \end{array}$$

For the inference rules we have used the notation

$$\frac{\varphi}{\psi} \quad \text{or} \quad \frac{\varphi_1 \quad \varphi_k}{\psi}$$

to indicate that there is one premise φ or two premises φ_1 and φ_2 and the conclusion is ψ. Strictly speaking, the given axioms are not axioms but axiom schema: we obtain an axiom by replacing φ and ψ by arbitrary propositional formulae.

We can now define a notion of proof to be no more than a sequence of formulae such that each one is either an axiom or justified by an application of a rule of inference to one or more earlier formulae.

Definition 7.4.1 *Let Γ be an arbitrary set of propositional formulae. A derivation or a proof from Γ in $\mathcal{H}$ is a sequence of formulae*

$$\varphi_1, \varphi_2, \varphi_3, \dots, \varphi_n$$

where each φ_i is either an axiom, a formula in Γ or the conclusion of a rule all of whose premise appear earlier on in the sequence.

Standard notation for a proof from given premise is as follows:

$$\underbrace{\psi_1, \psi_2, \dots, \psi_n}_{\text{premises}} \vdash \underbrace{\psi}_{\text{conclusion}}$$

Frege's turnstile is back. If there are no premises at all, one writes $\vdash \psi$.

How does one construct derivations in our Hilbert system? There are two principal heuristics: one is to start from the collection of axioms and premise and try to combine them in appropriate ways using rules of inference; after a number of steps the desired proof goal should appear. The other method is to work backwards from the proof goal and try to determine how it could have arisen as the result of a rule of inference. Of course, we can also mix both approaches and try to work forward and backwards at the same time.

Here is a small example of a proof.

Lemma 7.4.1 $p, \neg p \lor q \;\vdash\; q$

Proof. A complete proof looks like so:

$$p, q \lor p, \neg q \lor q, p \lor q, \neg p \lor q, q \lor q, q$$

□

The last presentation of a proof leaves much to be desired. Notably, there is no indication in the sequence how any of the formulae is justified by earlier ones. Of course, we can search the initial segment of the proof for applicability of a rule that yields the current formula but this requires a bit of effort. To improve legibility, one usually presents annotated proofs where each formula carries some additional information that justifies its appearance in the sequence. Annotated proofs can be displayed nicely in form of a numbered table.

1	p	*prem*
2	$q \lor p$	$e, 1$
3	$\neg q \lor q$	*axiom*
4	$p \lor q$	$cut, 2, 3$
5	$\neg p \lor q$	*prem*
6	$q \lor q$	$cut, 4, 5$
7	q	$c, 6$

The left column numbers the intermediate formulae, and the right column contains the annotations based on these line numbers. Thus, formula number 4, $p \lor q$ was obtained by applying the cut rule to formulae 2 and 3. Even annotated proof tables are not particularly easy to read for humans, it is often preferable to organize a derivation in tree form using the fractional notation from above:

$$\cfrac{\cfrac{\cfrac{p}{q \lor p}\,(e) \qquad \neg q \lor q}{p \lor q}\,(cut) \qquad \neg p \lor q}{\cfrac{q \lor q}{q}\,(c)}\,(cut)$$

The root is the proven formula, and the hypotheses and axioms are at the leaves. The labels indicate the type of rule used, making it easy to check correctness. Here is derivation that shows that our system treats disjunction as a commutative operation.

Lemma 7.4.2 $p \lor q \vdash q \lor p$

Proof.

$$\frac{p \lor q \quad \neg p \lor q}{q \lor p}\ (c)$$

□

We have stated the proofs above using propositional variables. But we could as well have established stronger results by showing that, say, $\varphi \lor \psi \vdash \psi \lor \varphi$ holds for any formula φ and ψ in $\mathsf{Pfml}(V)$. To see this note we can take a derivation for the variable case and simply replace the variables by the corresponding formulae. Because of the structure of our axioms and rules we obtain another valid derivation in this way.

Definition 7.4.2 *A substitution is a map $\theta : V \to \mathsf{Pfml}(V)$.*

It is customary to write the action of substitutions on the right as in $p\theta$. A substitution naturally extends to all of $\mathsf{Pfml}(V)$ by setting

$$\begin{aligned}(\neg\varphi)\theta &= \neg(\varphi\theta)\\ (\varphi \lor \psi)\theta &= \varphi\theta \lor \psi\theta\\ (\varphi \land \psi)\theta &= \varphi\theta \land \psi\theta\\ (\varphi \Rightarrow \psi)\theta &= \varphi\theta \Rightarrow \psi\theta\end{aligned}$$

We apply substitutions to sets or sequences of formulae pointwise.

Definition 7.4.3 *A deduction system is substitutive if $\Gamma \vdash \varphi$ implies $\Gamma\theta \vdash \varphi\theta$ for any substitution θ.*

A substitutive system produces results that are in a sense generic, there is no particular meaning attached to the variables. This is in contrast to situations where we try to describe specific situations by attaching special meaning to some variables. For example, if we want to model the ability of a knight to move on a chessboard we might introduce variables p_{ij} with the intention that p_{ij} is true if the knight can reach square (i, j). We can then describe the movements of the knight by formulae such as

$$p_{44} \Rightarrow p_{23} \lor p_{25} \lor \ldots \lor p_{32}$$

The ability to model various combinatorial problems as propositional formulae is very important, see section 7.3. At any rate, it is easy to check that our deduction system is substitutive.

Proposition 7.4.1 *The Hilbert system $\mathcal{H}$ is substitutive.*

It should be clear from these few examples that formal proofs in $\mathcal{H}$ do not model human reasoning particularly well: they consist of (sometimes long) sequences of small steps, and they are often annoyingly difficult to find, even when it seems intuitively clear that the conclusion follows from the premise. However, while proof search may be difficult, proof checking is easy. Little more that pattern matching needed to check an annotated proof for correctness; a plain proof requires some additional search for the proper handles.

Soundness and Completeness

There are some basic requirements that any reasonable deduction system has to satisfy. First of all, it has to be sound: in our case, we have to make sure that only tautologies can be derived (without premise). Soundness is easy to verify for our Hilbert system by induction on the length of the proof. Second, it is desirable that all valid statement be derivable in the system. In our case we would like every tautology to have a proof in $\mathcal{H}$.

Theorem 7.4.1 *Tautology Theorem*

The deduction system $\mathcal{H}$ is complete: all tautologies can be derived in it.

We will give a rather detailed proof of this theorem since the argument highlights the properties of the system nicely. It also provides a number of examples of fairly difficult derivations. We begin with a sequence of auxiliary lemmata. The first one shows that we can intersperse arbitrary formulae in a disjunction. To avoid unnecessary parentheses, we adopt the convention that logical or associates to the right. Hence, $\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$ is shorthand for $\varphi_1 \vee (\varphi_2 \vee (\ldots(\varphi_{n-1} \vee \varphi_n)\ldots)$. The assertion of the lemma is entirely straightforward, but it turns out that the proof is rather involved.

Lemma 7.4.3 *Expansion Lemma*

Let $1 \leq i_1, i_2, \ldots, i_m \leq n$. Then

$$\varphi_{i_1} \vee \varphi_{i_2} \vee \ldots \vee \varphi_{i_m} \vdash \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$$

Proof. The argument is by induction on m, the number of disjuncts on the left.

For $m = 1$ let $i = i_1$. The proof uses expansion, the commutativity lemma 7.4.2, and another chain of expansion steps. We refer to the lemma by label (L).

$$\begin{array}{cl}
\varphi_i & \\ \cline{1-1}
(\varphi_{i+1} \vee \ldots \vee \varphi_n) \vee \varphi_i & (e) \\ \cline{1-1}
\varphi_i \vee \varphi_{i+1} \vee \ldots \vee \varphi_n & (L) \\ \cline{1-1}
\varphi_{i-1} \vee \varphi_i \vee \ldots \vee \varphi_n & (e) \\ \cline{1-1}
\vdots & (e) \\ \cline{1-1}
\varphi_1 \vee \ldots \vee \varphi_n & (e)
\end{array}$$

Next consider $m = 2$ and write $i = i_1$, $j = i_2$. If $i = j$ we can use contraction in the first step and the proceed as in the case $m = 1$. By commutativity we may safely assume $i < j$. We use induction on $n \geq 2$. The case $n = 2$ is trivial, so assume $n > 2$ and let $\psi = \varphi_3 \vee \ldots \vee \varphi_n$. We consider 3 case depending on i and j. If $i \geq 2$ we have

$$\begin{array}{cl}
\varphi_i \vee \varphi_j & \\ \cline{1-1}
\varphi_2 \vee \psi & (IH) \\ \cline{1-1}
\varphi_1 \vee \varphi_2 \vee \psi & (L)
\end{array}$$

If $i = 1$ and $j \geq 3$ we have

$$\begin{array}{cl}
\varphi_1 \vee \varphi_j & \\ \cline{1-1}
\varphi_i \vee \psi & (IH) \\ \cline{1-1}
\psi \vee \varphi_i & (L) \\ \cline{1-1}
\varphi_2 \vee (\psi \vee \varphi_1) & (e) \\ \cline{1-1}
(\varphi_2 \vee \psi) \vee \varphi_1 & (a) \\ \cline{1-1}
\varphi_1 \vee \varphi_2 \vee \psi & (L)
\end{array}$$

Lastly, we consider $i = 1$ and $j = 2$.

$$\begin{array}{cl}
\varphi_1 \vee \varphi_2 & \\ \hline
\psi \vee (\varphi_1 \vee \varphi_2) & (e) \\ \hline
(\psi \vee \varphi_1) \vee \varphi_2 & (a) \\ \hline
\varphi_2 \vee (\psi \vee \varphi_1) & (L) \\ \hline
(\varphi_2 \vee \psi) \vee \varphi_1 & (a) \\ \hline
\varphi_1 \vee \varphi_2 \vee \psi & (L)
\end{array}$$

Now for the final argument: $m > 2$. Write $\psi = \varphi_1 \vee \varphi_2 \ldots \vee \varphi_n$.

$$\begin{array}{cl}
\varphi_{i_1} \vee (\varphi_{i_2} \ldots \varphi_{i_m}) & \\ \hline
(\varphi_{i_1} \vee \varphi_{i_2}) \vee (\varphi_{i_3} \ldots \varphi_{i_m}) & (a) \\ \hline
(\varphi_{i_1} \vee \varphi_{i_2}) \vee \psi & (IH) \\ \hline
\psi \vee (\varphi_{i_1} \vee \varphi_{i_2}) & (L) \\ \hline
(\psi \vee \varphi_{i_1}) \vee \varphi_{i_2} & (a) \\ \hline
(\psi \vee \varphi_{i_1}) \vee \psi & (IH) \\ \hline
\psi \vee (\psi \vee \varphi_{i_1}) & (L) \\ \hline
(\psi \vee \psi) \vee \varphi_{i_1} & (a) \\ \hline
(\psi \vee \psi) \vee (\psi \vee \psi) & (IH) \\ \hline
\psi \vee \psi & (c) \\ \hline
\psi & (c)
\end{array}$$

□

Though we refer to the last result as the expansion lemma note that it can also be used to rearrange terms in a disjunction, a fact that will be used in several places in the following proofs. Two more technical lemmata, then we can present a completeness proof for $\mathcal{H}$.

Lemma 7.4.4 $\psi \vee \varphi \vdash \neg\neg\psi \vee \varphi$.

Proof.

$$\begin{array}{cl}
\dfrac{\dfrac{\neg\neg\psi \vee \neg\psi}{\neg\psi \vee \neg\neg\psi}\,(L) \qquad \psi \vee \varphi}{\varphi \vee \neg\neg\psi}\,(c) & \\ \hline
\neg\neg\psi \vee \varphi & (L)
\end{array}$$

□

Lemma 7.4.5 $\neg\psi_1 \vee \varphi, \neg\psi_2 \vee \varphi \vdash \neg(\psi_1 \vee \psi_2) \vee \varphi$.

The proof is left as an exercise. If we write $\psi \Rightarrow \varphi$ for $\neg\psi \vee \varphi$, the last lemma can be rephrased in a more intelligible form:

$$\psi_1 \Rightarrow \varphi, \psi_2 \Rightarrow \varphi \vdash (\psi_1 \vee \psi_2) \Rightarrow \varphi.$$

The next and crucial step in the proof of the completeness is to show that tautological disjunctions can be derived.

Lemma 7.4.6 *Every tautology of the form*

$$\varphi = \varphi_1 \lor \varphi_2 \lor \ldots \lor \varphi_n$$

where $n \geq 2$ is provable in the system.

Proof. It might be tempting to use induction on n, but it turns out that we need to perform induction on the total number of connectives in φ. If all the φ_i are literals, we must have a variable and its negation among the literals. But $\neg p \lor p$ is an axiom, so an application of expansion yields φ. So suppose one of the formulae fails to be a literal. By expansion, we may safely assume that φ_1 is that formula. There are three cases, depending on the structure of φ_1. Write $\psi = \varphi_2 \lor \ldots \lor \varphi_n$. First consider $\varphi_1 = \alpha \lor \beta$. Then $\alpha \lor \beta \lor \psi$ is a tautology and hence derivable by IH (recall, we are counting connectives). Now assume $\varphi_1 = \neg\neg\alpha$. Then $\alpha \lor \psi$ is a tautology and hence derivable by induction hypothesis. By lemma 7.4.4 φ is derivable. Lastly, let $\varphi_1 = \neg(\alpha \lor \beta)$. Then both $\neg\alpha \lor \psi$ and $\neg\beta \lor \psi$ are tautologies and hence derivable. By lemma 7.4.5 φ is derivable. □

We can now assemble a complete proof of the tautology theorem.

Proof. (of theorem 7.4.1) Suppose φ is a tautology. Then $\varphi \lor \varphi$ is also a tautology, and by lemma 7.4.6 derivable. A single application of the reduction rule produces a derivation of φ. □

Note that we could actually reconstruct a derivation of a given tautology from the proof; in particular lemma 7.4.6 shows how to construct a derivation for any tautology that is a disjunction. Of course, the derivations would often be unnecessarily complicated and are of little practical interest.

There are other ways to establish the tautology theorem that are somewhat less focused on syntactical arguments. One approach is based on the following lemma by Kalmar which is also of independent interest. For any formula φ and valuation σ define

$$\widehat{\varphi} = \begin{cases} \varphi & \text{if } [\![\varphi]\!]_\sigma = 1, \\ \neg\varphi & \text{otherwise.} \end{cases}$$

Lemma 7.4.7 *Let $\varphi(p_1, \ldots, p_n)$ be a formula with propositional variables as indicated, σ a valuation. Then $\widehat{p_1}, \ldots, \widehat{p_n} \vdash \widehat{\varphi}$*

See the exercises for an outline of the full proof of the tautology theorem using this lemma. The next result describes another essential property of our deduction system: logical implication can express derivability.

Theorem 7.4.2 *Deduction Theorem*

$\Gamma, \varphi \vdash \psi$ implies $\Gamma \vdash \varphi \Rightarrow \psi$.

Proof. Induction on the length n of the derivation of ψ from Γ, φ. If $n = 0$ the conclusion must be one of the premise. If $\psi = \varphi$ then $\varphi \Rightarrow \psi$ is really the axiom $\neg\varphi \lor \varphi$ and we are done; on the other hand if $\psi \in \Gamma$ we can apply expansion. So suppose that $n > 0$. We distinguish four cases depending on the last step in the derivation. If the last step is an expansion $\frac{\alpha}{\beta \lor \alpha}$ then by IH we must have $\Gamma \vdash \neg\varphi \lor \alpha$. By the expansion lemma we get $\neg\varphi \lor \beta \lor \alpha$. If the last step is a contraction $\frac{\alpha \lor \alpha}{\alpha}$ then by IH we must have $\Gamma \vdash \neg\varphi \lor \alpha \lor \alpha$. Again by the expansion lemma (used to perform a contraction, no less) we get $\neg\varphi \lor \alpha$. Next consider the associativity rule $\frac{\alpha \lor (\beta \lor \gamma)}{(\alpha \lor \beta) \lor \gamma}$. By IH we must have $\Gamma \vdash \neg\varphi \lor (\alpha \lor (\beta \lor \gamma))$.

By expansion we can rearrange the formulae to obtain $\alpha \lor \beta \lor \gamma \lor \neg\varphi$ which turns into $(\alpha \lor \beta) \lor \gamma \lor \neg\varphi$ by associativity. Another application of expansion yields $\neg\varphi \lor (\alpha \lor \beta) \lor \gamma$. The last case is a cut $\frac{\alpha\lor\beta \quad \neg\alpha\lor\gamma}{\beta\lor\gamma}$. By IH Γ derives both $\neg\varphi \lor \alpha \lor \beta$ and $\neg\varphi \lor \neg\alpha \lor \gamma$. By expansion we can rearrange these formulae to $\alpha \lor \neg\varphi \lor \beta$ and $\neg\alpha \lor \neg\varphi \lor \gamma$ so that a cut can be applied to obtain $(\neg\varphi \lor \beta) \lor (\neg\varphi \lor \gamma)$. One last application of expansion produces $\neg\varphi \lor (\beta \lor \gamma)$ and we are done. □

We could exploit the Deduction Theorem by adding another rule to our Hilbert system, this one a rule operating on proofs:

$$\text{From} \quad \frac{\Gamma \quad \varphi}{\psi} \quad \text{infer} \quad \frac{\Gamma}{\varphi \Rightarrow \psi}$$

This rule is qualitatively different from the previous ones: it transforms a valid proof into another valid proof. We will return to this idea later. In most applications we need a slightly stronger form of the tautology theorem: even relative to an arbitrary set of assumptions Γ, syntactic and semantic consequence are the same in $\mathcal{H}$. To prove the stronger form we first consider the question of when a set of formulae admits a model.

Definition 7.4.4 *A set Γ of formulae is consistent if $\bot$ is not provable from Γ, inconsistent otherwise. Γ is maximally consistent if Γ is consistent, but none of its proper supersets are consistent.*

It is clear that an inconsistent set will not have a satisfying truth assignment, but it is somewhat surprising that consistency alone suffices to guarantee the existence of a satisfying assignment.

Lemma 7.4.8 *Let Γ be an arbitrary consistent set of formulae. Then Γ is maximally consistent if, and only if, Γ contains either φ or $\neg\varphi$ for each formula φ.*

Proof. Suppose Γ is maximally consistent and neither φ nor $\neg\varphi$ is in Γ. Then $\Gamma, \varphi \vdash \bot$ and $\Gamma, \neg\varphi \vdash \bot$ so that $\Gamma \vdash \bot$, contradicting the consistency of Γ. Of course, by consistency we cannot have both φ and $\neg\varphi$ in Γ.

For the direction from right to left suppose φ is not in Γ. But then $\neg\varphi$ is in Γ and $\Gamma \cup \{\varphi\}$ cannot be consistent. □

Maximally consistent sets are also closed under derivations.

Lemma 7.4.9 *Let Γ be maximally consistent. Then $\Gamma \vdash \varphi$ implies $\varphi \in \Gamma$.*

Proof. Suppose for the sake of a contradiction that $\varphi \notin \Gamma$. Then $\Gamma \cup \varphi$ is inconsistent, so $\Gamma, \varphi \vdash \bot$. But then $\Gamma \vdash \neg\varphi$ and Γ is inconsistent, contradiction. □

As we will see, any consistent set can be extended to a maximally consistent set. In fact, maximally consistent sets turn out to be truth sets. We need an alternative characterization of maximally consistent sets to construct these extensions.

Lemma 7.4.10 *A set of formulae is maximally consistent if, and only if, it is a truth set.*

Proof. Let Γ be the truth set of σ. Γ is trivially consistent, so let $\varphi \notin \Gamma$. Then $[\![\varphi]\!]_\sigma = 0$ so that $\neg\varphi \in \Gamma$. But then $\Gamma \cup \{\varphi\}$ is inconsistent.

For the opposite direction let $\mathit{\Gamma}$ maximally consistent and define a truth assignment by $\sigma(p) = 1$ for $p \in \mathit{\Gamma}$, $\sigma(p) = 0$ otherwise, for all propositional variables p. We will show by induction on the build-up of φ that $\varphi \in \mathit{\Gamma}$ iff $[\![\varphi]\!]_\sigma = 1$. The claim is trivial for propositional variables, so assume that $\varphi = \alpha \vee \beta$. Then $[\![\varphi]\!]_\sigma = 1$ implies $[\![\alpha]\!]_\sigma = 1$ or $[\![\beta]\!]_\sigma = 1$, so by IH we have $\alpha \in \mathit{\Gamma}$ or $\beta \in \mathit{\Gamma}$. In either case φ is derivable from $\mathit{\Gamma}$ and thus an element of $\mathit{\Gamma}$. If, on the other hand, $\varphi \in \mathit{\Gamma}$ then by maximality $\neg\varphi$ cannot be in $\mathit{\Gamma}$. But then α or β must be in $\mathit{\Gamma}$ by lemma 7.4.8. Hence σ models α or β and therefore also models φ. Next consider $\varphi = \neg\psi$. If φ is in $\mathit{\Gamma}$ then ψ is not in $\mathit{\Gamma}$. By IH σ does not model ψ and thus models φ. On the other hand, if σ models φ then σ does not model ψ and by IH ψ is not in $\mathit{\Gamma}$. By the lemma φ must be in $\mathit{\Gamma}$.

The other connectives are left as an exercise. □

There are two versions of the completeness theorem that are easily provable from each other. We begin with the version that asserts that consistency is the same as satisfiability.

Theorem 7.4.3 *Completeness Theorem I*

A set of formulae $\mathit{\Gamma}$ is consistent if, and only if, it has a model.

Proof. It is clear that any $\mathit{\Gamma}$ with a satisfying truth assignment must be consistent. For the opposite direction, assume that $\mathit{\Gamma}$ is consistent and consider a maximally consistent superset $\mathit{\Gamma}'$ of $\mathit{\Gamma}$. As we have seen, $\mathit{\Gamma}'$ is a truth set. But any model of $\mathit{\Gamma}'$ is also a model of $\mathit{\Gamma}$ and we are done. □

The second version of completeness is perhaps the more useful one: it shows that semantic consequence coincides with derivability.

Theorem 7.4.4 *Completeness Theorem II*

Let $\mathit{\Gamma}$ be any set of formulae, φ a formula. Then φ is a semantic consequence of $\mathit{\Gamma}$ if, and only if, it is derivable from $\mathit{\Gamma}$:

$$\mathit{\Gamma} \models \varphi \qquad \textit{iff} \qquad \mathit{\Gamma} \vdash \varphi.$$

Proof. An easy induction on the length of a proof shows that derivability implies semantic consequence. So suppose that $\mathit{\Gamma} \models \varphi$ but that φ is not derivable from $\mathit{\Gamma}$. Then $\mathit{\Gamma} \cup \{\neg\varphi\}$ is consistent and thus has a model σ by theorem 7.4.3. Since φ is a semantic consequence of $\mathit{\Gamma}$ we have $\sigma \models \varphi$, contradicting the fact that $\sigma \models \neg\varphi \in \mathit{\Gamma}'$. □

The completeness theorem has yet another corollary that has quite astonishing consequences.

Theorem 7.4.5 *Compactness Theorem*

Let $\mathit{\Gamma}$ be a collection of formulae. If every finite subset of $\mathit{\Gamma}$ has a model, then $\mathit{\Gamma}$ too has a model.

Proof. If all of $\mathit{\Gamma}$ had no model, we could conclude that $\mathit{\Gamma} \models \bot$. But then by completeness $\mathit{\Gamma} \vdash \bot$. Since any derivation is a finite sequence, it can use only a finite subset $\mathit{\Gamma}_0$ of formulae in $\mathit{\Gamma}$. Thus $\mathit{\Gamma}_0 \vdash \bot$ and we have a contradiction. □

As an application of compactness consider the following claim. A k-coloring of a graph is an assignment of k colors to the vertices so that no two adjacent vertices have the same color.

Proposition 7.4.2 *If every finite subgraph of an undirected graph G is k-colorable, then the whole graph is also k-colorable.*

To see this we express coloring as a propositional formula. More precisely, introduce k variables $p_{x,i}$, $i = 1, \ldots, k$, for each vertex x in the graph. Let Γ consist of all formulae $\mathsf{EO}_k(p_{x,1}, p_{x,2}, \ldots, p_{x,k})$ and $\neg p_{x,i} \vee \neg p_{y,i}$ for any pair of adjacent vertices x and y. Any finite subset of Γ is satisfiable by our assumption about the k-colorability of finite subgraphs of G, so by compactness all of Γ has a model σ. We can read a k-coloring off σ: the color of vertex x is the unique i such that $\sigma(p_{x,i}) = 1$.

7.4.2 Other Deduction Systems

There is nothing sacred about the Hilbert system from above, sound and complete deduction systems for propositional logic can be constructed in many different ways. Here are two more examples of deduction systems with different axioms and rules of inference.

The first system is based on connectives $\neg$ and $\Rightarrow$. There is only one rule of inference, the modus ponens $\dfrac{\varphi \quad \varphi \Rightarrow \psi}{\psi}$. There are three types of axioms:

$$\begin{aligned} &\varphi \Rightarrow \psi \Rightarrow \varphi && (K) \\ &(\varphi \Rightarrow \psi \Rightarrow \gamma) \Rightarrow (\varphi \Rightarrow \psi) \Rightarrow \varphi \Rightarrow \gamma && (S) \\ &(\neg\varphi \Rightarrow \neg\psi) \Rightarrow \psi \Rightarrow \varphi && (T) \end{aligned}$$

We have chosen axiom schemata expressed in terms of meta-variables that range over formulae. We could also have chosen to phrase the axioms as formulae with specific propositional variables, and then add a rule of inference that allows for substitutions. Again it easy to show that this system is sound, but completeness requires some effort. The choice of axioms may seem more or less arbitrary; as long as the formulae are tautologies they won't do any harm – but there is the question of elegance, we want few axioms and preferably they should make intuitive sense. As it turns out, there is an excellent reason to choose (K) and (S). Let $\mathcal{S}$ be a system for propositional logic that contains modus ponens as a rule of inference.

Theorem 7.4.6 *$\mathcal{S}$ satisfies the Deduction Theorem if, and only if, $\mathcal{S}$ proves all instances of axioms (K) and (S).*

We will skip the proof.

Here is one last example of a system with a much richer set of axioms and rules of inferences. We use the positive/negative part machinery from 7.2.4 to define a system KAL (for Klassische Aussagenlogik), see [?]. Write $\Phi_+(\varphi)$ for any formula that contains φ as a positive part, similarly $\Phi_-(\varphi)$ for a negative part, and $\Phi_\pm(\varphi)$ if φ appears both as a positive and negative part.

$$\begin{array}{rl} \text{axioms} & \Phi_\pm(\varphi, \varphi) \\ & \Phi_-(\bot) \\ \text{rules of inference:} & \Phi_+(\psi), \Phi_+(\varphi) \vdash \Phi_+(\psi \wedge \varphi) \\ & \Phi_-(\psi), \Phi_-(\varphi) \vdash \Phi_-(\psi \vee \varphi) \\ & \Phi_-(\psi \Rightarrow \bot), \Phi_-(\varphi) \vdash \Phi_-(\psi \Rightarrow \varphi) \end{array}$$

Hence $p \wedge q \Rightarrow p \vee q$ is an axiom as is $\bot \Rightarrow p$. Note that these axioms and rules are really schemata: each defines an infinite class of individual axioms and rules. The axioms and rules are complicated, but the completeness proof is surprisingly simple given the results from section 7.2.4.

Theorem 7.4.7 *The system KAL is sound and complete.*

Proof. Soundness follows easily from induction on the length of a proof. For completeness, use induction on the number of logical connectives $\land$, $\lor$ and $\Rightarrow$. Any irreducible tautology is an axiom, but any reducible tautology is the conclusion of a rule of inference, whose premise contain fewer connectives and are also tautologies. By induction hypothesis, these are already derivable. □

Note that it becomes quite a bit more difficult to check an alleged proof in this setting. In fact, it is no longer obvious whether a given formula is an axiom, we first have to compute positive and negative parts.

7.4.3 Natural Deduction

Hilbert-style deduction systems are far removed from any kind of reasoning used in practice in a mathematical argument. As a consequence, proofs are notoriously difficult to find, even if the formulae involved are rather modest in size and complexity. Moreover, even if a proof can be found, its structure does not correspond well to the kind of argument a human would put forward. A solution to this problem was first suggested by Gerhard Gentzen in 1933. Consider how a real mathematician might establish a proof for an implication $\varphi \Rightarrow \psi$.

- First, assume φ, then
- pursue some argument till ψ appears, and, lastly,
- conclude that φ indeed implies ψ (discharge the assumption).

The deduction theorem shows that this type of reasoning is indirectly supported by a Hilbert-style system: we can think of φ as being given as an additional axiom used to derive ψ. In the end, the temporary axiom is removed and instead placed as the hypothesis of an implication.

It seems reasonable to try to design a deduction system that does not require a detour via the deduction theorem but allows temporary assumption directly. Such systems are called natural deduction systems. They are much easier to use than Hilbert-style systems, but a bit harder to explain: the rules of inference are more complicated than in a Hilbert system. Typically we have an introduction rule and an elimination rule for each logical connective. The good news: we do not need any logical axioms at all! Everything is built into the rules, but the rules are now more complicated than in a Hilbert system. As it turns out, this arrangement conveys a big advantage overall. Also note that all connectives appear directly, there is no need to define them in terms of others.

And

$$\frac{\phi \quad \psi}{\phi \land \psi}\ (\land i) \qquad \frac{\phi \land \psi}{\phi}\ (\land e) \qquad \frac{\phi \land \psi}{\psi}\ (\land e)$$

Or

$$\frac{\phi}{\phi \lor \psi}\ (\lor i) \qquad \frac{\psi}{\phi \lor \psi}\ (\lor i) \qquad \frac{\phi \lor \psi \quad \begin{matrix}[\phi]\\ \vdots \\ \chi\end{matrix} \quad \begin{matrix}[\psi]\\ \vdots \\ \chi\end{matrix}}{\chi}\ (\lor e)$$

Implication

$$\frac{\phi \quad \phi \Rightarrow \psi}{\psi}\ (\Rightarrow e) \qquad \frac{\begin{matrix}[\phi]\\ \vdots \\ \psi\end{matrix}}{\phi \Rightarrow \psi}\ (\Rightarrow i)$$

These rules are supposed to be interpreted as operating on proofs: if we have a proof for ϕ, and another proof for ψ, then $(\wedge i)$ allows us to combine them into a single proof for $\phi \wedge \psi$. The order in which the proofs are given does not matter. For $(\vee e)$ we need three component proofs. It might be more suggestive to write, say, $(\wedge i)$ as

$$\frac{\begin{array}{c}\vdots\\ \phi\end{array}\quad\begin{array}{c}\vdots\\ \psi\end{array}}{\phi\wedge\psi}\;(\wedge i)$$

In reality, assumptions should be tagged (say, by a numerical subscript) so that one can keep track of where they are discharged. The rule $(\vee e)$ bears some explanation: Assume ϕ, and somehow reason that χ is a consequence, then assume ψ, and somehow reason that χ is again a consequence. Given $\phi \vee \psi$, we can then discharge the assumptions ϕ and ψ and conclude χ.

The rules for negation also require a bit of attention.

Falsum, negation

$$\frac{\bot}{\phi}\;(\bot e)\qquad\frac{\neg\phi\quad\phi}{\bot}\;(\neg e)\qquad\frac{\begin{array}{c}[\phi]\\ \vdots\\ \bot\end{array}}{\neg\phi}\;(\neg i)$$

Double negation

$$\frac{\neg\neg\phi}{\phi}\;(\neg\neg e)\qquad\frac{\phi}{\neg\neg\phi}\;(\neg\neg i)$$

This completes the rules for our natural deduction calculus. The first negation rule is the classical *ex falsum quodlibet*, the principle of explosion: once there is a contradiction, anything follows. The third rule formalizes proof by contradiction. Again, we are dealing with a classical, non-constructive calculus here. For example, if the assumption $\neg\phi$ leads to a contradiction, then we obtain a proof of ϕ as follows, without ever constructing the object in question:

$$\frac{\dfrac{\begin{array}{c}[\neg\phi]\\ \vdots\\ \psi\end{array}\quad\begin{array}{c}[\neg\phi]\\ \vdots\\ \neg\psi\end{array}}{\bot}\;(\neg e)}{\phi}\;(\bot e)$$

Hilbert would approve.

As a general principle, rules like $(\wedge e)$ or $(\vee i)$ are easy to use. But there are three rules where an assumption is made temporarily, and then discharged later: or elimination, implication introduction and negation introduction. These indirect rules are somewhat more difficult to deal with, but essential for the whole system. For natural deduction it is helpful to think of proofs as trees rather than linear sequences; the proof rules then explain how to combine proof trees into larger ones.

Definition 7.4.5 *Let Γ be a set of propositional formulae and φ a formula. There is a derivation or proof of φ from Γ if there is a derivation tree built up from the natural deduction rules that has φ at the root and all of whose assumptions that are not discharged are in Γ.*

Here an assumption is any formula that appears as a leaf in the tree. Unless the formula is in Γ there must be a rule application somewhere along the branch leading to the formula that discharges it. In a proof without premise all assumptions must be discharged. We write $\Gamma \vdash \varphi$ and $\vdash \varphi$ as before for Hilbert-style systems.

Summarizing, the crucial features of natural deduction systems are

- Proofs take place in the context of assumptions.
- Arbitrary assumptions can be made at any time, but
- they all must be discharged to complete the proof.

Note that unlike with a Hilbert style system it is no longer obvious that the system is sound: one has to be careful to address the discharge mechanism.

Here are some sample derivations. We defined natural deduction proofs as trees, but let us start with carefully annotated proof tables. Here is a proof for $p \wedge q, r \vdash q \wedge r$ that requires no assumptions outside of Γ.

1	$p \wedge q$	premise
2	r	premise
3	q	$\wedge\, e_2$, 1
4	$q \wedge r$	$\wedge\, i$, 3, 2

The next proof is slightly more complicated. We will prove that $(p \wedge q) \wedge r \vdash p \wedge (q \wedge r)$.

1	$(p \wedge q) \wedge r$	premise
2	$p \wedge q$	$\wedge\, e_1$, 1
3	p	$\wedge\, e_1$, 2
4	q	$\wedge\, e_2$, 2
5	r	$\wedge\, e_2$, 1
6	$q \wedge r$	$\wedge\, i$, 4, 5
7	$p \wedge (q \wedge r)$	$\wedge\, i$, 3, 6

Rule $(\Rightarrow e)$ is the classical modus ponens. We can show that a proof rule analogous to modus tollens

$$\frac{\neg\psi \qquad \phi \Rightarrow \psi}{\neg\phi}\;(mt)$$

is admissible in this system, in the sense that whenever $\Gamma \vdash \phi \Rightarrow \psi$ and $\Gamma \vdash \neg\psi$ then $\Gamma \vdash \neg\phi$. We'll use variables p and q for clarity.

1	$p \Rightarrow q$	premise
2	p	assmp.
3	q	$\Rightarrow\; e$, 1, 2
4	$\neg q$	premise
5	$\bot$	$\neg\, e$, 3, 4
6	$\neg p$	$\neg\, i$, 2-5

Note that the assumption in line 2 is properly discharged in line 6. Here are some more proofs rendered as trees.

Claim 7.4.1 $p \vee q, p \Rightarrow r, q \Rightarrow s \vdash r \vee s$

$$\dfrac{p \vee q \qquad \dfrac{\dfrac{[p] \quad p \Rightarrow r}{r}\,(\Rightarrow e)}{r \vee s}\,(\vee i) \qquad \dfrac{\dfrac{[q] \quad q \Rightarrow s}{s}\,(\Rightarrow e)}{r \vee s}\,(\vee i)}{r \vee s}\,(\vee e)$$

Claim 7.4.2 $(p \Rightarrow q) \Rightarrow r \vdash p \Rightarrow (q \Rightarrow r)$

$$\cfrac{[p] \qquad \cfrac{\cfrac{\cfrac{[p] \quad [q]}{p \Rightarrow q}\,(\Rightarrow i) \qquad (p \Rightarrow q) \Rightarrow r}{r}\,(\Rightarrow e)}{q \Rightarrow r}\,(\Rightarrow i)}{(p \Rightarrow q) \Rightarrow r}\,(\Rightarrow i)$$

Note that we also have shown $(p \Rightarrow q) \Rightarrow r \vdash q \Rightarrow r$

Derivations involving negation can be a bit more problematic.

Claim 7.4.3 $p \Rightarrow q, \neg p \Rightarrow q \vdash q$

1	$p \Rightarrow q$	$prem$
2	$\neg p \Rightarrow q$	$prem$
3	$\neg q$	$assm$
4	$\neg p$	$(mt), 1, 3$
5	$\neg\neg p$	$(mt), 2, 3$
6	p	$(\neg\neg e), 5$
7	$\bot$	$(\neg e), 4, 6$
8	$\neg\neg q$	$(\neg i), 3 - 7$
9	q	$(\neg\neg e), 8$

As a corollary to the last claim we obtain the tertium non datur.

Corollary 7.4.1 $\vdash p \vee \neg p$

To see this note that $p \Rightarrow p \vee \neg p$ and $\neg p \Rightarrow p \vee \neg p$ are provable.

No doubt, natural deduction is a major improvement over Hilbert-style calculi when it comes to human-like proofs. Still, there are a few technical details that can be glossed over if only a general exposition of natural deduction is intended, but that become crucial in an actual implementation of the system. Notably, we have never explained carefully how the assumptions in a proof are managed. Here is a typical problem: prove $p \Rightarrow p \wedge p$. It is easy to get $p \Rightarrow p \Rightarrow p \wedge p$ by introducing p as an assumption twice, using $\wedge$ introduction to obtain the conjunction and then discharge the two assumptions:

$$\cfrac{\cfrac{\cfrac{p \quad p}{p \wedge p}\,(\wedge i)}{p \Rightarrow p \wedge p}\,(\Rightarrow i)}{p \Rightarrow p \Rightarrow p \wedge p}\,(\Rightarrow i)$$

But to cope with the actual proof goal $p \Rightarrow p \wedge p$ we need to allow for the simultaneous discharge of multiple occurrences of an assumption or some kind of replication mechanism. Alas, not all assumptions of the same formula are necessarily discharged at the same time. Thus, it becomes necessary to tag each assumption, say, with an integer, and at any point during the proof when one of the three discharging rules is applied, all currently active assumptions with the same formula and the same tag are discharged. Their number could be 0, 1 or larger. In the following example the tags are indicated by superscripts.

$$\cfrac{\cfrac{p^1 \quad p^1}{p \wedge p}\,(\wedge i)}{p \Rightarrow p \wedge p}\,(\Rightarrow i)^1$$

Here is an example of 0 assumptions being discharged: q is not an assumption.

$$\dfrac{\dfrac{p^1}{q \Rightarrow p}\;(\Rightarrow i)}{p \Rightarrow q \Rightarrow p}\;(\Rightarrow i)^1$$

7.4.4 Logical Consequence in Natural Deduction

What are the properties of the derivability relation in natural deduction? We have the following analogue of the derivation theorem.

Theorem 7.4.8 *$\Gamma, \phi \vdash \psi$ if, and only if, $\Gamma \vdash (\phi \Rightarrow \psi)$.*

Sketch of proof. The direction from right to left is easy, we can simply apply modus ponens. But the opposite direction is difficult. We use induction on on the length of the proof of ψ. For simplicity, consider only a binary rule as the last step where some inference rule R is applied:

$$\dfrac{\gamma_1 \quad \gamma_2}{\psi}\;(R)$$

Then by IH, $\Gamma \vdash \phi \Rightarrow \gamma_i$. Now consider

$$\dfrac{\dfrac{\dfrac{[\phi] \quad \phi \Rightarrow \gamma_1}{\gamma_1}\;(\Rightarrow e) \qquad \dfrac{[\phi] \quad \phi \Rightarrow \gamma_2}{\gamma_2}\;(\Rightarrow e)}{\psi}\;(R)}{\phi \Rightarrow \psi}\;(\Rightarrow i)$$

Similar arguments work in the remaining cases. □

As for Hilbert-style calculi we have soundness in natural deduction. Alas, the assumptions mechanism makes the proof a bit more complicated.

Theorem 7.4.9 *Gentzen*
Natural deduction is sound and complete: exactly all tautologies can be derived in the system.

Proof. We show that if $\Gamma \vdash \varphi$ and $\sigma \models \Gamma$ then $\sigma \models \varphi$ by induction on the proof φ from Γ.

As in the deduction theorem we have to distinguish cases depending on the last proof rule used. For the rules not discharging an assumption it is not hard to check that any truth assignment satisfying Γ, and thus by IH the premise of the rule, also satisfy the conclusion. As an example for a rule eliminating assumptions consider $(\vee e)$. If truth assignment σ models Γ then is must also model the disjunction $\varphi \vee \psi$. By IH it must then also satisfy χ.

The completeness argument is again based on maximally consistent sets and is very similar to the argument for Hilbert-style systems. Of course, one has to check that the basic properties of maximally consistent sets of formulae still hold in the new setting, but this is fairly straightforward. □

For our purposes, we usually assume a logic with equality: the language contains a special binary symbol $=$ that is always interpreted as identity (as opposed to all the other relation symbols). We add formulae $s = t$ for any two terms s and t. One could argue that $=$ is the most overloaded symbol in all of math, but this usage is entirely standard. To deal with equality, we need two more rules:

$$\begin{array}{c}[t=t]\\ \hline \vdots\end{array}\;(=i) \qquad\qquad \frac{\phi(s) \quad s=t}{\phi(t)}\;(=e)$$

7.4.5 Sequent Calculus

Natural deduction systems are a better environment than Hilbert-style systems. Still, they have one rather cumbersome aspect: the assumption mechanism requires a bit of care. One might suspect that an even more natural system could be built around the idea of bringing the assumptions out into the open and make them a direct part of the inference rules.

Definition 7.4.6 *A sequent consists of two finite sets of formulae $\mathit{\Gamma}$ and $\mathit{\Delta}$. $\mathit{\Gamma}$ is the antecedent and $\mathit{\Delta}$ is the consequent of the sequent. Notation: $\mathit{\Gamma} \supset \mathit{\Delta}$.*

The intended meaning of a sequent is that the conjunction of the formulae in $\mathit{\Gamma}$ (all the assumptions) implies the disjunction of the formulae in $\mathit{\Delta}$. Loosely speaking, we have a list of hypotheses $\gamma_1, \ldots, \gamma_n$ and a list of goals $\delta_1, \ldots, \delta_m$ and we would like to establish the implication

$$\gamma_1 \land \ldots \land \gamma_n \;\Rightarrow\; \delta_1 \lor \ldots \lor \delta_m$$

Both lists may be empty, if the antecedent is empty, the consequent is a tautology; if the consequent is empty, the antecedent is a contradiction.

It may sound tempting to replace $\mathit{\Delta}$ by a single formula but it turns out that the given version is more appropriate for out purposes (we are not concerned with intuitionistic logic).

Keeping the interpretation of "conjunction of antecedent implies disjunction of consequent" in mind, all sequents where the antecedent and consequent contain the same formula are trivially valid. These are called basic sequents and are adopted as axioms:

$$\frac{-}{\mathit{\Gamma}, A \supset A, \mathit{\Delta}.}$$

In natural deduction this corresponds to inferring A from assumption A. Again there will be two rules associated with each connective, this time depending on whether the connective appears in the antecedent or the consequent.

Negation

$$\frac{\mathit{\Gamma} \supset \varphi, \mathit{\Delta}}{\neg\varphi, \mathit{\Gamma} \supset \mathit{\Delta}}\;\mathit{notL} \qquad \frac{\varphi, \mathit{\Gamma} \supset \mathit{\Delta}}{\mathit{\Gamma} \supset \neg\varphi, \mathit{\Delta}}\;\mathit{notR}$$

And

$$\frac{\varphi, \psi, \mathit{\Gamma} \supset \mathit{\Delta}}{\varphi \land \psi, \mathit{\Gamma} \supset \mathit{\Delta}}\;\mathit{andL} \qquad \frac{\mathit{\Gamma} \supset \varphi, \mathit{\Delta} \quad \mathit{\Gamma} \supset \psi, \mathit{\Delta}}{\mathit{\Gamma} \supset \varphi \land \psi, \mathit{\Delta}}\;\mathit{andR}$$

Or

$$\frac{\varphi, \mathit{\Gamma} \supset \mathit{\Delta} \quad \psi, \mathit{\Gamma} \supset \mathit{\Delta}}{\mathit{\Gamma} \supset \varphi \lor \psi, \mathit{\Delta}}\;\mathit{orL} \qquad \frac{\mathit{\Gamma} \supset \varphi, \psi, \mathit{\Delta}}{\mathit{\Gamma} \supset \varphi \lor \psi, \mathit{\Delta}}\;\mathit{orR}$$

Implication

$$\frac{\mathit{\Gamma} \supset \varphi, \mathit{\Delta} \quad \psi, \mathit{\Gamma} \supset \mathit{\Delta}}{\varphi \Rightarrow \psi, \mathit{\Gamma} \supset \mathit{\Delta}}\;\mathit{impL} \qquad \frac{\varphi, \mathit{\Gamma} \supset \psi, \mathit{\Delta}}{\mathit{\Gamma} \supset \varphi \Rightarrow \psi, \mathit{\Delta}}\;\mathit{impR}$$

Cut

$$\frac{\Gamma \supset \varphi, \Delta \quad \varphi, \Gamma \supset \Delta}{\Gamma \supset \Delta}\, \mathit{cut}$$

There is a remarkable theorem due to G. Gentzen that shows that the cut rule, the engine behind many proofs in this calculus, is actually superfluous.

Theorem 7.4.10 (Gentzen, Cut Elimination) *Removing the cut rule does not change the set of provable sequents.*

The proof is based on a fairly tedious induction argument, on the size of cut formula φ and the size of the derivations of the sequents involved. One important application of cut elimination is that it helps to establish consistency: if there were a proof of a contradiction at all, there would also be a proof without using cuts. But it is often not hard to check that the latter kind proofs cannot exist. As to the claim that the cut rule is "superfluous," one should note that proofs without cuts can be hopelessly much longer, one would never want to construct an actual proof of any completexity without cuts.

As presented, we are dealing with sets of formulae rather than lists, so order and multiplicity do not matter. Of course, in implementations the opposite choice is more natural. If antecedent and consequent are lists one has to add more structural rules: Exchange, weakening and contraction for the antecedent.

$$\frac{\Gamma, \psi, \varphi, \Delta \supset \Theta}{\Gamma, \varphi, \psi, \Delta \supset \Theta}\, \mathit{exch} \qquad \frac{\Gamma \supset \Delta}{\varphi, \Gamma \supset \Delta}\, \mathit{weak} \qquad \frac{\varphi, \varphi, \Gamma \supset \Delta}{\varphi, \Gamma \supset \Delta}\, \mathit{cont}$$

A proof is again a tree whose nodes are labeled by sequents corresponding to applications of these rules. To prove φ we have to establish the sequent $\supset \varphi$. Here is an example of such a proof.

$$\frac{\dfrac{p, q \supset p}{p \wedge q \supset p}}{\supset p \wedge q \Rightarrow p}$$

Exercises

Exercise 7.4.1 Show that $A \Rightarrow A$ is derivable in the KST deduction system.

Exercise 7.4.2 Prove theorem 7.4.6. The direction from left to right is easy; for the opposite direction use induction on the length of a Gödel proof for $\Gamma, A \vdash B$.

Exercise 7.4.3 Establish an soundness theorem for $\mathcal{H}$: every derivable formula is a tautology.

Exercise 7.4.4 Write down a few individual axioms and rules of inference in the positive/negative part system. Check that all these individual rules are sound.

Exercise 7.4.5 Explain how to check if a formula is an axiom in the positive/negative system.

Exercise 7.4.6 First, show that $\Gamma, \psi \vdash \varphi$ and $\Gamma, \neg\psi \vdash \varphi$ implies that $\Gamma \vdash \varphi$. Then prove Kalmar's lemma 7.4.7 and use it to give an alternative proof of completeness.

Exercise 7.4.7 Consider a deduction system using only a mystery logical connective $\circ$. The axioms (really schemata) are $p \circ p$, $(p \circ q) \circ (q \circ p)$ and $((p \circ q) \circ r) \circ (p \circ (q \circ r))$. The only rule of inference is $\frac{\varphi \quad \varphi \circ \psi}{\psi}$ and a substitution rule $\frac{\varphi \quad \psi \circ \rho}{\varphi[\rho/\psi]}$. Characterize the derivable formulae in this system. Is there are natural interpretation for $\circ$?

Exercise 7.4.8 Give a detailed proof of the deduction theorem.

Exercise 7.4.9 Carry out the details of the soundness and completeness proof.

Exercise 7.4.10 Establish all the equivalences in lemma 7.2.2.

Exercise 7.4.11 Consider the implication “If it rains, the street is wet.” Use it to verify that an implication is equivalent to its contrapositive, but not its converse and inverse.

Exercise 7.4.12 Give a detailed proof of tertium non datur in natural deduction.

Exercise 7.4.13 Give a natural deduction proof of Peirce’s law $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$. This is more difficult than one might expect. Explain the difficulties that arise in this proof.

7.5 Equational Logic

Propositional logic does not address the issue where the truth values for basic assertions come from. Clearly we need a substantially more expressive language to deal with ordinary mathematical arguments. As a first step in this direction, let us figure out how to deal with equations. As in the case of propositional formulae we start with a graded alphabet to define an appropriate language. This time, the set of constants $\Sigma^{(0)}$ may be arbitrarily large: these symbols are intended to correspond to elements in some mathematical structure, so there is no reason to have only two of them around.

Definition 7.5.1 *We define the rectype of terms $\mathcal{T} = \mathcal{T}(\Sigma) = \mathcal{T}(\mathsf{Var}, \Sigma)$ over Var and Σ as follows:*

- *Every symbol $x \in \mathsf{Var} \cup \Sigma^{(0)}$ is a term, and*
- *given terms $t_1, \ldots, t_n$ and a function symbol $f \in \Sigma^{(n)}$ $f\, t_1 \ldots t_n$ is a term.*

While the strict definition is written using prefix notation in order to avoid the need for auxiliary symbols, we will often informally write operators in infix notation and use parentheses freely to disambiguate.

So far, a term is a purely syntactical object. In order to attach meaning to these expressions we need to interpret them over a concrete structure, a domain of discourse.

First suppose $\Sigma = \{f_1, \ldots, f_k\}$ is finite. Then a suitable structure has the form

$$\mathcal{A} = \langle A, F_1, F_2, \ldots, F_k \rangle$$

where $F_i : A^{\mathsf{ar}(f_i)} \to A$: F_i is an actual function of the right arity that is used to interpret the pure function symbol f_i.

When one thinks of a mathematical structure, the integers, reals, matrices, trees, graphs and so on come to mind. No doubt these are the critical examples, but there is another class of structures that is very important in developing logic: structures whose elements are terms (of equivalence classes thereof). More precisely, we can think of $\mathcal{T} = \mathcal{T}(\Sigma)$ as a structure: the carrier set is the set of terms. Suppose we have a binary function symbol f, then the corresponding map is

$$F(s,t) = fst$$

Clearly $\mathcal{T}$ is a suitable structure for our signature, albeit a particularly sterile one.

Definition 7.5.2 (Valuations) *A valuation or assignment is a map that assigns elements in a structure to variables:*

$$\sigma : \mathsf{Var} \to A$$

By a straightforward induction we can extend a valuation to all terms (rather than just variables):

$$\sigma(f(t_1, \ldots, t_n)) = F(\sigma(t_1), \ldots, \sigma(t_n)).$$

This is just the standard process of evaluation of an expression, given values for all variables and a way to interpret the function symbols. f here is just a syntactic object, a symbol, whereas F is the actual function (the implementation, perhaps even a piece of executable code).

Theorem 7.5.1 (Free Model) *Suppose $\sigma : \mathsf{Var} \to A$ is a valuation over $\mathcal{A}$. Then σ can be extended uniquely to a homomorphism from the term model $\mathcal{T}$ to $\mathcal{A}$.*

In other words, once we assign values to the free variables we can assign values to all terms, in a canonical fashion. Note that this provides an elegant way to specify the corresponding homomorphism, in particular when we are dealing with finitely many variables.

Definition 7.5.3 (Equations) *An equation or identity is an expression $s \approx t$ where s and t are terms.*

Note that this is just syntactic sugar, we might as well have defined an identity to be a pair (s,t). Also, we have deliberately chosen to write $\approx$ rather than the customary and hopelessly overloaded equality sign $=$ to make clear that we are defining a special class of syntactic objects. Later we will be sloppy and write $s = t$.

Now consider an equation $s \approx t$ and a valuation σ over some structure $\mathcal{A}$.

Definition 7.5.4 (Validity) *$s \approx t$ is valid over $\mathcal{A}$ with respect to σ if $\sigma(s) = \sigma(t)$. We also say that $\mathcal{A}$ is a model of $s \approx t$. $s \approx t$ is valid over $\mathcal{A}$ if it is valid for all valuations.*

We use the same terminology for a set E of equations: E is valid over $\mathcal{A}$ if all the equations in E are so valid. Notation:

$$\mathcal{A} \models_\sigma s \approx t$$
$$\mathcal{A} \models s \approx t$$

For a set of equations E we require that all the equations in E hold over $\mathcal{A}$:

$$\mathcal{A} \models E$$

Example 7.5.1 The usual associative law has $\Sigma = \{*\}$ where $\mathsf{ar}(*) = 2$:

$$x * (y * z) \approx (x * y) * z \qquad \text{(assc)}$$

Then (assc) is valid over $\langle \mathbb{N}, + \rangle$ and $\langle \Gamma^\star, \cdot \rangle$ but not over $\langle \mathbb{N}, x^y \rangle$.

Structures with this property are called semigroups.

Example 7.5.2 To write equations in elementary arithmetic we could use 5 function symbols $\Sigma = \{\oplus, \otimes, S, \mathbf{0}, \mathbf{1}\}$ where $\mathsf{ar}(\oplus) = \mathsf{ar}(\otimes) = 2$, where $\mathsf{ar}(S) = 1$ and $\mathsf{ar}(\mathbf{0}) = \mathsf{ar}(\mathbf{1}) = 0$.

Typical examples of equations are

$$\begin{aligned} x \oplus y &\approx y \oplus x \\ x \oplus \mathbf{0} &\approx x \\ x \oplus S(y) &\approx S(x \oplus y) \end{aligned}$$

They are all valid over $\langle \mathbb{N}, +, S, 0 \rangle$.

Example 7.5.3 To pin down monoids we expand our semigroup signature to $\Sigma = (*, 1)$ with arities $(2, 0)$. The axiom system (Mon) for monoids has three equations:

$$\begin{aligned} x * (y * z) &\approx (x * y) * z \\ x * 1 &\approx x \\ 1 * x &\approx x \end{aligned}$$

Then a structure $\mathcal{A}$ of type $(2, 0)$ is a monoid iff $\mathcal{A} \models (\text{Mon})$. One can then show, for example, that any monoid also satisfies $((a * b) * c) * d = a * (b * (c * d))$.

A binary operation $*$ is commutative if the identity

$$x * y \approx y * x$$

A semigroup or monoid whose multiplication is commutative is also called commutative, or Abelian. It is a universal convention to write commutative structures with addition rather than multiplication, and to use 0 for the neutral element. Thus in an Abelian monoid we have

$$x + y \approx y + x \qquad \text{and} \qquad x + 0 \approx 0 + x \approx x$$

Example 7.5.4 The following Cayley table defines a monoid on a 6-element set.

$*$	e	a	b	c	d	f
e	e	a	b	c	d	f
a	a	c	f	e	b	d
b	b	f	a	d	e	c
c	c	e	d	a	f	b
d	d	b	e	f	c	a
f	f	d	c	b	a	e

It is clear that e is the neutral element, and that the monoid fails to be Abelian, but but otherwise it is difficult to read off the structure of the monoid from this table.

Many important monoids satisfy another critical property: one can solve basic equations of the form $a * x = b$.

Definition 7.5.5 *Given a monoid and a element x, an inverse of x is an element y such that*

$$x * y \approx 1 \land y * x \approx 1$$

A monoid where each element has an inverse is a group. The size of a group is called its order.

To axiomatize groups it is best to change the signature: use $\Sigma = (*, ^{-1}, 1)$ with arities $(2, 1, 0)$. Then we can add two more identities to (Mon):

$$x * x^{-1} \approx x^{-1} * x \approx 1$$

So groups also have a simple equational characterization. More precisely, suppose we have a monoid where every element has an inverse. Let's write $\mathsf{inv}(x, y)$ as abbreviation for $x * y \approx 1 \wedge y * x \approx 1$. Then we can prove the following from (Mon):

Claim 7.5.1 The inverse element is always unique:

$$\mathsf{inv}(x, y) \wedge \mathsf{inv}(x, z) \;\; \Rightarrow \;\; z \approx y$$

Hence it makes sense to introduce a new, unary function $^{-1}$ that denotes this uniquely determined y: we are defining an abbreviation

$$x^{-1} \approx y \iff \mathsf{inv}(x, y)$$

Here is a purely equational and somewhat pedantic proof.

Proof. Assuming $\mathsf{inv}(x, y) \wedge \mathsf{inv}(x, z)$ we have

$$z \approx 1 * z \approx (y * x) * z \approx y * (x * z) \approx y * 1 \approx y$$

□

Equational descriptions have the advantage that they can never be self-contradictory, any purely equational set of axioms always has a model.

Proposition 7.5.1 *Every system of equations has a model.*

Proof. A trivial model can be constructed by choosing $A = \{\bullet\}$ and setting all functions to be constant with value $\bullet$. □

Of course, this trivial model is utterly useless, we need a carrier set of size at least 2 for an equation to be interesting. Things change drastically if we allow inequalities or implications between equations. For example, the field axioms force the existence of at least two elements zero and one. At any rate, here are some more interesting examples of groups.

Example 7.5.5 The Cayley table of the 6-element monoid from above actually defines a group. It requires a bit of effort to show that this group is (isomorphic to) the symmetric group on 3 letters.

Example 7.5.6 The additive group $\mathbb{Z}$ of integers:

A	integers
$*$	addition
$^{-1}$	unary minus
1	zero

Example 7.5.7 The multiplicative group $\mathbb{R}_+$ of positive reals:

A	positive reals
$*$	multiplication
$^{-1}$	inverse
1	one

Example 7.5.8 The additive group $\mathbb{Z}_n$ of modular numbers:

A	modular numbers $\{0, 1, \ldots, n-1\}$
$*$	addition modulo n
$^{-1}$	negation
1	zero

Example 7.5.9 The multiplicative subgroup group $\mathbb{Z}_n^\star$ of modular numbers:

A	modular numbers $\{\, x \mid 0 \leq x < n \wedge \gcd(x, n) = 1 \,\}$
$*$	multiplication modulo n
$^{-1}$	inverse modulo n
1	one

The last example requires a small argument to establish the existence of the inverse function. Note that the modulus n is prime iff $Z_n^\star$ has cardinality n.

Example 7.5.10 The special linear group $\mathsf{SL}(n, \mathbb{R})$ of all non-singular $n \times n$ matrices of reals:

A	non-singular matrices $\{\, A \in \mathbb{R}^{n \times n} \mid \lvert A \rvert \neq 0 \,\}$
$*$	matrix multiplication
$^{-1}$	matrix inverse
1	diagonal matrix with all 1s

Of the examples of groups so far, G_6 and the special linear groups are notably different from the others: these groups fail to be commutative (except for 1×1 "matrices"). In fact, it turns out that G_6 is the smallest non-commutative group. Checking the group properties from the table requires work, as usual, but non-commutativity is easy to see. Note the 3 by 3 square in the upper left hand corner: $\{e, r, s\}$ is a commutative subgroup.

One reason to approach groups axiomatically is that the additional level of abstraction actually simplifies certain arguments. For example, using the uniqueness claim from above, we can fairly easily prove the following lemma which describes the interaction between $^{-1}$ and multiplication.

Lemma 7.5.1

$$(x * y)^{-1} \approx y^{-1} * x^{-1}$$

This result holds for all groups, in particular for $\mathsf{SL}(n, \mathbb{R})$. But a direct proof, based on algorithms for matrix multiplication and inverse, is hopelessly complicated, even for small values of n. Try.

Though the group axioms are nearly trivial, just to understand their finite models took a huge amount of effort, the famous *classification of finite simple groups*, completed roughly in 1985, and involving some 15,000 pages of proofs. The classification theorem says in essence that all finite groups can be decomposed into just 5 basic types. Two of these basic groups are very straightforward, e.g., cyclic groups of prime order or alternating groups, two are a bit more messy (certain Lie-type groups) but the 5th class is utterly bizarre: it consists of 26 rather strange groups, the so-called sporadic groups. The largest one of these is called the monster. It can be construed as a group of rotations, albeit in 196883-dimensional space. Its order is

$$808017424794512875886459904961710757005754368000000000$$

This is hugely surprising: how on earth does this number somehow emanate from the ever so modest group axioms?

Suppose we have a system E of equations over some graded alphabet Σ. We already know that the term model $\mathcal{T}$ works when there are no equations in E. Otherwise, define the following equivalence relation $\widetilde{E}$ on $\mathcal{T}$. $\widetilde{E}$ is the least equivalence relation that

- contains E, construed as a relation on $\mathcal{T}$, and
- is closed under substitution in the following two ways:
  - if $s\ \widetilde{E}\ t$ then $s[u/x]\ \widetilde{E}\ t[u/x]$, and
  - if $s\ \widetilde{E}\ t$ then $u[s/x]\ \widetilde{E}\ u[t/x]$.

  Here u is any term and x any variable.

So $\widetilde{E}$ can be a bit bigger than E, it includes all "conclusions" that can be drawn from the equations in E. We write $\mathcal{T}_E$ for the structure obtained from $\mathcal{T}$ by factoring with respect to $\widetilde{E}$. It is easy to see that $\mathcal{T}_E$ is again a suitable structure.

Theorem 7.5.2 *$\mathcal{T}_E$ is the free E-structure generated by* Var.

More precisely, if we have any model $\mathcal{A}$ of E and a map $\sigma : \mathsf{Var} \to A$ then there is a unique homomorphism extending σ from $\mathcal{T}_E$ to $\mathcal{A}$.

As we have seen for the associative law, it may well happen that the validity of one equation in a structure entails the validity of others. More generally, suppose E is a system of equations and e a single equation.

Definition 7.5.6 *e is a semantic consequence of E if for all structures $\mathcal{A}$ such that $\mathcal{A} \models E$ we also have $\mathcal{A} \models e$. Notation: $E \models e$.*

Example 7.5.11 Equation $x * (y * (z * u)) \approx ((x * y) * z) * u$ is a semantic consequence of $x * (y * z) \approx (x * y) * z$. But $x * y = y * x$ is not a consequence of associativity.

Boolean Algebras

Here is another example of a class of important algebraic structures that have a simple equational description. A Boolean algebra is a structure $\mathcal{B} = \langle B, +, \cdot, ^{-}, 0, 1\rangle$ where the following system of equations (BA) is valid:

$$\begin{array}{cc}
x + (y + z) = (x + y) + z & x \cdot (y \cdot z) = (x \cdot y) \cdot z \\
x + y = y + x & x \cdot y = y \cdot x \\
x + 0 = x & x \cdot 1 = x \\
x + (y \cdot z) = (x + y) \cdot (x + z) & x \cdot (y + z) = (x \cdot y) + (x \cdot z) \\
x + \overline{x} = 1 & x \cdot \overline{x} = 0
\end{array}$$

Note that we are in full fudge-mode now, no more $\approx$, $\oplus$ and the like, we are writing he equations the way the would appear in the literature. But be aware that (BA) is just a collection of formal expressions.

The axioms say that $\langle B, +, 0\rangle$ and $\langle B, \cdot, 1\rangle$ are commutative monoids, that full distributivity holds between the two operations, and they describe a "complementation" operation $\overline{x}$. Note the duality between plus and times in (BA).

(BA) has many interesting semantic consequences. Some particularly important ones are:

$$\begin{array}{cc}
x + x = x & x \cdot x = x \\
x + x \cdot y = x & x \cdot (x + y) = x \\
x + 1 = 1 & x \cdot 0 = 0 \\
\overline{x + y} = \overline{x} \cdot \overline{y} & \overline{x \cdot y} = \overline{x} + \overline{y} \\
\overline{\overline{x}} = x &
\end{array}$$

As we have seen, equational proofs of these assertions are fairly tedious for humans, and often require counter-intuitive steps (e.g., making an intermediate expression more complicated).

Claim 7.5.2 $\mathsf{BA} \models x + x = x$

Proof.

$$x + x = (x + x) \cdot 1 = (x + x) \cdot (x + \overline{x}) = x + x \cdot \overline{x} = x + 0 = x$$

The third step uses distributivity of plus over times in the "opposite" direction. □

By duality, we also have $xx = x$ in any Boolean algebra.

7.5.1 Equational Reasoning

With a view towards implementation, let us try to pin down how these arguments really work: we repeatedly substitute equal terms for equal terms and use the obvious properties of equality. The given equations in (BA) and the basic properties of equality are the only source for the equalities that can be used in the argument. As a consequence, the argument is easy to check for correctness: we just have to identify which axiom and what substitution was used. In an annotated proof these would also be given explicitly so the correctness check comes down to pattern matching.

Alas, the real problem is not to check the proof for correctness but to find it in the first place. Following the Hilbert-Gödel approach, we can define an equational proof of e (over some system $\mathcal{E}$) to be a sequence of equations

$$e_1, e_2, \ldots, e_{n-1}, e_n = e$$

where each e_i is either in $\mathcal{E}$, is a trivial identity $s \approx s$ or can be derived from a previous equation e_j, $j < i$ by a simple rule to be spelled out below. For example, we may switch the two sides of an equation. There is one small exception: for transitivity of equality we need two premises. Also, a better representation of an equational proof (at least for humans) is a tree with e as the root. A node is the consequence of its children.

First we need rules that express the fact that equality is reflexive, symmetric and transitive.

$$\frac{-}{s \approx s} \qquad \frac{s \approx t}{t \approx s} \qquad \frac{s \approx t \quad t \approx u}{s \approx u}$$

The first rule has no premise, the second one, and the last two. Also, given a system of equations $\mathcal{E}$ as assumptions, we are allowed to use any equation $s \approx t \in \mathcal{E}$:

$$\frac{-}{s \approx t}$$

Note that these rules are not quite enough: think about the associativity example from above. Two rules relate to substitutions: instantiation and congruence

$$\frac{s \approx t}{s\theta \approx t\theta} \qquad\qquad \frac{s \approx t}{f(\ldots s \ldots) \approx f(\ldots t \ldots)}$$

In other words, we are allowed to substitute arbitrary terms for variables on both sides of an equation $s \approx t$, thereby generating an instance $s\theta \approx t\theta$ of the equation. Here θ is a substitution, a map $\mathsf{Var} \to \mathcal{T}$.

Moreover, if we have $s \approx t$ then we can replace s by t in an arbitrary term and obtain another equation (the ellipses here are sloppy notation for the same sequence of terms on botch sides).

Definition 7.5.7 *e is a syntactic consequence of $\mathcal{E}$ if e can be derived from $\mathcal{E}$ by finitely many applications of these rules. Notation: $E \vdash e$.*

Theorem 7.5.3 (Soundness Theorem) *The deduction system for equations is sound: every syntactic consequence of $\mathcal{E}$ is also a semantic consequence.*

Proof. Straightforward induction on the length of the deduction. □

Soundness is just a sanity check: our deduction system contains no rules that would produce invalid conclusions. But how do we know that no rules are missing? Suppose $\mathcal{E} \models e$. Is there always an equational proof of e from $\mathcal{E}$? The answer is given by a famous theorem:

Theorem 7.5.4 (Birkhoff 1935) *The deduction system for equations is complete: every semantic consequence of $\mathcal{E}$ is also a syntactic consequence.*

Hence

$$\mathcal{E} \models e \iff \mathcal{E} \vdash e$$

So we have a deduction system that is perfect in principle. Indeed, in practice it turns out that using Birkhoff's rules directly often leads to arguments that are quite long and are often very, very tedious to discover – humans are not particularly good at this kind of task.

It helps to have a few other tools lying around that can help in constructing proofs. For example, here is a lemma that helps in establishing equalities in Boolean algebras.

Lemma 7.5.2 (BA) *The complement $\overline{x}$ is unique in the sense that $x+y=1$ and $x \cdot y = 0$ implies $y = \overline{x}$.*

So instead of trying to prove $y = \overline{x}$ directly, we can attempt to prove $x+y=1$ and $x \cdot y = 0$, a task that may well be easier. Of course, we have to prove first that the lemma holds, using only equational reasoning with the given identities.

Proof. Assume $x+y=1$ and $xy=0$.

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

Claim 7.5.3 $(BA) \models \overline{\overline{x}} = x$.

Proof. By the lemma, we only have to show that x behaves like the complement of $\overline{x}$. But

$$\overline{x} + x = x + \overline{x} = 1 \qquad \text{and} \qquad \overline{x}x = x\overline{x} = 0.$$

□

So suppose we have a set of equations $\mathcal{E}$ over some graded alphabet Σ. The big question is: How do we computationally check whether an equation e is a semantic consequence of $\mathcal{E}$? By Birkhoff's theorem, we can simply enumerate all possible equational proofs with axioms in $\mathcal{E}$. If e follows from $\mathcal{E}$ we will discover a proof in finitely many steps. Hence, the problem is semidecidable. Of course, if were to implement proof search we would not simply generate potential proof sequences blindly but would try to "get from $\mathcal{E}$ to e" in some systematic way. One seemingly reasonable approach would be try to generate longer and longer terms that ultimately lead to the desired target equation e. Unfortunately, in the presence of the sufficiently ill-behaved axioms no such simple strategy is going to work: the proof may involve terms of arbitrary size.

Theorem 7.5.5 *In general, it is undecidable whether e is a semantic consequence of $\mathcal{E}$.*

Furthermore, undecidability already rears its ugly head when the signature is very simple. Here is one particularly small example, due to G. S. Tseitin. There are 5 constants, a, b, c, d, e and one binary operation written as concatenation. We omit the associativity axiom and focus on axioms involving the constants.

4 axioms deal with commutation (a and b commute with c and d):

$$ac = ca, ad = da, bc = cb, bd = db.$$

The others are a bit strange:

$$abac = abacc, eca = ae, edb = be.$$

It is undecidable whether an equation between ground terms (no free variables) follows from these axioms.

7.5.2 Axioms for BA

In the 1930s, E. V. Huntington discovered a very simple axiomatization for Boolean algebras. The signature is reduced to $+$ and $\overline{}$.

$$\begin{aligned} x + y &= y + x \\ (x + y) + z &= x + (y + z) \\ x &= \overline{\overline{x} + y} + \overline{\overline{x} + \overline{y}} \end{aligned}$$

According to Huntington's axioms, the operation plus is associative and commutative, and the relation between plus and complement is regulated by the third axiom. It is easy to see that the third axiom holds in any Boolean algebra, but it requires more work to establish the converse. In 1933 H. Robbins conjectured that the third axiom can be replaced by the slightly more cryptic:

$$x = \overline{\overline{x + y} + \overline{x + \overline{y}}}$$

Again it is easy to check that this equation holds in any Boolean algebra, but it is far from clear that the opposite direction also works. It was shown in the 1970s that to prove this conjecture, it suffices to show that Robbins' axioms imply the double negation property $\overline{\overline{x}} = x$ but no one knew how to do this. Robbins' Conjecture was not proven until 1996, and then by the automatic prover EQT, not a human. One annoying feature of this machine generated proof is that it provides little insight: it consists of just 16 steps of equational reasoning, applied to uncomfortably large expressions. Originally, the proof search took 8 days on a workstation.

Small Axiom Systems

The standard axiomatization of Boolean algebra in terms of $\sqcup$, $\sqcap$ and $^{-}$ uses 10 axioms (two commutative monoids, distributivity and complement). To get shorter and presumably easier to work with descriptions one usually focuses on a single universal operator such as *nand* or *nor*. For legibility, write nand as the Sheffer stroke $x \mid y$. Here is a classical axiom system due to Sheffer.

$$\begin{aligned}((x \mid x) \mid (x \mid x)) &= x\\ (x \mid (y \mid (y \mid y))) &= (x \mid x)\\ ((x \mid (y \mid z)) \mid (x \mid (y \mid z))) &= (((y \mid y) \mid x) \mid ((z \mid z) \mid x))\end{aligned}$$

It took Meredith some 20 years to come up with this, smaller system:

$$\begin{aligned}((x|x)|(y|x)) &= x\\ (x|(y|(x|z))) &= (((z|y)|y)|x)\end{aligned}$$

One can do slightly better, as discovered by Wolfram:

$$\begin{aligned}((x \mid y) \mid (x \mid (y \mid z))) &= x\\ (x \mid y) &= (y \mid x)\end{aligned}$$

At this point, one might wonder whether one can push further and axiomatize Boolean algebras with just a single equation. Surprisingly, the answer is yes.

$$(((x \mid z) \mid y) \mid ((x \mid (x \mid y)) \mid x)) = y$$

The equational theorem prover Waldmeister was used to find this axiom. Note that it is quite difficult to recover the standard axioms from this single axiom.

7.5.3 Rewrite Systems

Birkhoff's theorem immediately implies that testing whether $\mathcal{E} \vdash e$ is semidecidable, at least as long as $\mathcal{E}$ is itself decidable). Of course, we really want decidability, and a way to generate proof objects. The main idea to accomplish this is simple: we translate the given identities into rewrite rules. Then we use these rules to simplify the terms: we rewrite term t into a normal form $\nu(t)$. Then, instead of trying to derive an equation $s \approx t$, we rewrite both s and t into normal form and check that $\nu(s) = \nu(t)$. The latter is equality of terms and is trivial to check. Alas, there are several obstructions.

First off, any identity $s \approx t$ produces two rewrite rules $s \to t$ and $t \to s$. Applied blindly, these will produce an infinite loop. Picking only one direction $s \to t$ is always sound, but we may well lose completeness. Second, there are typically many ways to apply the rules corresponding to the given identities, the whole rewrite process is highly nondeterministic. Lastly, the rewrite process may not terminate.

Still, at least in some cases our strategy should work. Ideally we would come up with a collection of rewrite rules that make it unnecessary to worry about the proper order and direction of application: every possible chain of rewrites ultimately ends in the desired normal form. Of course, it is not clear which systems of equations $\mathcal{E}$ have a corresponding civilized rewrite system. Needless to say, if the nice system exists, we really need an algorithm to construct it. And, even if we can construct a nice rewrite system, it is not clear how long the rewrite chains will be, or how big the intermediate expressions may become.

Let's look at a small example: monoids. Suppose we wish to express algebraic simplification in a monoid as a set of rewrite rules. The appropriate language is $\mathcal{L}(*, 1)$ with signature $(2, 0)$. Among others, we adopt the rule

$$x * 1 \to x$$

to express the fact that 1 is a neutral element (on the right). In order to apply this rule we need to operate on subterms as in

$$y * (x * 1) \to y * x$$

and we need to be able to substitute terms for the variable x:

$$(y * z) * 1 \to y * z$$

Fix some graded alphabet Σ and a set Var of variables; let $\mathcal{T}$ be the corresponding term algebra. To make the idea of rewrite rules more precise, write $u[s]_p$ for the term obtained from u by replacing the subterm in position p by s. This is called replacement in context. Other occurrences of s in u are not affected by this operation (unlike with variable substitution $u[x/s]$).

In implementations, the term u is given by a parse tree and the position of a subterm can be specified as a sequence of natural numbers: the sequence expresses a path from the root to some node t. The term (tree) s then replaces the subtree rooted at t.

We can now explain axiomatically what is meant by a general rewrite relation.

Definition 7.5.8 *A binary relation on ρ on $\mathcal{T}$ is compatible if $s \; \rho \; t$ implies that $u[s]_p \; \rho \; u[t]_p$, for all terms u and all positions p in u. The relation is stable if $s \; \rho \; t$ implies that $s\theta \; \rho \; t\theta$ for any substitution θ. ρ is a rewrite relation if it compatible and stable.*

We are mostly interested in rewrite relations that have a nice finite description in terms of a few basic rules: construct the least rewrite relation that encompasses the pairs on the given list. Suppose we have some finite list $\mathcal{E}$ of identities on $\mathcal{T}$:

$$\mathcal{E} = \{s_1 \approx t_1, s_2 \approx t_2, \ldots, s_n \approx t_n\}$$

We can define an associated rewrite relation as follows. First, a one-step relation (often called one-step reduction):

$$u[s\theta]_p \to_{\mathcal{R}} u[t\theta]_p$$

whenever $s \approx t \in \mathcal{E}$ and θ is some substitution. So we pick a subterm in u that is an instance of s and replace it by the corresponding instance of t.

Note that we are replacing identities (which are meant to be undirectional) by directed rewrite rules, there is a natural tension between the two. A single rewrite step is usually not interesting, we need to iterate. Let

$$\begin{array}{ll} \overset{+}{\to}_{\mathcal{E}} & \text{transitive closure of } \to_{\mathcal{E}} \\ \overset{*}{\to}_{\mathcal{E}} & \text{transitive reflexive closureof } \to_{\mathcal{E}} \end{array}$$

As long as $\mathcal{E}$ is finite (or at least semidecidable), these relations are always semidecidable. As an example, consider the following r/w-system $\mathcal{M}$ for monoids. The language is $\mathcal{L}(*, 1)$ with signature $(2, 0)$. $\mathcal{M}$ has three rules:

$$\begin{aligned} (x * y) * z &\to x * (y * z) \\ x * 1 &\to x \\ 1 * x &\to x \end{aligned}$$

Then we have the following reduction:

$$((1 * (a * 1)) * (1 * a)) * b \overset{+}{\to} a * (a * b)$$

Note that no further reduction steps are possible at this point: none of the rules apply to $a * (a * b)$. This is no coincidence, starting at any term in this system we will ultimately be unable to apply any more reduction steps: eliminate all products involving 1 and move the parens to the right.

Definition 7.5.9 *A term s is irreducible or in normal form if there exists no term t such that $s \to_{\mathcal{E}} t$.*

As the monoid example suggests, we would like for each term s to have $s \overset{*}{\to} s'$ where s' is irreducible. In fact, it would be nice if there were exactly one such s', in which case we could define $\nu(s) = s'$. One possible problem is lack of termination:

$$s = s_0 \to s_1 \to s_2 \to \ldots \to s_n \to \ldots$$

Second, since $\to$ is usually nondeterministic, the reduction may fork into to two chains that never again merge.

$$\begin{tikzcd}
 & & u \arrow[r, "*"] & u' \\
s \arrow[r, "*"] & t \arrow[ru] \arrow[rd] & & \\
 & & v \arrow[r, "*"] & v'
\end{tikzcd}$$

Even if the chains starting at u and v are both finite we don't have a unique normal form: there is no reason in general why u' should be the same as v'. In order to rule out these problems we need to impose further conditions on the r/w-system.

Definition 7.5.10 *$\mathcal{E}$ is confluent if for any $s, t_1, t_2 \in \mathcal{T}$ such that $s \to t_1$ and $s \to t_2$ there exists a term t such that $t_1 \overset{*}{\to} t$ and $t_2 \overset{*}{\to} t$.*

$\mathcal{E}$ is terminating or Noetherian if for there are no infinite rewrite chains

$$s_0 \to s_1 \to s_2 \to \ldots \to s_n \to \ldots$$

Note that it is far from clear how one would test whether a given r/w-system has these properties. And, we cannot symmetrize an identity $s \approx t$ to two rules $s \to t$ and $t \to s$ if we want termination.

Claim 7.5.4 The r/w-system $\mathcal{M}$ for monoids is confluent and terminating.

Proof. Recall that the rules are

$$(x * y) * z \to x * (y * z)$$
$$x * 1 \to x \qquad 1 * x \to x$$

For termination, first note that the unit-removal rules $x * 1 \to x, 1 * x \to x$ can be applied only as many times as the term contains the constant 1: there is no way another 1 can be introduced.

Now note that the unit-removal rules commute with the associativity rule. Hence we can move all applications of unit-removal rules to the front of the rewrite sequence. So let's assume t contains no occurrence of 1 and we only apply the associativity rule $(x * y) * z \to x * (y * z)$.

Here is a slightly overkill but elegant proof for termination. Recall that a full binary tree T can be coded as a binary sequence $\lambda(T)$ as follows (this method pushes right up against the information theoretic bound).

- A single-node tree is coded as 0.
- A tree with left subtree a and right subtree b is coded as $1\,\lambda(a)\,\lambda(b)$

Hence if T has n leaves then $\#_0\lambda(T) = n - 1$ and $\#_1\lambda(T) = n$. It so happens that if T' is the result of applying the associativity rule to T then $\lambda(T') < \lambda(T)$. Hence we must have termination.

For confluence, use induction on the size of the tree to show the following. Let $a_1, a_2, \ldots, a_n$ the frontier of the tree of t (from left to right). Then t will be rewritten to

$$a_1 * (a_2 * (\ldots (a_{n-1} * a_n) \ldots))$$

□

Normal Form Systems

The combination of confluence and termination is all we need for unique normalization.

Lemma 7.5.3 *If a rewrite system is confluent and terminating, then every term has a unique normal form.*

Proof. It follows from termination that for some irreducible t we have $s \xrightarrow{*} t$. So suppose that t_1 and t_2 are irreducible and $s \xrightarrow{*} t_1$, $s \xrightarrow{*} t_2$. By confluence there is some term t such that $t_1 \xrightarrow{*} t$ and $t_2 \xrightarrow{*} t$. By irreducibility, this implies $t_1 = t_2$.

□

Again: if we have a confluent and terminating r/w-system we are done in a way: we can blindly apply the rewrite rules and we will always arrive at the same irreducible term, the normal form. Of course, this totally ignores questions of efficiency: the number of steps needed to get to irreducibility/size of intermediate terms may well vary. In reality, we need better strategies to apply the rules.

Let $\overset{*}{\leftrightarrow}$ be the reflexive, symmetric and transitive closure of $\rightarrow$. So this is the equivalence relation that $\mathcal{E}$ defines if we start from a system of identities, this is before the original sin of making things directional. So suppose $\mathcal{E}$ is some system of equations over $\mathcal{T}$. Impose an arbitrary direction on these equations to obtain a rewrite system $\mathcal{E}$.

Lemma 7.5.4 $\mathcal{E} \vdash s \approx t$ *if, and only if,* $s \overset{*}{\leftrightarrow} t$.

Note that $\overset{*}{\leftrightarrow}$ is an equivalence relation and partitions $\mathcal{T}$ into classes of "rewriteable" terms. If we have confluence and termination, we have a canonical representative for each of these equivalence classes:

$$s \overset{*}{\leftrightarrow} t \iff \nu(x) = \nu(t)$$

Example 7.5.12 Augment the three rewrite rules for monoids by the following, in the extended language $\mathcal{L}(*, {}^{-1}, 1)$ of signature $(2, 1, 0)$. Rewrite system $\mathcal{G}$:

$$\begin{aligned}
1^{-1} &\rightarrow 1\\
x^{-1} * x &\rightarrow 1\\
x * x^{-1} &\rightarrow 1\\
(x^{-1})^{-1} &\rightarrow x\\
x * (x^{-1} * y) &\rightarrow y\\
x^{-1} * (x * y) &\rightarrow y\\
(x * y)^{-1} &\rightarrow y^{-1} * x^{-1}
\end{aligned}$$

Church-Rosser

Confluence means: any forking reduction can be made to merge again later. There is a property closely related to confluence that is important in the theory of r/w-systems.

Definition 7.5.11 *A rewrite system is Church-Rosser if for any $s, t \in \mathcal{T}$ such that $s \overset{*}{\leftrightarrow} t$ there exists a term r such that $s \xrightarrow{*} r$ and $t \xrightarrow{*} r$.*

Lemma 7.5.5 *A rewrite system is confluent if, and only if, it has the Church-Rosser property.*

There are several important decision problems associated with any finite rewrite system $\mathcal{E}$.

Problem:	**Common Ancestor Problem**
Instance:	Two terms s and t.
Question:	Is there a common ancestor $r \xrightarrow{*} s$ and $r \xrightarrow{*} t$?

Problem:	**Common Descendant Problem**
Instance:	Two terms s and t.
Question:	Is there a common descendant $s \xrightarrow{*} r$ and $t \xrightarrow{*} r$?

Problem:	**Word Problem**
Instance:	Two terms s and t.
Question:	Are s and t equivalent under $\overset{*}{\leftrightarrow}$?

The terminology for the third problem goes back to the famous word problem for finitely presented groups.

Note that we have fixed the rewrite system for these problems. Technically, this is known as the non-uniform version of the problem. There is an obvious generalization, the uniform version of the problem, where $\mathcal{E}$ is part of the input. So, we may be interested in the Word Problem for a specific rewrite system $\mathcal{E}$ (which may be designed to describe, say, a specific group) or we may try to tackle this problem uniformly for all possible rewrite systems or at least for all rewrite systems in a certain class. Needless to say, the uniform version is harder. In fact, even if the non-uniform version is decidable for each rewrite system in some class, the uniform version may well not be: there may be no effective way to derive the appropriate decision algorithm from the system.

In general, it is undecidable whether $\mathcal{E}$ is terminating. This follows easily from an analysis of Turing machines: we can encode the workings of a Turing machine as a rewrite system.

Theorem 7.5.6 *It is undecidable whether a finite rewrite system is terminating.*

Note that termination is undecidable even for a single fixed starting term (corresponding to, say, the computation of a Turing machine on empty tape). But in concrete cases, motivated by an attempt to model some specific mathematical structure, one can often get by associating a measure of complexity to each term s. The natural expectation is that each rewrite step reduces the complexity of the term.

Definition 7.5.12 *A reduction order is a strict pre-order (reflexive and transitive) $\lhd$ on $\mathcal{T}$ that is stable and compatible.*

Of course, we are interested in reduction orderings that extend the one-step relation (actually, its converse) of the r/w-system in question:

$$s \to t \qquad \text{implies} \qquad s \rhd t$$

If we can find such a reduction order that is also well-founded then there cannot be infinite chains of reductions. Note that ordinary length of terms is compatible and well-founded but not stable.

Lemma 7.5.6 *A rewrite system $\mathcal{E}$ is terminating if, and only if, it has a well-founded reduction order.*

Proof. If $\mathcal{E}$ is terminating we can use the converse of $\xrightarrow{+}$ as the reduction order.

In the opposite direction, consider a rewrite step $s_1 \to s_2$. Then $s_1 = t[u\theta]_p$ and $s_2 = t[v\theta]$ where $(u, v) \in \mathcal{E}$ for some substitution θ and some term t. Since the order is compatible and stable, reduction chains in $\mathcal{E}$ translate into strictly descending chains in the order. Since the order is well-founded we are done. □

7.5.4 Unification and Critical Pairs

For the critical pair method below we need to find a substitution that identifies to terms.

Definition 7.5.13 *Given two terms s and t with free variables among $\mathsf{Var} = \{x_1, \dots, x_n\}$, a unifier for s and t is a substitution $\theta : \mathsf{Var} \to \mathcal{T}$ such that $s\theta = t\,\theta$.*

For example, in the language of rings, let

$$s = (x + y) \cdot x \qquad\qquad t = (y + x) \cdot y$$

There are many unifiers, e.g. $(1/x, 1/y)$, $(x + 1/x, x + 1/y)$ and so on. But one of these is the most general: $\theta = (x/x, x/y)$. Most general means that every unifier σ is already an instance of θ: $\sigma = \tau \circ \theta$. This is the most general unifier (MGU) for s and t.

It was shown by A. Robinson in 1965 that

- If two terms are unifiable at all then they also have a most general unifier.
- In which case the most general unifier is unique.
- There is an algorithm to test unifiability. The algorithm returns the MGU if it exists, `No` otherwise.

The idea is that one scans s and t from left to right (assuming prefix form) and finds the first subterms s' and t' where s and t disagree. These are called critical subterms. In order to be able to unify s' and t', the critical subterm condition (CSC) must be satisfied:

- one term is a variable x, and
- the other term is some term not containing x.

In which case $x \mapsto t'$ is part of the substitution; otherwise unification has failed. The whole algorithm consists of repeated applications of this basic step.

```
// Robinson unification
    $\theta = I$
    while there are critical subterms $s'$, $t'$
        if CSC is satisfied
        then
            apply substitution $x/t'$ to $s$ and $t$
            apply substitution $x/t'$ to $\theta$
        else
            return NO
    return $\theta$
```

Logic programming languages such as PROLOG rely heavily on Horn clauses since the more general case is prohibitively complex. Note that to get real mileage out of this machinery one needs unification. The problem is that one has to deal with more than just propositional variables: there will be terms such as

$$R(x, f(y)) \to S(x)$$

that express some relationship and/or operations on a domain of individuals (R is a binary relation symbol, S a unary relation symbol and f a unary function). The problem now is that the "literals" involve patterns and one has to work harder to make things match up. Here is just one example and a little application. Suppose we have clauses

$$T(w) \to R(g(z), w), R(x, f(y)) \to S(x)$$

We would like to conclude that from T one can infer S, but one must take the relation and function symbols into account, and the variables. The way to handle this by finding a unifier, a substitution θ, such that

$$R(g(z), w)[\theta] = R(x, f(y))[\theta]$$

In this case $x \mapsto g(z), w \mapsto f(y)$ works. Substitution $x \mapsto g(z), w \mapsto f(y)$ produces

$$R(x, f(y))[\theta] = R(g(z), f(y)) = R(g(z), w)[\theta]$$

Once the unifier is applied the literals match up directly and we get

$$T(f(y)) \to R(g(z), f(y)), R(g(z), f(y)) \to S(g(z))$$

from which we can conclude

$$T(f(y)) \to S(g(z))$$

Using such general terms allows one to perform computations as proofs.

$$\begin{aligned} &\to \mathsf{add}(x, 0, x) \\ \mathsf{add}(x, y, z) &\to \mathsf{add}(x, y^+, z^+) \\ &\to \mathsf{mult}(x, 0, 0) \\ \mathsf{mult}(x, y, t), \mathsf{add}(t, x, z) &\to \mathsf{mult}(x, y^+, z) \end{aligned}$$

Note that these are really just the primitive recursive definitions of addition and multiplication. From these axioms Φ we can prove assertions such as

$$\mathsf{mult}(0^{++}, 0^{+++}, 0^{++++++})$$

Note that since we can only deal with refutations here, the right way to do this is to show that

$$\Phi \cup \{\mathsf{mult}(0^{++}, 0^{+++}, 0^{++++++}) \to \bot\}$$

is a contradiction. This particular result is admittedly not too impressive, but the method shows a deep connection between computation and logic. Writing $\mathsf{res}(z)$ for "result" we can even ask the system to show that $\mathsf{res}(z)$ holds for some z, given our axioms: it has to refute

$$\Phi \cup \{\mathsf{mult}(0^{++}, 0^{+++}, z) \to \mathsf{res}(z), \mathsf{res}(z) \to \bot\}$$

In fact, by keeping track of the unifiers we can get a concrete value for z, in this case $6 = 0^{++++++}$. So the refutation produces the correct value of the computation.

Knuth-Bendix

Suppose we have a terminating r/w-system $\mathcal{E}$. How do we check confluence? Note that the problem is not even obviously semidecidable: it seems to involve a search over all forking reductions followed by an application of the Common Descendant Problem. If two rules have non-overlapping handles (which we won't define precisely here) then they coexist: we can apply them in arbitrary order without any problem. Potential obstructions to confluence come from the pairs of rules that overlap in a sense. We need to make this precise. Consider two rules

$$s_1 \to t_1 \quad \text{and} \quad s_2, \to t_2$$

where, without loss of generality, the variables in s_1 are distinct from the variables in s_2. Now suppose s_1 has a subterm s', not a variable, such that s' and s_2 are unifiable, and let θ be a MGU. Then $s_1\theta$ admits two different applications of the rules:

- We can replace all of $s_1\theta$ by $t_1\theta$ according to rule 1.
- We can replace only $s'\theta$ by $t_2\theta$ according to rule 2.

The resulting two terms are called a critical pair.

Theorem 7.5.7 *Let $\mathcal{E}$ be terminating. Then $\mathcal{E}$ is already confluent if it is confluent on all critical pairs.*

It is now natural to try to force confluence as follows:

- First, compute critical pairs for the terms currently under consideration.
- Then add rules that guarantee confluence for each critical pair.

Unfortunately, this method may not terminate–but often it does so terminate, and produces reasonable normal forms.

Exercises

Exercise 7.5.1 Find a way to represent terms over, say, the graded alphabet for groups and implement this process of substitution.

Exercise 7.5.2 Show that the tree coding used in termination really works as advertised.

Exercise 7.5.3 Fill in the details for the confluence argument.

Exercise 7.5.4 Show that a rewrite system that has a rule $x \to t$ where x is a variable cannot be terminating. Show that a rewrite system that has a rule $s \to t$ where the variables in t are not a subset of the variables in s cannot be terminating.

Exercise 7.5.5 Suppose $\mathcal{E}$ is a rewrite system so that for every rule $s \to t$ we have $|s| > |t|$ and for every variable x: $|s|_x \geq |t|_x$. Show that $\mathcal{E}$ must be terminating.

Exercise 7.5.6 Add two constants a and b to $\mathcal{M}$. Argue that the new system represents all words over a two-letter alphabet. What happens if we add one more rule $b * a \to b * b$?

Exercise 7.5.7 Show how to test whether a particular rule $s \to t \in \mathcal{E}$ is applicable to a term u: is there a position p in u (i.e. a subterm of u) and a substitution θ such that $u = u[s\theta]_p$.

Exercise 7.5.8 Determine whether the rewrite system from example 7.5.12 is terminating and/or confluent.

Exercise 7.5.9 Prove the lemma 7.5.5 characterizing confluent rewrite systems.

Exercise 7.5.10 Construct a well-founded reduction order for the r/w-system for monoids.

Exercise 7.5.11 Implement the critical pair algorithm. A high level language such as Ocaml or Mathematica is highly recommended.

7.6 First-Order Logic

In the 20th century, mathematics has settled down on a general framework that has become the (nearly) unchallenged standard:

Structures Domains of discourse (such as group theory, topology, analysis, number theory, ...) are captured in structures. This is one of Bourbaki's central contributions.

Sets Set theory is the background theory describing structures and their relationships.

Logic Logic expresses propositions and organizes proofs.

For this to work reasonably well, we the "right" logic: expressive enough to capture all the assertions we are interested in, with a solid proof theory to support reasoning about these assertions. Yet the logical system should not be too cumbersome and plausibly support the style of argument that is traditional in mathematics. The standard answer to this challenge is first-order logic. As a practical matter, most actual work in mathematics and TCS can be construed as the study of particular structures and the interconnections between them. When it comes to foundational matter, some lip service is typically paid to set theory, but no one ever bothers to really express matters in strict set theoretic terms (Bourbaki first volume being a noteworthy exception). Likewise, the rôle of logic is rather limited: there are boilerplate claims that the actual arguments given could be rephrased entirely in terms of first-order logic and some set theory. These claims are then promptly ignored: "strict first-order proofs would be too large, too cryptic, too tedious". While there is much truth to these objections, the easy availability of powerful digital computers is likely to change the landscape. At present, reconstructing difficult mathematical arguments with the help of proof checkers, proof assistants and theorem provers is hard labor, and the benefits are not always clear. Yet, as Y. I. Manin commented:

> Generally, computer science, that no-nonsense child of logic, will exert growing influence on our thinking about the languages by which we express our vision of mathematics.

As usual, in the design of a logic there are three major parts to attend to:

- language (syntax)
- model theory (semantics)
- proof theory (deductions)

Describing a language is fairly straightforward, in particular if one is not interested in actually building a parser or implementing various algorithms. Finding highly efficient implementations for the use in proof assistants and the like, though, is quite a challenge. For first-order logic, the corresponding model theory brushes up against set theory, at least for infinite models, and can be quite daunting. Fortunately, for finite models there are no foundational issues. Proof theory is arguably the most difficult part: one can define deduction systems for first-order logic that are perfect in the sense that they are sound and complete. Proofs here are fairly simple finite data structures, but understanding and organizing them in detail is quite hard and efficiency questions become rather difficult to address.

First, let us develop an appropriate language. We would like to be able to formalize assertions such as

> There are infinitely many primes.

It might be tempting to introduce a logic construct "there are infinitely many number," but it turns out we can get away with far less. First, a formula that expresses the infinitude of primes will be interpreted over the natural numbers, so we don't have to worry about specifying the type of objects under discussion. We do want to be able to assert existence of certain objects, so we will allow an existential quantifier, written $\exists\, x$, where x is a type of variable that is supposed to range over the domain. Similarly, we want to be able to reference all objects in the given domain, so there will be a universal quantifier, written $\forall\, x$. We retain propositional logic, though without the propositional variables, and we allow for terms and identities as in equational logic. We also introduce predicates to be applied to terms as in $P(s,t)$; these will be collected in second graded alphabet Σ'.

To be able to make statements about arithmetic in particular we would admit the following symbols: constants 0, 1, binary function symbols $+,*$ and a binary relation symbol $<$, with the obvious interpretations over the structure of natural numbers. In addition, we allow equality. In this language, primality can be expressed by a formula $\mathsf{prime}(x)$ as follows:

$$1 < x \wedge \forall\, u, v\, (x = u \cdot v \Rightarrow u = 1 \vee v = 1)$$

It remains to deal with the "infinitely many" part. Though we have to corresponding quantifier, we can exploit the natural order on $\mathbb{N}$ to fake one: the sentence

$$\forall\, x\, \exists\, y\, (x < y \wedge \mathsf{prime}(y))$$

interpreted over the naturals clearly means that there are infinitely many primes y; thus, we would declare this sentence to be true. But note the the truth value of the minor modification

$$\forall\, x\, \exists\, y\, (x < y \wedge \mathsf{prime}(y) \wedge \mathsf{prime}(y + 1 + 1))$$

is currently open.

The language of first-order logic consists of the following pieces:

constants	that denote individual objects,
variables	that range over individual objects,
relation symbols	that denote relations,
function symbols	that denote functions,
logical connectives	"and," "or," "not," "implies," "false,"
existential quantifiers	that express "there exists,"
universal quantifiers	that express "for all."

Clearly one can organize these symbols into graded alphabets similar to the approach taken with propositional and equational logic. Hence, we can convey the structure of the non-logical part of the language in a signature or similarity type: a list that indicates the arities of all the function and relation symbols. For details, see the next section. Since there are many syntactical categories to contend with, and in order to avoid tedious details, it is a good idea to rely on notational conventions such as the following:

- a, b, c, ... for constants,
- x, y, z, ... for variables,
- $\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\bot$ for the logical connectives,
- $\exists$ for the existential quantifier,
- $\forall$ for the universal quantifier,
- f, g, h, ... for function symbols,
- R, P, Q, ... for relation symbols.
- Always allow $=$ for equality.

Again, constants and variables are interpreted to range over the carrier set of a structure, there is not need to express particular types such as naturals, rationals and reals in terms of the language. If need be, one may express arities by superscripts as in $R^2(x, y)$: R is a binary relation symbol.

There are occasions when one wants to use countably many function or relation symbols. These will always be given in form a rectype, so that the collection of first-order formulae is again a rectype. For example, we could allow for constants c_n, $n \in \mathbb{N}$; together with axioms $c_i \neq c_j$ for $i < j$ this would make sure that all models are infinite. In mathematical logic one also considers languages of some higher cardinality $\kappa > \aleph_0$, but this will be of no interest to us. Note that some amount of set theory is needed just to define such a language.

Example 7.6.1 (Fields) For the classical algebraic theory of fields one minimally uses a language

$$\mathcal{L}(+, \cdot, 0, 1) \quad \text{of signature } (2, 2, 0, 0)$$

So there are two binary function symbols, and two constants; there are no relations. It is then true in any field that

$$\forall\, x\, \exists\, z\, (x \neq 0 \Rightarrow z * x = 1)$$

where $x \neq 0$ is an abbreviation for $\neg(x = 0)$. Of course, more functions could be added. For example, we could add a multiplicative inverse (a Skolem function) and write the last assertion as

$$\forall\, x\, (x \neq 0 \Rightarrow x^{-1} * x = 1)$$

On the other hand, the formula $1 + 1 = 0$ holds only in some fields, those of characteristic 2.

Example 7.6.2 (Boolean Algebra) In Boolean algebra one uses the language

$$\mathcal{L}(\sqcup, \sqcap, {}^{-}, 0, 1) \quad \text{of signature } (2, 2, 1, 0, 0)$$

Some example of universally true assertions for Boolean algebras:

$$\begin{aligned}
&\forall\, x, y\, \exists\, z\, (x \sqcup y = \overline{z})\\
&\exists\, x\, \forall\, y\, (x \sqcap y = 0)\\
&\forall\, x\, \exists\, y\, (x \sqcup y = 1 \wedge x \sqcap y = 0)
\end{aligned}$$

Example 7.6.3 (Linear Orders) Suppose we have a single binary relation symbol $<$. We can axiomatize a (strict) linear order as follows:

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

expressing irreflexivity, transitivity and trichotomy, respectively. In the non-strict case we traditionally use a symbol like $\leq$ and we have $\forall\, x\, (x \leq x)$ and antisymmetry instead: $\forall\, x, y\, (x \leq y \wedge y \leq x \Rightarrow x = y)$. Density is easy to express:

$$\forall\, x, y\, \exists\, z\, (x < y \Rightarrow x < z \wedge z < y)$$

Assertions written out in first-order logic may seem overly terse and difficult to parse, but consider the following classical statement:

> There do not exist four numbers, the last being larger than two, such that the sum of the first two, both raised to the power of the fourth, are equal to the third, also raised to the power of the fourth.

Much better is a natural-language/logic notion mix, in this case

> There are no positive integers x, y, z and n, where $n > 2$, such that $x^n + y^n = z^n$.

We can now recognize Fermat's Last Theorem. We can push things one step further and write

$$\neg\, \exists\, x, y, z, n\, \Big(n > 2 \wedge x^n + y^n = z^n\Big)$$

assuming our language has exponentiation. Hybrid notation seems to work particularly well with the human cognitive system, and essentially all the modern literature is written this way. On the other hand, pure first-order logic formulae are better suited as input to algorithms. Note, though, that the formalization tends to clobber the narrative, all the informal hints that can be expressed in natural language.

7.6.1 Syntax

Given appropriate graded alphabets for function and relation symbols, one indicates the corresponding first-order language by $\mathcal{L}(\Sigma, \Sigma')$. While we are not interested in writing a parser or theorem prover, we still need to be a bit more careful about the syntax of our language of first-order logic. The components of a formula can be organized into a taxonomy like so:

- variables, constants and terms,
- equations,
- atomic formulae,

- propositional connectives, and
- quantifiers.

Every programming language has a defining report (which no one ever reads, other than perhaps compiler writers, it's the document that uses the imperative "shall" a lot), so think of this as the defining report for first-order logic. Only the quantification part is really new; propositional connectives will be the same as in propositional logic while terms and equations will be the same as in equational logic.

We need a supply of variables Var as well as function symbols and predicate symbols, both organized into graded alphabets Σ' and Σ''. We write $\Sigma = (\Sigma', \Sigma'')$. Then $\mathcal{L}(\Sigma)$ denotes the language constructed from these alphabets. Function symbols of arity 0 are constants and relation symbols of arity 0 are Boolean values (true or false) and will always be written $\top$ and $\bot$. The signature of $\mathcal{L}(\Sigma', \Sigma'')$ consists of the two arity maps, without the actual symbols. In most concrete application, the signature will be finite. However, in the literature you will often find a big system approach that introduces a countable supply of function and relation symbols for each arity. For our purposes a signature custom-designed for a particular application is more helpful.

Definition 7.6.1 (Terms) *The set $\mathcal{T} = \mathcal{T}(\Sigma) = \mathcal{T}(\mathsf{Var}, \Sigma)$ of all terms is defined by*

- *Every variable is a term.*
- *If f is an n-ary function symbol, and $t_1, \ldots, t_n$ are terms, $n \geq 0$, then $f(t_1, \ldots, t_n)$ is also a term.*

A ground term is a term that contains no variables.

Note that $f()$ is a term for each constant (0-ary function symbol) f. For clarity, we write a, b, c and the like for constants.

The idea is that every ground term corresponds to a specific element in the underlying structure. For arbitrary terms we first have to replace all variables by constants. For example, in arithmetic the term $(1 + 1 + 1) \cdot (1 + 1)$ corresponds to the natural number 6. We could introduce constants for all the natural numbers, but there is no need to do so: we can build a corresponding term from the constant 1 and the binary operation $+$.

Given a few terms, we can apply a predicate to get a basic assertion like $x + 4 < y$ in arithmetic.

Definition 7.6.2 (Atomic Formulae) *An atomic formula is an expression of the form*

$$R(t_1, \ldots, t_n)$$

where R is an n-ary relation symbol, and the $t_1, \ldots, t_n$ are terms.

These atomic formulae correspond to the atomic assertions in propositional logic: once we have values for the variables that might appear in the terms, an atomic formula can be evaluated to true or false. But note that we do need bindings for the variables, $R(x, y)$ per se has no truth value.

Equality

Equality plays a special rôle in our setup. If we wanted to be extra careful, we would select a special binary relation symbol $=$ in our language, with the intent that $s = t$ means that terms s and t denote the same element. Thus, unlike with all the other relation symbols, the meaning of $=$ is fixed once and for all: it is always interpreted as equality. The system just described is first-order logic with equality.

One can also consider first-order logic without equality, where there is no direct way of asserting equality of terms. Lastly, one can consider the fragment of first-order logic that has no function symbols nor equality, only relations (the actual predicate logic). In the presence of equality, one still fake functions to a degree by considering relations F such that

$$\forall\, x\, \exists\, y\, F(x,y) \wedge \forall\, x, u, v\, (F(x,u) \wedge F(x,v) \Rightarrow u = v)$$

Definition 7.6.3 (First-order Formulae) *The set of formulae of first-order logic is defined by*

- *Every atomic formula is a is a formula.*
- *Formulae are closed under logical connectives.*
- *If φ is a formula and x a variable, then $(\exists\, x\, \varphi)$ and $(\forall\, x\, \varphi)$ are also formulae.*

The φ in $(\exists\, x\, \varphi)$ and $(\forall\, x\, \varphi)$ is called the matrix of the formula. In quantified formulae, we do not require the quantification variable to be free in the matrix, but we will often write expressions like $\exists\, x\, \varphi(x)$ to indicate the presence of x. Note that a term by itself is not a formula: it denotes an element (if it has no free variables) rather than a truth value. As usual employ infix notation and parentheses and write formulae like $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, and $(\varphi \Rightarrow \psi)$. Parens are omitted whenever possible. One often contracts quantifiers of the same kind into one block. For example,

$$\forall\, x\, \forall\, y\, \exists\, z\, (= (+(x,z), y))$$

turns into the eminently readable

$$\forall\, x, y\, \exists\, z\, (x + z = y)$$

As an aside, we note that some authors appear to take delight in obfuscation and things as difficult to parse as humanly possible. For example, you may find

$$(\forall\, x\,)(\forall\, y\,)(\exists\, z\,) f(x,y) = g(z)$$

as opposed to the much more readable

$$\forall\, x, y\, \exists\, z\, \big(f(x,y) = g(z)\big)$$

Definition 7.6.4 (Sentences) *A variable that is not in the range of a quantifier is free or unbound (as opposed to bound). We write $\mathsf{FV}(\varphi)$ for the set of free variables in φ. A formula without free variables is closed, or a sentence.*

One often indicates free variables like so:

$\varphi(x,y)$	x and y may be free
$\exists\, x\, \varphi(x,y)$	only y may be free
$\forall\, y\, \exists\, x\, \varphi(x,y)$	closed.

Substitutions of terms for free variables are indicated like so:

$\varphi(s,t)$	replace x by s, y by t
$\varphi[s/x, t/y]$	replace x by s, y by t

We will explain shortly how to compute the truth value of a sentence. Note well that the notation $\varphi(x_1, \ldots, x_n)$ only expresses the fact that $\mathsf{FV}(\varphi) \subseteq \{x_1, \ldots, x_n\}$, it does not mean that they all actually occur, some or even all of them may be missing. This turns out to be preferable over the alternative. This may initially sound strange, but it is really no different from saying that $2x^2 - y$ is a polynomial in variables x, y and z.

A priori, a formula $\varphi(x)$ with a free variable x has no truth value associated with it: we need to replace x by a ground term to be able to evaluate. However, taking inspiration from equational logic, it is convenient to define the truth value a formula with free variables to be the same as its universal closure: put universal quantifiers in front, one for each free variable.

$$\forall\, x, y, z \left(x * (y * z) = (x * y) * z \right)$$

is somewhat less elegant and harder to read than

$$x * (y * z) = (x * y) * z$$

In algebra, the latter form is much preferred.

Note that according to our definition it is perfectly agreeable to quantify over an already bound variable. To make the formula legible one then has to rename variables. For practical reasons it is best to simply disallow clashes between free and bound variables.

$$\forall\, x \left(R(x) \land \exists\, x\, \forall\, y\;\; S(x, y) \right)$$

Better: rename the x inside

$$\forall\, x \left(R(x) \land \exists\, z\, \forall\, y\;\; S(z, y) \right)$$

These issues are very similar to problems that arise in programming languages (global and local variables, scoping issues). They need to be addressed but are not of central importance.

7.6.2 Model Theory

So far, all we have is a mildly detailed description of the syntax of first-order logic. To be of real use, we have to have a clear definition of truth for first-order sentences. To this end, we follow Tarski's approach from the 1930s. The key is to define the truth value of a formula relative to a given structure, a mini-universe that allows us to make sense out of the components of the formula. Here is a simple example from basic arithmetic over the natural numbers. Consider the formula

$$\varphi = \forall\, x\, \exists\, y\, (x < y)$$[3]

Its components are

x, y	variables, range over natural numbers
$<$	comparison, the less-than relation

The intended meaning of the formula, expressed in standard math-speak, is then: "For any natural number x there exists a natural number y such that x is less than y." So this formula is clearly true, valid. The key problem is that the formula φ itself does not express any of the auxiliary information: there is no indication that the variables range over natural numbers, and there is no indication that the binary relation symbol $<$ corresponds to the standard order relation on the naturals. This would become more clear if we had written $\forall\, x\, \exists\, y\, R(x, y)$ instead, the use of a well-known symbol such as $<$ is just syntactic

[3] We are overloading the equality symbol $=$ here, but no harm is likely to come from this.

sugar. Moreover, if we were to interpret the variables as ranging over a finite segment of the naturals, the formula would be false, invalid. To settle all these issues we have to consider so-called first-order structures over which the formulae of first-order logic can be evaluated. Fix some signature and a language $L = \mathcal{L}(\Sigma)$ of this signature.

Definition 7.6.5 *A (first order) structure $\mathcal{A}$ is a set together with a collection of functions and relations on that set. $\mathcal{A}$ is an L-structure if for every function symbol f of L of arity k there is a function $f^{\mathcal{A}} : A^k \to A$ in $\mathcal{A}$, and for every relation symbol R of arity k there is a relation $R^{\mathcal{A}} \subseteq A^k$. The signature of a first order structure is the list of arities of its functions and relations.*

In order to interpret formulae in $\mathcal{L}(\Sigma)$ the signatures have to match, which we will tacitly assume from now on. So a structure in general looks like so:

$$\mathfrak{A} = \langle A; f_1, f_2, \dots, R_1, R_2, \dots \rangle$$

The set A is the carrier set of the structure. Unary and binary functions and relations are by far the most important in applications, but higher arities may occur. This is quite similar to abstract data types, where we are dealing with objects together with operations and predicates on them.

In actual use, one often fails to distinguish carefully between a function symbol and its interpretation over a structure:

$$\begin{array}{lcl} f \text{ function symbol} & \rightsquigarrow & f^{\mathfrak{A}} \text{ a function in } \mathfrak{A} \\ R \text{ relation symbol} & \rightsquigarrow & R^{\mathfrak{A}} \text{ a relation in } \mathfrak{A} \end{array}$$

For example, for a standard structure such as the natural numbers one may write + for addition, both for the syntactic symbol and the actual arithmetic operation.

The equality symbol = plays a special rôle among the binary relation symbols: its interpretation is fixed to be actual equality over $\mathfrak{A}$. Hence the following formulae are all valid

$$\begin{aligned}
&\forall x\,(x = x)\\
&\forall x\,, y(x = y \Rightarrow y = x)\\
&\forall x\,, y, z(x = y \wedge y = z \Rightarrow x = z)\\
&\forall \boldsymbol{x}, \boldsymbol{y}\,(\boldsymbol{x} = \boldsymbol{y} \Rightarrow (R(\boldsymbol{x}) \Leftrightarrow R(\boldsymbol{y})))\\
&\forall \boldsymbol{x}, \boldsymbol{y}\,(\boldsymbol{x} = \boldsymbol{y} \Rightarrow f(\boldsymbol{x}) = f(\boldsymbol{y}))
\end{aligned}$$

If we were to treat equality like any other relation symbols, we could use these assertions as axioms for equality.

At any rate, given this interpretation of function and relation symbols over $\mathfrak{A}$, we can determine whether a formula holds true over $\mathfrak{A}$. This is very different from trying to establish universal truth. All we are doing here is to confirm that, in the context of a particular structure, a certain formula is valid. As it turns out, this is all that is really needed in the real world. Of course, when the structure changes the formula may well become false.

We can now give a first informal definition of truth or validity.

Definition 7.6.6 *A formula of first-order logic is valid if it holds over any structure of the appropriate signature.*

We do not restrict the formula to be a sentence: for free variables as in $x * y = y * x$ validity means that we can bind the variables to all elements of the structure. In other words, we are really considering the universal closure of the formula, $\forall\, x, y\, (x * y = y * x)$. Note that we apply the same approach to function and relation symbols (other than equality): any possible interpretation has to be taken into account. Of course, we can pin down some of the properties of the corresponding functions and relations in the formula itself, but that's all we can do. There is no magic that forces, say, the symbol + to always be interpreted as some sort of commutative addition.

What are examples of valid formulae? It is clear that a formula like $\varphi \Rightarrow \varphi$ is valid. In fact, if we replace the propositional variables in any tautology by arbitrary sentences we obtain a valid sentence of first-order logic. More precisely, let

$$\varphi(p_1, p_2, \dots, p_n)$$

be a tautology with propositional variables $p_1, p_2, \dots, p_n$. Let $\psi_1, \dots, \psi_n$ be arbitrary sentences of first-order logic. Then

$$\varphi(\psi_1, \psi_2, \dots, \psi_n)$$

is a valid sentence of first-order logic. True, but not too interesting.

Again in analogy to propositional logic we can define satisfiability.

Definition 7.6.7 *A formula of first-order logic is satisfiable if it is true for some interpretation of the variables, functions and relations. It is a contradiction if it is true for no interpretation of the variables, functions and relations.*

For example, in the language of binary relations, the formula $x\ R\ x \wedge (x\ R\ y \wedge y\ R\ z \Rightarrow x\ R\ z)$ is satisfied exactly by any structure $\mathfrak{A}$ that carries a reflexive transitive relation $R^{\mathfrak{A}}$. On the other hand, $\forall\, x\, (x \neq c)$ where c is a constant, is a contradiction: we can always interpret x as the element in the structure denoted by c in which case equality holds.

Slightly more complicated examples for valid formulae can be obtained from quantifier manipulations:

$$\begin{aligned}
\forall\, x\, \forall\, y\, \varphi(x, y) &\Rightarrow \forall\, y\, \forall\, x\, \varphi(x, y) \\
\exists\, x\, \exists\, y\, \varphi(x, y) &\Rightarrow \exists\, y\, \exists\, x\, \varphi(x, y) \\
\exists\, x\, \forall\, y\, \varphi(x, y) &\Rightarrow \forall\, y\, \exists\, x\, \varphi(x, y)
\end{aligned}$$

Note, though, that the following is not valid:

$$\forall\, x\, \exists\, y\, \varphi(x, y) \ \Rightarrow\ \exists\, y\, \forall\, x\, \varphi(x, y)$$

Counting

Another good source of valid formulae are assertions about the number of elements in the underlying structure. How do we say "there are exactly n elements in the ground set" in first-order logic? First, a formula which states that there are at most n elements.

$$\mathsf{Most}n = \exists\, x_1, \dots, x_n\, \forall\, y\, (y = x_1 \vee y = x_2 \vee \dots \vee y = x_n)$$

Second, a formula which states that there are at least n elements.

$$\mathsf{Least}_n = \exists\, x_1, \dots, x_n\, (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge \dots \wedge x_{n-1} \neq x_n)$$

All these formulae are clearly satisfiable. The conjunction $\mathsf{Most}n \wedge \mathsf{Least}_n$ pins down the cardinality to exactly n. Also, a formula $\mathsf{Least}_n \Rightarrow \mathsf{Least}_m$ is valid whenever $m \leq n$.

How about a formula that states that there are infinitely many elements? Note that ham-fisted attempts such as

$$\mathsf{Least}_1 \wedge \mathsf{Least}_2 \wedge \ldots \wedge \mathsf{Least}_n \wedge \ldots \qquad \text{or} \qquad \forall\, n\, \mathsf{Least}_n$$

do not work since these expressions are not syntactically correct formulae.

Dedekind's Trick

After some more fruitless attempts one might suspect that the statement "there are infinitely many things" cannot be expressed in first-order logic. Nothing could be further from the truth. Let f be a unary function symbol and c a constant. Consider

$$\varphi = \forall\, x\, (f(x) \neq c) \wedge \forall\, x, y\, (f(x) = f(y) \Rightarrow x = y).$$

So φ states that f is injective but not surjective. Hence, in any interpretation that makes φ true, the carrier set must be infinite: for finite sets, injectivity is equivalent to surjectivity.

Again, there are natural decision problems associated with this classification.

Problem:	**Validity**
Instance:	A first-order logic formula φ.
Question:	Is φ valid?

Problem:	**Satisfiability**
Instance:	A first-order logic formula φ.
Question:	Is φ satisfiable?

In practice, one is often more interested in the search version of Satisfiability: we would like to construct a satisfying interpretation if one exists. Note that this may well entail the construction of an infinite structure. As one might suspect from the few examples, these problems are much harder in first-order logic than in propositional logic and will turn out to be highly undecidable in general. Incidentally, in computer science this is called model checking.

It is high time to give a precise definition of validity and satisfiability. We begin by defining assignments in the context of first-order logic.

Definition 7.6.8 *An assignment or valuation (over a structure $\mathfrak{A}$) associates variables of the language with elements in the ground set A.*

Given an assignment $\sigma : \mathsf{Var} \to A$, we can associate an element $\sigma(t)$ in A with each term t.

- $t = x$: then $\sigma(t) = \sigma(x)$
- $t = f(r_1, \ldots, r_n)$: then $\sigma(t) = f^{\mathfrak{A}}(\sigma(r_1), \ldots, \sigma(r_n))$

Example 7.6.4 Over $\mathfrak{N}$ let $\sigma(x) = 3$. Then $\sigma(x \cdot (1 + 1))) = 6$ whereas $\sigma(x) = 0$ produces $\sigma(x \cdot (1 + 1))) = 0$.

Once we have an assignment for all the free variables in an atomic formula we can determine a truth value for it.

Definition 7.6.9 *Let σ be an assignment over a structure $\mathfrak{A}$ and $\varphi = R(t_1, \dots, t_n)$ an atomic formula. Define the truth value of φ (under σ over $\mathfrak{A}$) to be*

$$\mathfrak{A}_\sigma(\varphi) = \begin{cases} \mathsf{tt} & \text{if } R^{\mathfrak{A}}(\sigma(t_1), \dots, \sigma(t_n)) \text{ holds,} \\ \mathsf{ff} & \text{otherwise.} \end{cases}$$

Example 7.6.5 Over the natural numbers $\mathfrak{N}$ suppose $\sigma(x) = 0$ and $\sigma(y) = 1$. Then

$$\mathfrak{N}_\sigma(x + y < 1 + 1) = \mathfrak{N}_\sigma(0 + 1 < 1 + 1) = \mathsf{tt}$$

but for $\sigma(x) = \sigma(y) = 1$ we get

$$\mathfrak{N}_\sigma(x + y < 1 + 1) = \mathfrak{N}_\sigma(1 + 1 < 1 + 1) = \mathsf{ff}$$

Once we have a truth value for atomic formulae, we can extend this evaluation to compound formulae without quantifiers.

Definition 7.6.10 (Propositional Connectives)

- $\varphi = \psi \wedge \chi$: *then* $\mathfrak{A}_\sigma(\varphi) = H_{and}(\mathfrak{A}_\sigma(\psi), \mathfrak{A}_\sigma(\chi))$
- $\varphi = \psi \vee \chi$: *then* $\mathfrak{A}_\sigma(\varphi) = H_{or}(\mathfrak{A}_\sigma(\psi), \mathfrak{A}_\sigma(\chi))$
- $\varphi = \neg\psi$: *then* $\mathfrak{A}_\sigma(\varphi) = H_{not}(\mathfrak{A}_\sigma(\psi))$

Example 7.6.6 Suppose $\sigma(x) = 0$ and $\sigma(y) = 1$. Then

$$\begin{aligned} \mathfrak{N}_\sigma(x < y \vee y < x) &= H_{or}(\mathfrak{N}_\sigma(x < y), \mathfrak{N}_\sigma(y < x)) \\ &= H_{or}(\mathfrak{N}(0 < 1), \mathfrak{N}(1 < 0)) \\ &= H_{or}(\mathsf{tt}, \mathsf{ff}) \\ &= \mathsf{tt} \end{aligned}$$

For an assignment σ, let us write $\sigma[a/x]$ for the assignment that is the same as σ everywhere, except that $\sigma[a/x](x) = a$. Think of this as substituting "a for x".

Definition 7.6.11 (Quantifiers)

- $\varphi = \exists x \, \psi$:
  Then $\mathfrak{A}_\sigma(\varphi) = \mathsf{tt}$ *if there is an a in A such that* $\mathfrak{A}_{\sigma[a/x]}(\psi) = \mathsf{tt}$.
- $\varphi = \forall x \, \psi$:
  Then $\mathfrak{A}_\sigma(\varphi) = \mathsf{tt}$ *if for all a in A* $\mathfrak{A}_{\sigma[a/x]}(\psi) = \mathsf{tt}$.

Note that σ only needs to be defined on the free variables of φ to produce a truth value for φ, the values anywhere else do not matter. If φ is a sentence, σ can be totally undefined.

Definition 7.6.12 (Formulae) *A formula φ is valid in a suitable structure $\mathfrak{A}$ under assignment σ if $\mathfrak{A}_\sigma(\varphi) = \mathsf{tt}$. A formula φ is valid in $\mathfrak{A}$ if it is valid in $\mathfrak{A}$ for all assignments σ. The structure $\mathfrak{A}$ is then said to be a model for φ or to satisfy φ. A sentence is valid (or true) if it is valid over any structure (of the appropriate signature).*

Notation: $\mathfrak{A} \models_\sigma \varphi$, $\mathfrak{A} \models \varphi$, $\models \varphi$ One uses the same notation for sets of formulae Γ. So $\mathfrak{A} \models \Gamma$ means that $\mathfrak{A} \models \varphi$ for all $\varphi \in \Gamma$.

Note the condition for validity: the formula has to hold in all structures.

Definition 7.6.13 *A formula is satisfiable if there is some structure $\mathfrak{A}$ and some assignment σ for all the free variables in φ such that $\mathfrak{A} \models_\sigma \varphi$.*

In other words, the existentially quantified formula

$$\exists\, x_1, \ldots, x_n\ \varphi(x_1, \ldots, x_n\,)$$

has a model, where $x_1, \ldots, x_n$ are all the free variables of φ. In shorthand: $\exists\, \boldsymbol{x}\, \varphi(\boldsymbol{x})$.

This is analogous to validity where we insist that $\forall\, x_1, \ldots, x_n\ \varphi(x_1, \ldots, x_n\,)$ holds, or $\forall\, \boldsymbol{x}\, \varphi(\boldsymbol{x})$ in compact notation.

The definitions of truth given here is due to A. Tarski (two seminal papers, one in 1933 and a second one in 1956, with R. Vaught). A frequent objection to this approach is that we are using "for all" to define what a universal quantifier means. True, but the formulae in our logic are syntactic objects, and define their meaning in terms of structures, which are not syntactic, they are real (in the world of mathematics and TCS). Think of this as a program: you can "compute" the truth value of a formula, as long as you can perform certain operations in the structure (evaluate $f^{\mathfrak{A}}$, loop over all elements, search over all elements, ...). This is a bit problematic over infinite structures, but for finite ones we can actually perform the computation (at least if we ignore efficiency).

More precisely, suppose we wanted to construct an algorithm

$$\mathsf{ValidQ}(\mathfrak{A}, \varphi)$$

that checks if formula φ is valid over structure $\mathfrak{A}$. What would be the appropriate input for such an algorithm? The easy part is the formula: any standard representation (string, parse tree, sequence number) will be fine. The real problem is the structure $\mathfrak{A}$. For standard structures such as the natural numbers or reals we understand (more or less) how to interpret the operations. But in the general case we need some representation: we need to specify a first-order structure in some finitary way.

Suppose $\mathfrak{A}$ is a structure of some signature Σ. In order to describe $\mathfrak{A}$ the first step is to augment the language $\mathcal{L}(\Sigma)$ by constant symbols c_a for each element a in the carrier set A of $\mathfrak{A}$, obtaining a new signature $\Sigma_{\mathfrak{A}}$. $\mathfrak{A}$ is naturally also a structure of signature $\Sigma_{\mathfrak{A}}$. This step is not necessary when there already are terms in the language for all the elements of the structure. E.g., in arithmetic we can denote every natural number by a numeral, a ground term of the form

$$\underline{n} = 1 + 1 + \ldots + 1$$

This works since we are dealing with the naturals, but in general there is no reason why every element in a structure should be denoted by a term.

Definition 7.6.14 (Atomic Diagram) *The (atomic) diagram of a structure of some fixed signature is the set of all atomic sentences and their negations in $\mathcal{L}(\Sigma_{\mathfrak{A}})$ that are valid in $\mathfrak{A}$. In symbols:* $\mathsf{diag}\,\mathfrak{A}$.

The point is that the validity of any formula over $\mathfrak{A}$ is completely determined by $\mathsf{diag}\,\mathfrak{A}$: no other information is used in our definition of truth. Hence our algorithm should take as inputs the diagram and the formula and use recursion to obtain the answer:

$$\mathsf{ValidQ}(\mathsf{diag}\,\mathfrak{A}, \varphi)$$

If the underlying structure $\mathfrak{A}$ is finite (and thus the diagram is finite), then we can actually perform this computation: it is just recursion and copious table lookups. In fact, ValidQ will be primitive recursive given any reasonable coding.

How do we represent the atomic diagram $\mathsf{diag}\,\mathfrak{A}$? Given enough constants we can simply write down table. E.g., for a unary function symbol f we can use a table with entries c_a and c_b provided that $b = f^{\mathfrak{A}}(a)$. Each entry corresponds to an identity $f(c_a) = c_b$ in the diagram. For binary function symbols we get a classical Cayley style "multiplication table", and higher dimensional tables for functions of higher arity. Relations can be handled by similar tables with entries in $\mathbb{B}$. This corresponds to the familiar interpretation of a relation $R \subseteq A^k$ as a function $R : A^k \to \mathbb{B}$. So, the whole diagram is just a bunch of tables using special constants for all elements in the structure and Boolean values. For small structures, this is a perfectly good representation, though even for finite but large structures explicit tables are not feasible.

Pushing ahead into the realm of infinite structures things become much more complicated. If the carrier set is uncountable our machinery from classical computability theory simply does not apply – the individual elements are not finitary objects and we have no handle. However, if the carrier set is countable computability theory does apply and we can use it to measure the complexity of the structure.

Definition 7.6.15 (Complete Diagram) *The (complete) diagram is the collection of all sentences in $\mathcal{L}(\Sigma_{\mathfrak{A}})$ that are valid in $\mathfrak{A}$. In symbols: $\mathsf{diag}^c\mathfrak{A}$.*

$\mathfrak{A}$ is computable if its atomic diagram is decidable. $\mathfrak{A}$ is decidable if its complete diagram is decidable.

All finite structures are trivially decidable and even primitive recursive, though things become more complicated if we consider lower levels of the computational hierarchy. For example, satisfiability of a propositional formula is easily expressed as a validity problem of a formula over a two-element structure, yet no polynomial time algorithm is known for this problem. Also, often finite structures are parametrized (for example by the number of elements) and one really would like a solution uniformly in terms of the parameter. This is usually much harder than staring at a single finite structure.

Definability

Suppose we have some first-order structure $\mathfrak{A}$ over carrier set A and a relation $R \subseteq A^n$. The question arises whether first-order logic is expressive enough to describe R in terms of a formula.

Definition 7.6.16 (First-order Definability) *R is first-order definable (over $\mathfrak{A}$) if there is a formula $\varphi(x_1, \ldots, x_n)$ such that*

$$R = \{ (a_1, \ldots, a_n) \in A^n \mid \mathfrak{A} \models \varphi(a_1, \ldots, a_n) \}$$

As we have seen already, primality is definable over $\mathfrak{N}$, the structure of arithmetic. On the other hand, reachability in graphs is not first-order definable: to assert the existence of a path requires quantification over sequences of arbitrary length (which is provably impossible in first-order logic). We will give a proof of this claim later.

7.6.3 Theories and Models

We now have a formal definition of what it means for a structure being a model of a sentence, or a whole set of sentences, as in $\mathcal{A} \models \Gamma$. The set of sentences Γ is a fairly simple object as a data structure, but the first-order structure $\mathcal{A}$ is more problematic. Indeed, we need some background theory that allows us to talk about these structures, typically Zermelo-Fraenkel set theory. It is well worth exploring the properties of $\models$ a bit further.

Definition 7.6.17 (Semantic Consequence) *Given a set Γ of sentences and a sentence φ, we say that φ is valid in Γ or a semantic consequence of Γ if any structure $\mathcal{A}$ that satisfies all formulae in Γ also satisfies φ: $\mathcal{A} \models \Gamma$ implies $\mathcal{A} \models \varphi$. Notation: $\Gamma \models \varphi$.*

There are purely logical semantic consequences such as $\varphi(t) \models \exists\, x\, \varphi(x)$ or $\varphi, \varphi \Rightarrow \psi \models \psi$. These are straightforward, but in general, semantic entailment is not always so easy to see. For example, the standard group axioms entail $(x * y)^{-1} = y^{-1} * x^{-1}$.

Fix some first-order language L once and for all. As always, we are only interested in languages with a finite or at most countable supply of non-logical symbols, and decidable syntax.

Definition 7.6.18 *A theory (over L) is an arbitrary collection of sentences (over L). A theory T is satisfiable if there is some structure $\mathfrak{A}$ such that $\mathfrak{A} \models T$.*

This is the minimalist definition of theory, some authors insist that a theory be closed under semantic consequence: $\Gamma \models \varphi$ implies $\varphi \in \Gamma$. Obviously, theories that have no models at all are of little interest, they are self-contradictory. Alas, it is surprisingly difficult to avoid contradictions when one constructs some logical system (not necessarily first-order logic). For Hilbert, satisfiability was the key to existence:

> If the arbitrarily given axioms do not contradict each other through their consequences, then they are true, then the objects defined through the axioms exist. That, for me, is the criterion of truth and existence.

Constructivists recoil at this idea, but is hard to see how one could convince mainstream mathematicians to let go of Hilbert's dream.

Example: Axiomatizing Groups

Let us see how this plays out with our standard examples from algebra: semigroups, monoids and groups. Initially, we will use the small language $\mathcal{L}(*)$ of type (2). So there is only one binary function symbol $*$, and no constants. We will write $*$ in infix notation to keep things readable and drop the universal quantifiers up front.

We can axiomatize semigroups is any structure satisfying the following associativity axiom:

$$x * (y * z) \approx (x * y) * z \tag{7.1}$$

If, in addition, there is a neutral element

$$\exists\, y\, \forall\, x\, (x * y \approx x \;\wedge\; y * x \approx x) \tag{7.2}$$

then we have a monoid. This is better expressed in an extended language $\mathcal{L}(*, e)$ of type $(2, 0)$; the second axiom then turns into the more intelligible

$$x * e \approx x \;\wedge\; e * x \approx x \tag{7.3}$$

Similarly, we could describe the critical property of a group, the existence of inverse elements, via

$$\forall\, x\, \exists\, y\, (x * y \approx e \;\wedge\; y * x \approx e) \tag{7.4}$$

Again, it is convenient to extend the language to $\mathcal{L}(*, {}^{-1}, e)$ of type $(2, 1, 0)$ and write the last condition as

$$x * x^{-1} \approx e \;\wedge\; x^{-1} * x \approx e \tag{7.5}$$

The message is that judicious use of function (and relation) symbols can cut down on quantifier clutter and improve legibility. One can also argue that the last language corresponds most closely to what an actual algebraist would use.

Equivalence

If one thinks of theories as axiom systems for a particular domain, it is natural to ask when they do actually describe the same class of objects.

Definition 7.6.19 (Theory Equivalence) *Two theories T_1 and T_2 are equivalent if they have the same models.*

Note that T_1 and T_2 have to be over the same language here–this is not quite true, it is possible to handle extensions. For example, all the standard descriptions of groups over languages $\mathcal{L}(*)$, $\mathcal{L}(*, e)$ and $\mathcal{L}(*, {}^{-1}, e)$ are equivalent in the technical sense.

Definition 7.6.20 (Theory of Structures) *Given a class $\mathfrak{C}$ of structures, we define the theory of $\mathfrak{C}$ to be the collection of sentences valid in all the structures in $\mathfrak{C}$:*

$$\mathsf{Th}(\mathfrak{C}) = \{\, \varphi \mid \text{ for all } \mathcal{A} \in \mathfrak{C} : \mathcal{A} \models \varphi \,\}$$

If there is only one structure $\mathcal{A}$ in the class, we write $\mathsf{Th}(\mathcal{A})$. For example, $\mathsf{Th}(\mathfrak{N})$ is the theory of arithmetic. On the other hand, $\mathsf{Th}(\text{Grp})$ is the theory of groups, all assertions that hold true in all groups; $\mathsf{Th}(\text{BoolAlg})$ is the theory of Boolean algebras and so on. This is quite different from trying to understand the theory of a specific group, or a specific Boolean algebra.

Definition 7.6.21 (Semantic Closure) *Let T be a theory. The semantic closure of T is defined by*

$$\mathsf{Th}(T) = \{\, \varphi \mid T \models \varphi \,\}$$

In other words, if $\mathfrak{M}$ is the class of all models of T, then $\mathsf{Th}(T) = \mathsf{Th}(\mathfrak{M})$. It is entirely possible that $\mathsf{Th}(T)$ is much more complicated than T itself: in general, we know little about the class of all possible models.

Definition 7.6.22 (Theory Models) *For any theory T, the class of models of T is defined to be*

$$\mathsf{Mod}(T) = \{\, \mathcal{A} \mid \mathcal{A} \models T \,\}.$$

where all the structures have the appropriate signature.

It follows that $\mathsf{Th}(T) = \mathsf{Th}(\mathsf{Mod}(T))$. For example, the collection of all groups is described concisely by the theories from above. Similarly, we can describe Abelian groups or infinite groups. But note that this does not work for the collection of all finite groups.

It has become one of the standard methods in math and CS to describe (classes of) structures $\mathfrak{C}$ by selecting a theory Γ such that $\mathfrak{C} = \mathsf{Mod}(\Gamma)$.

Definition 7.6.23 *Let T be any theory. A theory Γ is a set of axioms for T if T and Γ are equivalent. Similarly, a theory Γ is a set of axioms for $\mathfrak{C}$ if T is equivalent to $\mathsf{Th}(\mathfrak{C})$.*

This is a white lie: axioms are supposed to be few in number and self-evident, they should encapsulate the fundamental truths of the domain of discussion, and there should be no point in trying to analyze them any further. Another tacit requirement: all axioms should be logically independent, no one axiom should follow from the others. Elegance matters greatly in the selection of axioms, but it seems to be rather difficult to formalize.

It is important to note that axiom systems can have two entirely different purposes: one may try to

- describe a large class of models; e.g., all groups or all fields of characteristic 2.
- pin down one particular structure; e.g., the natural numbers, the rationals or the reals.

As we will see, first-order logic is not particularly good at the second task, one often has to contend with unintended models. These bad models are not due to a poor choice of axioms, they just cannot be avoided in a first-order system.

The question arises whether axioms can always be kept simple in a technical sense. Let T be any consistent theory.

Definition 7.6.24 *T is finitely axiomatizable if it admits a finite set of axioms. T is recursively axiomatizable if it admits a decidable set of axioms. We use the same terminology for classes of structures. In addition, $\mathfrak{C}$ is axiomatizable if there is some set of axioms for it.*

Theories that fail to be recursively axiomatizable are essentially too complicated for first-order logic (see below for proofs in first-order logic). Also note that a theory is always axiomatizable, but a class of structures may not be.

The following are all finitely axiomatizable:

- standard structures from algebra such as groups and fields
- ordered fields
- lattices
- Boolean algebras
- Neumann-Bernays-Gödel set theory

These are recursively axiomatizable:

- torsion-free groups
- fields of characteristic 0
- algebraically closed fields
- finite fields
- Peano arithmetic
- Zermelo-Fraenkel set theory

Not axiomatizable at all are the following.

- The collection of finite groups FinGrp is not axiomatizable: statements that hold for arbitrarily large groups also hold for some infinite groups.
- The collection of all connected graphs ConGrf is not axiomatizable: there is no way to express the existence of a path of unbounded length in first-order logic.
- The class of all well-orderings is not axiomatizable: we cannot quantify over sets or sequences of individuals.

We mention in passing that parallel terminology is used in some papers. There, a class of structures $\mathfrak{C}$ is elementary if there is a theory T such that $\mathfrak{C} = \mathsf{Mod}(T)$. A class of structures $\mathfrak{C}$ is basic elementary if there is a finite theory T such that $\mathfrak{C} = \mathsf{Mod}(T)$. On top, some authors use “elementary” in place of “basic elementary.” We will avoid this terminology and stick with axiomatizable, finitely axiomatizable and recursively axiomatizable.

7.6.4 Dedekind-Peano Arithmetic

MISSING

7.6.5 Decidability and Completeness

Again, fix some first-order language L once and for all. We assume that L is at most countable, and has decidable syntax.

Definition 7.6.25 (Decidable Theories) *A theory T is decidable if $\mathsf{Th}(T)$ is decidable.*

There is a potential source of confusion here: we can think of T as a set of sentences in its own right, and T may be decidable in this sense. However, $\mathsf{Th}(T)$ may still be undecidable, and that is the property that is of greater interest, whence the definition. As an example of a highly undecidable theory, consider

$$T = \text{ all true statements of arithmetic}$$

in the language $\mathcal{L}(+,*,0,1;<)$ of signature $(2,2,0,0;2)$. In fact, not even the universal sentences in T (such as the Riemann hypothesis) admit a decision procedure. On the other hand, Peano arithmetic (PA) is a small subtheory of T that has a recursive set of axioms (i.e., is easily decidable in the first sense). Alas, (PA) is still undecidable and thoroughly fails to axiomatize T–through no fault of Peano's, first-order logic is too weak for this purpose.

Some decidable theories are:

- $\mathsf{Th}(f)$, the theory of a single function
- Presburger arithmetic
- Skolem arithmetic
- Abelian groups
- Boolean algebras
- Algebraically closed fields
- Real closed fields

On the other hand, the following are undecidable:

- $\mathsf{Th}(f, g, \approx)$
- $\mathsf{Th}(R^2, \approx)$
- Arithmetic (plus and times)
- Robinson arithmetic
- Algebraic structures (semigroups, groups, finite groups, rings, fields)

Since axiom systems are typically small, the problem arises whether they contain enough information to really pin down the objects under discussion. Ideally we would like the axioms to settle all possible questions (at least questions that can be phrased in the particular language we have chosen).

Definition 7.6.26 *A set of sentences Γ is complete if for every sentence φ of the language we have either $\Gamma \models \varphi$ or $\Gamma \models \neg\varphi$.*

A cheap example for a complete theory is $\mathsf{Th}(\mathfrak{C})$. Completeness is a rather strong property; for example, Γ has to pin down statements about cardinality. Most standard axiom systems (such as the group axioms) are indeed incomplete.

Example 7.6.7 The theory of algebraically closed fields of characteristic 0 is complete, as is the theory of real closed fields.

Another way of thinking about completeness is to consider the models, they are all very similar in the following sense:

Definition 7.6.27 *Two structures $\mathcal{A}$ and $\mathcal{B}$ are elementarily equivalent if* $\mathsf{Th}(\mathcal{A}) = \mathsf{Th}(\mathcal{B})$.

As always, we assume that the structures have the same signature. Thus, as long as we restrict ourselves to the limitations of first-order logic we cannot distinguish between the two structures. But note that this notion is much weaker than the existence of an isomorphism between the structures.

Proposition 7.6.1 *Γ is complete if, and only if, any two models of Γ are elementarily equivalent.*

Here are some complete theories:

- Presburger arithmetic.
- Dense linear orders.
- Algebraically closed fields of a given characteristic.
- Real closed fields.
- Tarski's axioms for Euclidean geometry
- Any $\aleph_0$-categorical theory.

7.6.6 Derivations and Proofs

So far, we have a solid syntax for a language, and a natural semantics for the formulae in this language. Alas, to really get mileage out of first-order logic, we need a better grip on $\mathsf{Th}(\Gamma)$: we need to be able to find the consequences φ of Γ directly, without complete knowledge of all the models. We would like to argue solely from the axioms themselves, without any reference to the possibly highly complicated assortment of models. For example, we know informally how to derive the assertion

$$\forall\, x, y\, ((x * y)^{-1} = y^{-1} * x^{-1})$$

directly from the group axioms, using some basic equational reasoning and "obvious" rules for quantification. We need to formalize these rules and develop a notion of provability

$$\Gamma \vdash \varphi$$

analogous to provability in propositional logic.

7.6.7 Natural Deduction for First-Order Logic

There are many different ways of doing this, for the sake of brevity let us only indicate how the *natural deduction system* from our discussion of propositional logic can be expanded to first-order logic. Needless to say, we can retain the propositional rules, they are still sound in the new context. To extend natural deduction from propositional logic to first-order logic, we need to add rules for quantifiers. Intuitively, that's not hard: we wish to go from witnesses to quantifiers, and back:

$$\frac{\phi(t)}{\exists\, x\, \phi(x)}\ (\exists e) \qquad\qquad \frac{\exists\, x\, \phi(x)}{\phi(c)}\ (\exists i)$$

$$\frac{\phi(c)}{\forall\, x\, \phi(x)}\ (\forall i) \qquad\qquad \frac{\forall\, x\, \phi(x)}{\phi(t)}\ (\forall e)$$

where x is a variable, c a constant, and t a term. Alas, while these rules are correct in spirit, as stated they are absolutely not sound. Suppose we adopt the unconstrained quantifier rules from above. Then we can perform a derivation along the following lines (this is Hilbert style).

$$\begin{array}{ll}
\forall\, x\, \exists\, y\, (x < y) & \text{premise} \\
\exists\, y\, (c < y) & (\forall e) \\
(c < d) & (\exists e) \\
\forall\, x\, (x < d) & (\forall i) \\
\exists\, y\, \forall\, x\, (x < y) & (\exists i)
\end{array}$$

A disaster, this is the wrong direction of the valid implication $\exists\, x\, \forall\, y\, \varphi(x, y) \Rightarrow \forall\, y\, \exists\, x\, \varphi(x, y)$. The problem is that the “constant” d really depends on c, but our rules are too hamfisted to take this into account. Or consider the “proof”

$$\frac{\dfrac{\forall\, x\, Pxx}{Paa}\,(\forall e)}{\forall\, x\, Pax}\,(\forall i)$$

clearly a wrong inference. To address these and similar problems, one has to add some technical conditions to the quantifier rules that eliminate the faulty derivation from above.

- For $(\forall i)$ we insist that c is “fresh”: it must not occur in any undischarged assumptions in the derivation of $\varphi(c)$ nor in the conclusion $\forall\, x\, \phi(x)$ itself.
- For $(\exists e)$ again we insist on a fresh constant c:

$$\frac{\exists\, x\, \phi(x) \qquad \begin{array}{c}[\phi(c)] \\ \vdots \\ \chi\end{array}}{\chi}\,(\exists e)$$

- For $(\forall e)$ and $(\exists i)$ we insist that term t is substitutable for x in φ: no variable in t becomes bound as a result of the replacement process.

We won’t belabor the details, though they are extremely important when one tries to actually implement a deduction system in first-order logic. Needless to say, they are also critical in a careful correctness proof.

Given a collection of formulae (assumptions, axioms), we can now consider proofs that are allowed to use formulae in Γ without any further justification. Again, these proofs are purely syntactic objects, without any reference to model theory.

Definition 7.6.28 (Syntactic Consequence) *The set of all provable theorems of a collection of sentences in first-order logic is called its (syntactic) theory or its set of consequences.*

Notation: $\mathsf{Cn}(\Gamma) = \{\, \varphi \mid \Gamma \vdash \varphi \,\}$

Note that on the face of it, $\mathsf{Cn}(\Gamma)$ is a much less complicated notion than $\mathsf{Th}(\Gamma)$: proofs are merely syntactic objects (finite, discrete data structures) whereas truth and semantic consequence depend on actual structures. There can be a great many such structures and they are possibly infinite, so some amount of set theory is required to even talk about them. It can be quite daunting to determine what they look like. As a consequence, it is far from clear that there exists a set of non-logical axioms, which, together with the logical axioms and the rules of inference, would achieve the goal of matching soundness and completeness:

- Soundness: $\Gamma \vdash \varphi \Rightarrow \Gamma \models \varphi$, and

- Completeness: $\Gamma \models \varphi \Rightarrow \Gamma \vdash \varphi$.

Soundness can be established by careful analysis of the axioms and rules, but note that this is much more challenging than in the propositional logic case because of the quantifier rules. Completeness is harder yet, the first proof of completeness is due to Gödel in 1930.

Theorem 7.6.1 (Completeness Theorem for First-Order Logic) *There is a formal system for first-order logic that is sound and complete. Thus, a sentence of a theory T is a theorem of T if, and only if, it is valid in T: $T \models \varphi$ iff $T \vdash \varphi$.*

We will indicate the proof in a moment. We will need to generalize the deduction theorem from propositional logic: as far as implications are concerned, first-order logic behaves just the same.

Theorem 7.6.2 (Deduction Theorem) *Let Γ, φ, ψ be sentences. Then $\Gamma, \varphi \vdash \psi$ if, and only if, $\Gamma \vdash \varphi \Rightarrow \psi$.*

So if there is a proof $\Gamma \vdash \varphi$ between sentences, then

$$\vdash \psi_1 \wedge \psi_2 \ldots \wedge \psi_n \to \varphi$$

for some ψ_i in Γ. But note the condition on sentences here, free variables cause problems. For example, $P(x) \vdash P(y)$ but $\vdash P(x) \Rightarrow P(y)$ is false.

Could it ever happen that some set of axioms Γ has no models at all? Sure, $\Gamma = \{a \approx b, b \approx c, a \not\approx c\}$ or, more radically, $\Gamma = \{\bot\}$ will do. Of course, no one would use these formulae directly as axioms. But, it might happen that $\bot$ is provable from an ill-chosen set of axioms: there will be no models, but that may not at all be obvious from the axioms.

Definition 7.6.29 (Consistency) *Γ is inconsistent if $\Gamma \vdash \bot$, and consistent otherwise.*

Note that in an inconsistent theory all sentences are provable by the principle of explosion. Clearly, any inconsistent set of axioms has no models; the properties we were trying to pin down contradict each other. Surprisingly, this is the only thing that can go wrong.

Theorem 7.6.3 (Completeness Theorem II) *A theory has a model if, and only if, it is consistent: $\mathsf{Th}(T) = \mathsf{Cn}(T)$.*

It is not hard to see that the second version implies the first:

$$\begin{aligned} T \models \varphi &\Leftrightarrow\ T + \neg\varphi \text{ has no model} \\ &\Leftrightarrow_{II}\ T + \neg\varphi \text{ inconsistent} \\ &\Leftrightarrow\ T + \neg\varphi \vdash \bot \\ &\Leftrightarrow_{DT}\ T \vdash \varphi \end{aligned}$$

Sketch of proof. It remains to establish the second version of the deduction theorem. Without going into details, the idea of a proof by L. Henkin is to

- add many constants c to the language, and
- extend T to a certain complete set Γ of sentences.

The constants are used to make sure that $\exists x\, \varphi(x) \in \Gamma$ implies $\varphi(c) \in \Gamma$ for some c. Then we can construct a term model for Γ: the collection of all terms, modulo equality provable in Γ, turns out to be a model of Γ and thus of T. □

The fact that consistency is the only requirement for the existence of a complete extension of an axiom system is really a separate theorem by Tarski.

Theorem 7.6.4 (Tarski) *Every consistent set of sentences T can be extended to a complete set of sentences $T' \supseteq T$.*

Of course, T' might be very complicated. For example, we might lose decidability.

7.6.8 Compactness

Another crucial property of first-order theories is based on the following simple observation: any single derivation in first-order logic uses only finitely many formulae. One can certainly imagine stronger systems where a single proof involves infinitely many formulae, but in first-order logic this is not the case.

Hence, Γ inconsistent means that there is a finite subset Γ_0 of Γ that is already inconsistent. By completeness we get the following surprising consequence:

Theorem 7.6.5 (Compactness theorem) *Γ has a model if, and only if, every finite subset Γ_0 of Γ has a model.*

This is positively wild, because infinitely many axioms can be used to construct weird conditions, when every finite subset is perfectly harmless.

Lemma 7.6.1 *The class of all finite structures (of some signature) is not axiomatizable.*

For assume that Γ is some axiom system that characterizes finite structures. Hence Γ has arbitrarily large finite models. But then Γ must have an infinite model. To see this, add new constants c_i, $i \in \mathbb{N}$, to the language and add new axioms

$$c_i \not\approx c_j \qquad \text{for } i < j$$

to Γ to obtain an extension Γ'. Every finite subset of Γ' has a model; by compactness Γ' has a model which must be infinite by choice of the additional axioms.

Lemma 7.6.2 *The class of all infinite structures (of some signature) is not finitely axiomatizable.*

For otherwise the class of finite structures would also be elementary. But note that the class of infinite structures is first-order definable: we need infinitely many sentences of the form "there are at least n elements".

$$\mathsf{Least}_n = \exists\, x_1, \ldots, x_n \,(x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge \ldots \wedge x_{n-1} \neq x_n)$$

Dedekind-Peano Arithmetic (PA)

Here is an example of a hugely important axiom system due to Guiseppe Peano in 1889 that describes the salient properties of the structure of natural numbers $\mathcal{N} = \langle \mathbb{N}; +, \cdot, S, 0, 1, < \rangle$, signature $(2, 2, 1, 0, 0; 2)$, with the usual arithmetic operations and order. Actually, R. Dedekind, Gauss's last student, developed the same system a year before Peano, but never published.

Dedekind-Peano Axioms

successor	$S(x) \neq 0$	$S(x) \approx S(y) \Rightarrow x \approx y$
addition	$x + 0 \approx x$	$x + S(y) \approx S(x + y)$
multiplication	$x \cdot 0 \approx 0$	$x \cdot S(y) \approx (x \cdot y) + x$
order	$\neg(x < 0)$	$x < S(y) \Leftrightarrow x \approx y \vee x < y$

Arithmetic operations alone are not enough, we are missing one essential feature of the natural numbers: induction. To capture induction we add the Induction Axiom:

$$\varphi(0) \wedge \forall\, x\, \big(\varphi(x) \Rightarrow \varphi(S(x))\big) \Rightarrow \forall\, x\, \varphi(x)$$

Strictly speaking, this is not an axiom but an axiom schema: we get one axiom for each choice of φ. At any rate, (PA) is a very succinct representation of the essential features of arithmetic. Here is a sample proof.

Lemma 7.6.3 (PA) *proves that* $\forall\, x\, (0 + x \approx x)$

Proof. Consider the formula $\varphi(x) \equiv (0 + x \approx x)$. Then $\varphi(0)$ is the first addition axiom (more precisely, the substitution image $x \mapsto 0$). Now assume $\varphi(x)$. Then by the second addition axiom

$$0 + S(x) \approx S(0 + x) \approx S(x)$$

Hence we have shown $\varphi(0) \wedge \forall\, x\, (\varphi(x) \Rightarrow \varphi(S(x)))$. By the Induction Axiom and modus ponens we get $\forall\, x\, \varphi(x)$. □

Axiomatizing Arithmetic

One would hope that the Peano axioms allow one to derive all true statements of arithmetic. But note that there is a little problem here: the true statements of arithmetic are the theory of a single structure:

$$\mathsf{Th}(\mathfrak{N}) = \{\, \varphi \mid \mathfrak{N} \models \varphi \,\}$$

But all we get from our notion of provability is the statements that hold in all models of (PA) . That would be fine if $\mathfrak{N}$ were the only such model (up to isomorphism or even elementary equivalence). Unfortunately, there are others as we shall see shortly. If we think of (PA) as pinning down axiomatically the properties of the natural numbers, here is some very bad news: $\mathfrak{N}$ is not the only model of (PA) .

To see why, first note that any model $\mathcal{A}$ of (PA) must have an isomorphic copy of the natural numbers as an initial segment. For simplicity, let us assume $\mathbb{N} \subseteq \mathcal{A}$ and consider the possibility that $a \in \mathcal{A} - \mathbb{N}$. From the axioms, it follows that $a + 1$, $a + 2$, ... are all in $\mathcal{A}$ (we should be using numerals). Similarly, there must be $a - 1$, $a - 2$, ..., so we wind up with a whole copy of Z containing a. It is easy to see that all elements in this copy are larger than all elements in $\mathbb{N}$. Moreover, $2a$ must span another, larger copy of $\mathbb{Z}$, $3a$ another one, and so on. Consider two even elements $a < b \in \mathcal{A} - \mathbb{N}$, in two different copies of $\mathbb{Z}$. Then $c = (a + b)/2$ must span another copy of $\mathbb{Z}$, wedged between the copies of a and b. So at the very least, $\mathcal{A}$ looks like $\mathbb{N} + \mathbb{Q}\mathbb{Z}$. It is not clear that we can define proper arithmetic operations on a structure like this. Much less is it clear that the induction axiom can be made to work: clearly, the $\mathbb{Q}\mathbb{Z}$ part has no least element. Completeness gets us around all these problems. We introduce a new constant c and add the following new axioms to (PA):

$$\underline{0} < c,\ \underline{1} < c,\ \underline{2} < c, \ldots, \underline{n} < c, \ldots$$

Call the new set of axioms (PA^∞).

Claim 7.6.1 (PA^∞) has a model, a so-called non-standard model of arithmetic.

Proof. To see this, exploit compactness. Any finite subset $\varGamma$ of (PA^∞) contains only finitely many of the new axioms. So there is a largest n such that $\underline{n} < c \in \varGamma$. But then we can simply interpret c as $n+1$ in the standard model $\mathfrak{N}$, done. □

So let $\mathcal{A}$ be a model of (PA^∞), a structure of the kind just described. This structure contains an "infinitely large" element $c^{\mathcal{A}}$: since c is a constant in the language it is interpreted by some element of $\mathcal{A}$, and it follows from the extra axioms that $\mathcal{A} \models \underline{n} < c$ for all $n \geq 0$. Of course, $c^{\mathcal{A}}$ is not infinitely large from the perspective of $\mathcal{A}$, it's just a "natural number" there. The reason the induction axiom still works is essentially that $\mathbb{N} \subseteq \mathcal{A}$ cannot be defined by a first-order formula over $\mathcal{A}$. Of course, it is a perfectly well-defined set from the view of set theory.

Note that the atomic diagram $\mathsf{diag}(\mathfrak{N})$ of the natural numbers is very simple (atomic, we are not making any claims about the complete diagram). A typical formula in the atomic diagram is $\underline{2} + \underline{3} \approx \underline{5}$ or $\neg \underline{3} < \underline{2}$. Certainly $\mathsf{diag}(\mathfrak{N})$, given some reasonable encoding, is decidable, even primitive recursive at a very low level of the hierarchy. For non-standard models, diagrams are much more complicated.

Theorem 7.6.6 (Tennenbaum, 1960) *No non-standard model $\mathcal{A}$ of* (PA) *is computable: the atomic diagram of $\mathcal{A}$ is always undecidable.*

Again, we are dealing with the atomic diagram here, not quantified sentences. The problem comes from prime divisors of integers in the model:

$$\{\, n \in \mathbb{N} \mid \mathcal{A} \models p_n \text{ divides } a \,\}$$

If a here is non-standard, these sets can be very complicated.

Non-standard models of Peano arithmetic may seem like a mere nuisance, vaguely interesting but essentially useless, in particular in view of Tennenbaum's theorem. A. Robinson realized in 1960, though, that this type of unintended model provides a perfect framework for analysis: one can construct strange models of the reals that contain infinitesimal elements. As a consequence, there is no need for limits in such a model. In essence, differentiation is just a quotient operation $\frac{f(x+h)-f(x)}{h}$ and integrals are just sums. The drawback is, of course, that someone trying to learn or apply calculus this way must already have a solid background in logic—perhaps unsurprisingly, non-standard analysis never really took off.

7.6.9 Incompleteness and Undecidability

So how about the completeness of (PA): are the axioms powerful enough to establish every true statement of arithmetic, at least those that can be expressed in first-order logic? One can check that quite a bit of elementary number theory can in fact be expressed in just (PA); in fact, it is quite a challenge to come up with a true assertion that cannot be proved in (PA) . And yet, Kurt Gödel showed in 1931 that Peano's axioms are too weak: there are facts about the natural numbers, expressible in first-order logic, that cannot be proven in (PA), nor in any other reasonable axiom system for arithmetic. Needless to say, the highly undecidable $\mathsf{Th}(\mathfrak{N})$ is not a good choice of axioms for arithmetic.

Theorem 7.6.7 (Gödel 1931) *Suppose* (PA) *is consistent. There is a sentence of arithmetic such that* (PA) *neither proves nor refutes this sentence.*

Of course, $\mathfrak{N}$ must be a model of either this sentence φ or its negation $\neg\varphi$, the problem is that (PA) is not strong enough to determine which is the case. One might still hope that such sentences are horribly complicated, but in fact a single universal quantifier suffices. Gödel's second Incompleteness theorem pins down one such sentence explicitly.

Theorem 7.6.8 (Gödel 1931)
If (PA) *is consistent, then its consistency cannot be proven in* (PA)*.*

This statement bears a bit of explanation. Gödel realized that first-order logic can be arithmetized: one can code formulae as natural numbers, and express all the standard syntactic operations on them as easily computable functions. A proof is just a sequence of formulae, and can also represented as a sequence number. Testing whether a number represents a valid proof is easily decidable and thus expressible in arithmetic. But then consistency can be written down as a universally quantified formula of arithmetic, something like

$$\mathsf{Cons} \equiv \neg\exists\, x\, (\mathsf{prf}(x, \ulcorner\bot\urcorner))$$

where $\mathsf{prf}(x, y)$ expresses the fact the x codes a proof for y, and $\ulcorner\bot\urcorner$ denotes the code number of $\bot$. Then Cons is true, but cannot be proven.

This does not mean that we cannot prove (PA) to be consistent, we just cannot do it inside of the system. For example, transfinite induction to ε_0 suffices. Or, you can just believe that $\mathfrak{N}$ is a model based on intuition. Still, the missing theorems tend to be a bit weird, they don't seem to matter much in "real life", that is, in applications of number theory. Note the hedge here, in 1975 Paris and Harrington showed how to construct statements of finite combinatorics that are true but not provable in (PA).

How does this relate to computer science? Suppose P is a program in some standard programming language that computes a numerical function $\widehat{P} : \mathbb{N} \to \mathbb{N}$. As always write $\underline{n}$ for the numeral representing $n \in \mathbb{N}$ in (PA) . Then there is a formula $\phi(x, y)$ of arithmetic such that

- $\widehat{P}(m) = n$ implies (PA) $\vdash \phi(\underline{m}, \underline{n})$.
- (PA) $\vdash \phi(x, y) \wedge \phi(x, z) \Rightarrow y \approx z$.

But even though $\widehat{P}$ is total, (PA) may well not be powerful enough to prove it: in general

$$(\mathsf{PA}) \nvdash \forall\, x\, \exists\, y\, \phi(x, y).$$

If we cannot prove all true statements of arithmetic, can we at least give a decision algorithm for them? It is a direct consequence of the way proof systems are built that all formulae provable from a decidable set of axioms are semidecidable: we can systematically enumerate all axioms and all proofs in the system based on these axioms – in principle, efficiency is not a consideration here.

Theorem 7.6.9 *Let Γ be decidable. Then $\mathsf{Cn}(\Gamma)$ is semidecidable.*

So for $\mathsf{Cn}(\Gamma)$ to be decidable we only need a semi-algorithm for the non-theorems, the sentences that do not follow from Γ. Alas, there is no obvious general way to obtain such an algorithm. But sometimes it happens that the set of axioms is very rich and "knows" facts about non-theorems in the following sense:

$$\Gamma \nvdash \varphi \qquad \text{implies} \qquad \Gamma \vdash \neg\varphi$$

Lemma 7.6.4 *Suppose T is a complete and axiomatizable theory. Then T is decidable.*

Proof. It suffices to show that the complement of T is semidecidable. But $\varphi \notin T \iff \neg\varphi \in T$, done. □

Unfortunately, completeness of a theory is not all that easy to check in general. Here is one valuable test. Let's assume throughout that the language we are using is countable. A set of sentences Γ is κ-categorical for some cardinal $\kappa \geq \aleph_0$ if all models of Γ of size κ are isomorphic. Here is the main result relating to categoricity.

Theorem 7.6.10 (Morley Categoricity Theorem) *Let T be a complete theory in a countable language with infinite models. T is κ-categorical for some uncountable κ if and only if T is λ-categorical for every uncountable λ.*

Theorem 7.6.11 (Los-Vaught Test) *Let T be a theory with no finite models. If T is κ-categorical for some cardinal $\kappa \geq \aleph_0$, then T is complete.*

Let (ACF_0) be the theory of algebraically closed fields of characteristic 0. Clearly, all models of (ACF_0) are infinite. Suppose $\mathcal{A}$ is a model of (ACF_0) of cardinality $\kappa = |\mathbb{R}|$. Then $\mathcal{A}$ must be an extension field of $\mathbb{Q}$. The transcendence degree of $\mathcal{A}$ is κ. But then it is easy to construct an isomorphism between any two such models. Hence (ACF_0) is complete.

Arithmetic

Peano arithmetic is not complete and indeed undecidable.

Theorem 7.6.12 (K. Gödel 1931, J.B. Rosser, 1936)
Peano arithmetic is undecidable.

Naturally one would like to understand the boundaries of undecidability in arithmetic. In particular, is there a small subsystem of Peano arithmetic that is still undecidable? Are there subsystems that are decidable?

As it turns out, we can avoid undecidability if we remove multiplication. Presburger arithmetic uses the language $\mathcal{L}(+,-,0,1;<)$ of signature $(2,2,0,0;2)$ and axiomatizes these operations over the integers $\mathbb{Z}$.

Theorem 7.6.13 (M. Presburger, 1929)
Presburger arithmetic is decidable.

add proof

But note that the best algorithm for Presburger arithmetic is not practical, it is triple exponential. In fact, even for quantifier-free formulae the problem is $\mathbb{NP}$-hard. An analogous theorem by Skolem shows that one can similarly remove addition, and only retain multiplication: the resulting Skolem arithmetic is similarly decidable.

System Q

How do we construct a small undecidable subsystem? We remove the induction axiom and retain axioms for successor, addition and multiplication.

$$\begin{array}{rll} \text{successor} & S(x) \not\approx 0 & S(x) \approx S(y) \to x \approx y \\ \text{addition} & x + 0 \approx x & x + S(y) \approx S(x+y) \\ \text{multiplication} & x \cdot 0 \approx 0 & x \cdot S(y) \approx (x \cdot y) + x \end{array}$$

Lastly, we add a weak induction axiom:

$$x \not\approx 0 \Rightarrow \exists\, y\, (x \approx S(y))$$

MISSING

Decidable Theories

Theorem 7.6.14 *The theory of Abelian groups is decidable. But the theory of general groups is undecidable.*

This result is rather remarkable since the axioms are nearly identical: only one commutativity axiom is needed to enforce decidability. This decidability result reflects nicely the fact that Abelian groups are much less complicated than general groups. In general, many familiar structures in algebra are undecidable: rings, commutative rings, integral domains, fields, fields of characteristic 0. Boolean algebra, on the other hand, is decidable.

For important structures $\mathcal{A}$ such as the rationals and reals, the question arises whether $\mathsf{Th}(\mathcal{A})$ is decidable (regardless of attempts at axiomatization).

Theorem 7.6.15 (J. Robinson, 1948) *The theory of the rationals with addition and multiplication is undecidable.*

This is in sharp contrast to a famous theorem by Tarski concerning the real numbers.

Theorem 7.6.16 (A. Tarski, 1948) *The theory of the reals with addition and multiplication is decidable.*

As a consequence, basic geometry is decidable. The theorem is proved by a very interesting technique that provides a direct decision algorithm: quantifier elimination. Tarski's original method was highly inefficient, though (not bounded by a stack of exponentials).

Definition 7.6.30 (Quantifier Elimination) *A theory T admits quantifier elimination if for every formula φ there is a quantifier-free formula φ_0 such that $T \vdash \varphi \leftrightarrow \varphi_0$.*

It actually suffices to show that one can eliminate existential quantifiers in a formula

$$\exists\, x\, (A_1 \land A_2 \land \ldots \land A_n)$$

where all the A_i are literals (atomic formulae or negations thereof).

Note that on the face of it this may seem utterly hopeless: the existence of x may depend on the values of the free variables. It is not clear how to capture this dependence without quantifiers.

As an example, here is how quantifier elimination might work for the theory of a single successor function. Consider the theory of the simple structure $\mathfrak{N}_S = \langle \mathbb{N}, S, 0 \rangle$. We can give an axiom system Γ for this theory as follows:

$$\begin{aligned} S(x) &\not\approx 0 \\ S(x) \approx S(y) &\Rightarrow x \approx y \\ y \not\approx 0 &\Rightarrow \exists\, x\, S(x) \approx y \\ S^n(x) &\not\approx x \qquad\qquad \text{schema } n \geq 1 \end{aligned}$$

It is quite obvious that $\mathsf{Cn}(\Gamma) \subseteq \mathsf{Th}(\mathfrak{N}_S)$, but because of non-standard models the opposite direction is far from clear.

What does an arbitrary, non-standard model $\mathcal{A}$ of Γ look like? Intuitively, similar to the situation with models of (PA), we must have

$$A = \mathbb{N} \cup \mathbb{Z} \cup \mathbb{Z} \cup \mathbb{Z} \ldots$$

where the unions are meant to be disjoint: There is one copy of $\mathbb{N}$, the standard model, plus a number of copies of $\mathbb{Z}$. We can make this precise by showing that $\exists\, n \geq 0\, (S^n(x) = y \vee S^n(y) = x)$ is an equivalence relation on A with the equivalence classes being isomorphic copies of $\mathbb{Z}$. At any rate, the cardinality of A is $\aleph_0 + \aleph_0 \kappa$ where κ is the number of copies of $\mathbb{Z}$. But then any two models of Γ with the same κ must be isomorphic. Hence, by Los-Vaught, $\mathsf{Cn}(\Gamma) = \mathsf{Th}(\mathfrak{N}_S)$ is complete and thus decidable.

To get a decision algorithm, we show that quantifier elimination holds. Consider a generic formula $\exists\, x\, (A_1 \wedge A_2 \wedge \ldots \wedge A_n)$ as above. We may safely assume that x occurs in each A_i. Now the positive atomic formulae must be of the form $S^n(x) \approx t$. If $t \approx S^m(x)$ the literal can easily be replaced by $\bot$ or $\top$. So we only have to deal with $S^n(x) \approx S^m(y)$ and $S^n(x) \approx 0$. If all the literals are negative, then we can replace the whole formula by $\top$. So suppose $A_1 = S^n(x) \approx t$ is positive. But then we can replace A_1 by

$$t \not\approx 0 \wedge t \not\approx S(0) \wedge \ldots \wedge t \not\approx S^{n-1}(0).$$

Any other literal $S^r(x) \approx z$ is replaced by $S^r(t) \approx S^m(z)$ (and likewise for negative ones). But then none of the literals contains x any more, so we can drop the quantifier. Note that if we apply our elimination process to a sentence we wind up with a quantifier-free sentence, which must be a Boolean combination of pieces $S^m(0) \approx S^n(0)$.

In fact, even if we disregard attempts to axiomatically describe some reasonable class of structures and only consider the underlying logic itself we run into undecidability.

Theorem 7.6.17 *First-order logic is undecidable: it is undecidable whether a sentence in first-order logic is derivable from the empty set of axioms.*

Theorem 7.6.18 *Satisfiability in first-order logic is undecidable: it is undecidable whether a sentence in first-order logic has a model.*

The wording of these theorems is a bit careless: one should say that there is a particular first-order language L for which the various claims hold: one has to be a bit careful about having enough function and relation symbols around. E.g., one binary relation symbol suffices, as does one unary and one ternary function symbol. Still, for any underlying language that is strong enough to be useful, provability and satisfiability are undecidable.

Exercises

Exercise 7.6.1 Show that there is an infinite field of characteristic 2. By contrast, give an algebraic construction of such a field.

Exercise 7.6.2 Fill in the details in the proof for lemma 7.6.1.

Exercise 7.6.3 Write down axioms for fields and for ordered fields.

Exercise 7.6.4 Write down axioms for finite fields, for infinite fields and for fields of characteristic p (both for $p = 0$ and p prime).

Exercise 7.6.5 How would you axiomatize vector spaces where one has to deal with a field and the actual vector space at the same time?

Exercise 7.6.6 Axiomatize the real numbers as best you can, starting from the field axioms. Repeat for complex numbers.

Exercise 7.6.7 Try to write down axioms for a stack of elements of some ground type. Repeat for queues, lists and trees.

Exercise 7.6.8 Prove the following assertions in (PA).

$$\forall\, x, y, z\ (x + (y + z) \approx (x + y) + z)$$
$$\forall\, x, y\ (x + y \approx y + x)$$

Conclude that $\langle \mathbb{N}; +, 0\rangle$ is a commutative monoid.

Exercise 7.6.9 Define the GCD and describe the Euclidean algorithm in (PA).

Exercise 7.6.10 Prove that there are infinitely many primes in (PA).

Exercise 7.6.11 Show that if Γ has arbitrarily large finite models then Γ must have an infinite model.

Conclude that there are infinite fields of characteristic $p > 0$.

Exercise 7.6.12 Show that every partial order can be embedded into a total order.

Exercise 7.6.13 Show that there is no FO set of axioms Γ that characterizes finiteness in the sense that $\mathcal{A} \models \Gamma$ if, and only if, $\mathcal{A}$ is infinite.

Exercise 7.6.14 Explain what would happen if we adopt the equality formulae as axioms for an otherwise undistinguished binary relation symbol $=$. Describe the structures that satisfy these axioms.

Exercise 7.6.15 Verify that the first three quantifier manipulation formulae are true, and come up with a counterexample for the last one.

Exercise 7.6.16 Use a total order to produce another formulae in first-order logic that forces the ground set to be infinite.

Exercise 7.6.17 In the theory of fields, write $\underline{n}$ for the ground term $\underbrace{1 + 1 + \ldots + 1}_{n}$, $n \geq 1$ and $\chi_n = \underline{n} \approx 0$, $n \geq 2$. What happens if we adjoin an axiom χ_n for n prime or composite? How about all axioms of the form $\neg\chi_n$?

Exercise 7.6.18 How can one axiomatize the theory of algebraically closed fields?

Exercise 7.6.19 Show the first version of the completeness theorem implies the second.

Exercise 7.6.20 What might a complete extension of the theory of fields look like? Note that it has to settle on a value for 0^{-1}.

Exercise 7.6.21 Fill in the details of the argument for algebraically closed fields.

Exercise 7.6.22 Show that the theory of atomless Boolean algebras is complete.

Exercise 7.6.23 Show that the theory of $\langle \mathbb{N}, S\rangle$ is complete.

Exercise 7.6.24 Prove that full quantifier elimination follows from the weaker form.

Exercise 7.6.25 Fill in all the details in the quantifier elimination argument for the theory of one successor.

Part II

Computability

Eight

Models of Computation

In this chapter we will answer the question: What is computability? In the age of ubiquitous digital computers, this may seem like a straightforward challenge: computable means that there is an algorithm, a sort of effective, mechanical procedure that solves the problem in question. Of course, we now have to explain what exactly we mean by an algorithm. It is tempting to handle the latter task by enumerating a few well-known examples of algorithms such as Euclid's method to to compute the greatest common divisor of two integers. In particular for anyone with some amount of experience in programming digital computers, this sort of explanation of computability is quite convincing and entirely natural.

Unfortunately, the computation-via-algorithms approach is insufficient for our purposes. To be sure, it works reasonably well when it comes to convincing oneself that some particular problem has a computable solution, but we are also interested in the opposite direction: we want to establish unsolvability results that rule out the possibility of any computational solution whatsoever. For example, we would like to be able to show that there is no algorithm that can determine whether a multivariate polynomial with integer coefficients has a root over the integers. We need definitions that are perfectly precise and, preferably, not too unwieldy. To this end we rely on abstract, mathematical models of computation that are still well connected to "real-world" computations. In other words, if we can establish unsolvability in our abstract model, it follows immediately that the problem in question cannot have a practical computational solution, either. In fact, we will see that much more is true: feasible computation is a rather tiny fragment of abstract computation. There are two main types of models of computation: machine models and program models.

Machine-based models of computation define a particular class of machine, some well-known examples being Turing machines, register machines and finite state machines. These models are somewhat related to digital computers, but are much simplified and make no attempt to account for any of the details of an actual digital computer. As a consequence, they are easily amenable to formalization and can be treated with methods from discrete mathematics. For example, for finite state machines, a computation is just a path in an edge-labeled directed graph. One important feature of these machine models is that they establish a direct link to physical computation, the machines can easily be realized by actual physical devices—at least if one ignores constraints relating to time, space and energy. If one tries to take these practical constraints into account, one is lead to complexity theory, the attempt to analyze feasible computation. Somewhat surprisingly, complexity theory appears to be more difficult than classical computability theory, or recursion theory to use older terminology. The infamous $\mathbb{P}$ versus $\mathbb{NP}$ problem is a perfect example of this situation.

For program-based computability it is important to understand that the corresponding programming languages are mathematical in nature and have little in common with practical programming languages such as C or Haskell. For example, following Herbrand and Gödel, one can define computability solely in terms of recursive equations over the natural numbers. Computing the value of a function in this model comes down to solving equations in the algebraic sense. One the other hand, there are models such as loop programs or while programs that are fairly close to actual programming languages, at least if one restricts these languages to operate only on natural numbers. Unlike with practical languages, the semantics of these models is fairly straightforward and can be defined with precision, without having to worry about compilers, runtimes, operating systems and the like.

In the classical theory one is usually content to explore computation on the natural numbers, more complicated data structures such as lists, trees, graphs or hash tables play no role here. At first glance, this may seem like a fatal flaw: how can the classical theory help understand practical computation? As it turns out, if one ignores efficiency issues, there is no problem whatsoever: one can express all these data structures solely in terms of natural numbers, using standard tricks in coding theory. In particular a non-computability result easily carries over the world of actual computation. For complexity theory, though, one needs to be much more careful. More on this later.

This explanation of computability is very helpful at the intuitive level, but our goal is to establish rigorous results, and to that end we need to have a clear, axiomatic setting at our disposal. In particular the limits of computability cannot be discussed without a formal framework: if we want to assert that certain problems fail to have a computable solution, we need to be able to delineate clearly what the potential solutions would look like. With a view towards the world of realizable computation we could attempt to axiomatize an actual digital computer, but this task turns out to be hopelessly complicated; it is preferable to abstract away the constraints of an actual physical machine and home in on some kind of "ideal computer." To be sure, the connection between physics, engineering and computation is a fascinating topic, but we will steer clear of it for the most part. Similarly, defining the semantics of a real-world programming language and operating system, that would allow us to execute algorithms on our digital computer, is hugely complicated and requires significant mathematical machinery.

To avoid these difficulties, we will initially focus on computable arithmetic functions, that is, functions of the form $f : \mathbb{N}^k \to \mathbb{N}$. Many examples of such functions are quite familiar: addition, multiplication, exponentiation, greatest common divisor, nth prime, and so on. We will introduce a simple class of computable functions that captures all these examples, and much more, based on composition and a simple type of recursion, the so-called primitive recursive functions. Most functions in, say, a number theory book that are intuitively computable will turn out to be primitive recursive. Alas, ultimately our effort to axiomatize computability will force us to admit partial functions: every reasonable notion of computability has to accommodate the possibility that a particular function on a particular input may fail to be defined, the root of the infamous Halting Problem. To this end we introduce register machines (aka counter machines), an intuitively appealing type of machine that directly manipulates natural numbers and requires no coding when it comes to arithmetic functions. Another major advantage of register machines is that it is relatively straightforward to construct a universal machine, one that is capable of simulating all others.

Our restriction to natural numbers, as opposed to standard data structures such as lists, trees, graphs, is quite awkward from the perspective of real algorithms. Coding tricks make it possible to express all these concepts as natural numbers and are perfectly acceptable in the general theory of computation: for example, we can think of a string as a finite sequence of letters, each coded as an integer, and the whole sequence expressed as a sequence number; simplicity of presentation wins out over practicality in this case. However, coding adds unnecessary auxiliary computations to the actual core of the algorithm, a most undesirable feature in the world of low complexity. To avoid at least some of these problems we introduce Turing machines, devices that operate directly on strings rather than numbers, and we obtain functions of the form $f : \Sigma^\star \nrightarrow \Sigma^\star$. Turing machines are also of critical historical importance, and are currently the gold standard in the context of complexity theory.

One problem all machine models have in common is that it is relatively difficult to establish that a particular function is indeed computable. This is true in particular when one is not interested in a more or less vague proof-of-concept argument but an actual concrete construction. For example, building a Turing machine that tries to find counterexamples to the Riemann hypothesis is fairly difficult. Even reasonably tight bounds on the number of states such a machine would need to have is difficult. In this regard, simple programming languages are much more convenient. For example, all primitive recursive functions can be obtained nicely from a language containing elementary arithmetic operations and loop control structure. Of course, these languages have little in common with real programming languages such as C++ or ML, but they have entirely natural semantics and are better suited for our purposes.

Our last type of computational model comes directly from mathematics: we consider equational descriptions of arithmetic functions due to Herbrand and Gödel. These systems of equations are arguably the most compact and most readily understood descriptions of the function in question. The drawback is that one has to work a little harder to justify the assertion that a function described by some system of equations is indeed computable. Often it also requires a bit more effort to establish computability, compared to the simple programming language approach.

All these different models have their own advantages and drawbacks, taken all together they provide better insight into the nature of computation than each individual approach. It is somewhat surprising that all our different descriptive devices are equivalent in a strict technical sense: one obtains precisely the same class of computable functions, and there are simple translations back and forth between the different models. Computability turns out to be a rather robust and natural concept. It is not unreasonable to conclude that our various equivalent definitions of computability (and a number of others that we do not discuss such as the λ-calculus or Markov algorithms) correspond exactly to the intuitive concept of computability. This notion was first proposed by Church–and resisted by Gödel, who seems to have changed his mind only when Turing introduced his model.

One comment about terminology: computable functions were classically referred to as *recursive functions*, decidable sets were referred to as *recursive sets* and semidecidable ones as *recursively enumerable*. This terminology is arguably a bit misleading, recursion is just one particular aspect of computability, albeit a very important one. Recently there have been efforts to clean up terminology and ban the old terms. We will be entirely undogmatic about this, and use both versions, though we prefer recursive in the context of the classical theory, and computable with respect to complexity and the theory of algorithms.

8.1 Primitive Recursive Functions

As already mentioned, for the time being we are only interested it total arithmetic functions of the form $f : \mathbb{N}^k \to \mathbb{N}$. Thus all inputs are natural numbers and a single natural number is returned as output. We discuss several classes of obviously computable arithmetic functions, in increasing order of computational power: rudimentary functions, elementary functions and, lastly, primitive recursive functions. One construction principle underlies all these classes: we accept certain basic, simple functions as given and allow certain "admissible" operations to construct more complicated ones to determine a collection of functions with reasonable closure properties. This approach is quite familiar in analysis where a function on the reals might be defined by $f(x) = 1/(x^2 + 1)$. The expression on the right explains how to reduce the computation of f to other functions such as squaring, adding 1, and reciprocals and functional composition. As long as the basic functions are computable and the admissible operations preserve computability, we only obtain computable functions in this way. This is yet another important example of an inductively defined structure.

The definition of primitive recursive functions dates back to Dedekind's 1888 paper "Was sind und was sollen die Zahlen?" It is noteworthy that a precise axiomatic description of the natural numbers directly involves computational ideas. Dedekind's work is a perfect example of *logicism*: the attempt to reduce mathematics, in this case arithmetic, to logic. The success or failure of this idea depends very much on what exactly one considers to be the domain of logic (as opposed to genuine mathematical ideas, whatever those may be), but it is certainly an interesting approach.

Below we will introduce several classes of arithmetic functions that capture various notions of computability, beginning with very simple and obviously incomplete ones and leading up to the final, apparently correct, solution. Before we begin, though, it is convenient to establish a framework for the kind of collections of functions that make sense in our context.

8.1.1 Function Clones

The first issue we need to address is arity, the number of arguments of a function. When dealing with functions of multiple different arities, it is sometimes convenient to be able to express the arity as part of the function symbol. We will use a superscript (n), where $n \geq 0$, for this purpose:

$$f^{(n)} \qquad f \text{ is a function of arity } n.$$

Likewise, given a collection $\mathcal{C}$ of functions, $\mathcal{C}^{(n)}$ indicates all the functions in $\mathcal{C}$ of arity $n \geq 0$. Note that we allow nullary functions, functions with no inputs. Clearly, these functions are boring in the sense that their output is necessarily fixed, they are really just constants. For emphasis, we refer to them as hard constants. Still, we want to think about evaluating nullary functions and will adopt the notation $f()$ or $f(*)$[1].

The next concern is composition of functions. It is intuitively clear that computability is preserved under functional composition: if both f and g are computable, then so is $f \circ g$. In the world of programs this is simply sequential composition. The form of composition that works best for our purposes is the following: suppose we have functions $g_i : \mathbb{N}^m \to \mathbb{N}$ for $i = 1, \ldots, n$, $h : \mathbb{N}^n \to \mathbb{N}$. We can then define a new function $f : \mathbb{N}^m \to \mathbb{N}$ as follows:

$$f(\boldsymbol{x}) = h(g_1(\boldsymbol{x}), \ldots, g_n(\boldsymbol{x}))$$

We will write $\mathsf{Comp}[h, g_1, \ldots, g_n]$ or, in slightly more algebraic form, $h \circ (g_1, \ldots, g_n)$ for this new function. There are really two steps in this process: first, we bundle the n functions $g_i^{(m)}$ into a single function $G^{(m)} : \mathbb{N}^m \to \mathbb{N}^n$; second, we compose that function with $f^{(n)}$, in the ordinary way.

A careful inspection of examples like $h(g_1(x), g_2(x, y))$ shows that we also need a bit of bureaucracy to deal with variables. In terms of the corresponding functions, we want to have projections available, all maps of the form

$$\mathsf{P}^{(\mathsf{n})}_{\mathsf{i}} : A^n \to A \qquad \mathsf{P}^{(\mathsf{n})}_{\mathsf{i}}(x_1, \ldots, x_n) = x_i$$

[1] In the algebraic literature (as opposed to category theoretic work) one often disallows nullary functions and defines constants as a separate species. This is mostly a matter of taste, no major technical differences arise either way.

where $1 \leq i \leq n$. To see how projections help, consider the task of defining a ternary version of addition, assuming that we already have a binary version $\mathsf{add}^{(2)}(x, y)$. We would like to set $\mathsf{add}^{(3)}(x, y, z) = \mathsf{add}(x, \mathsf{add}(y, z))$, but that does not conform with our definition of composition. In the presence of projections, this is not a problem:

$$\mathsf{add}^{(3)} = \mathsf{add}^{(2)} \circ \left(\mathsf{P}^{(3)}_1, \mathsf{add}^{(2)} \circ (\mathsf{P}^{(3)}_2, \mathsf{P}^{(3)}_3)\right)$$

works as intended.

Now define the collection of all functions on A by

$$\mathsf{Op}_A = \bigcup_{n \geq 0} (A^n \to A)$$

We write $\mathsf{Op}^{(+)}_A = \bigcup_{n>0}(A^n \to A)$ for the collection of all functions on A excluding nullary ones.

Definition 8.1.1 (Function Clones) *A family of functions $\mathcal{C} \subseteq \mathsf{Op}_a$ is a clone (over A) or a function algebra (over A) if it is closed under composition and contains all projections.*

Informally, a clone is simply the functions that can be described by writing down terms in a suitable language. Different signatures can give rise to the same clone, so we are abstracting away certain syntactical details.

Note that the collection of all projections forms a clone, the trivial clone; Op_A is the largest clone, the full clone. Define a function $f \in \mathsf{Op}^{(n)}$ to be idempotent if $f(x, x, \ldots, x) = x$ for all x; the idempotent functions together with projections form another clone. Another standard example is the clone of all continuous functions on some topological space. For us, the most important example is arguably the clone of (total) computable functions. We will discuss several more interesting interesting clones of computable functions in the next section.

In order to construct interesting clones $\mathcal{C}$, we usually insist that $\mathcal{C}$

- contains certain basic functions $\mathcal{F}$ (beyond projections), and/or
- is closed under certain operations O (beyond composition).

We write $\mathsf{clone}(\mathcal{F}; O)$ for the least clone containing $\mathcal{F}$ and closed under O. Thus, we do not have to explicitly mention projections and composition here, they are assumed by definition. For example, $\mathsf{clone}(;)$ consists exactly of all projections.

One last comment before we look at some examples. We will write $c^{(n)}_a$ for the n-ary constant map $\boldsymbol{x} \mapsto a$. Thus $c^{(0)}_a$ is a hard constant, we refer to the others as soft constants. Since we allow nullary functions, it is worthwhile to consider the effect of such functions in conjunction with composition. We have $f^{(n)}$, $g^{(m)}_i$, $i \in [n]$, produces $f \circ (g_1, \ldots, g_n) \in \mathsf{Op}^{(m)}_A$ where $n, m \geq 0$. If f is nullary, then for $m \geq 1$ we have $c^{(m)}_{f(*)} \in \mathcal{C}$. If the g_i are nullary, then for $n \geq 1$ we have $c^{(0)}_a \in \mathcal{C}$ where $a = f(g_1(*), \ldots, g_n(*))$.

The level of detail that we just outlined goes a bit beyond what one would find in the literature. For example, ordinarily there is no need for a distinction between $\mathsf{C}^{(0)}_a$ and $\mathsf{C}^{(1)}_a$, they are both just "the constant 0." In fact, most humans would find quite so much detail annoying and distracting, and rightfully so. However, taking inspiration from Thurston's and Knuth's comments in the introduction, we are here making an attempt to lay out the specifics needed to build an interpreter for the computable clones in the next section. We need a parser that can analyze the terms used to describe functions in the clone, and an evaluation mechanism that, given a term and an argument vector, computes the result of evaluating the function in question on the given arguments. On that level, all these details matter greatly—a human mathematician can simply fill them in on demand.

8.1.2 Rudimentary and Elementary Functions

Rudimentary Functions

We will start with a fairly small clone of arithmetic functions that clearly falls far short of capturing the notion of computability, the so-called rudimentary functions. Think of this as an exercise in getting acquainted with our framework, without any substantial technical difficulties. The following definition may appear to be rather arbitrary, but see theorem 8.3.2 for a reasonable description of the same class of functions in terms of programs.

Definition 8.1.2 *The class of rudimentary functions is the clone generated by constants* 0, 1, *addition, predecessor, division and remainder with fixed modulus, and if-then.*

Thus, we are dealing with the clone $\mathsf{clone}(0, 1, +, \mathsf{pred}, \mathbf{.} \text{ div } m, \mathbf{.} \text{ mod } m, \mathsf{ift};)$ where 0 and 1 are hard constants, + and if-then are binary functions and the others are unary. By if-then we mean the following arithmetic function:

$$\mathsf{ift}(x, y) = \begin{cases} y & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

This is similar to the standard question-mark operator in the C programming language `x ? y : 0` where the third argument is fixed to be 0. The predecessor function is defined by $\mathsf{pred}(x) = x - 1$ for $x > 0$, and 0 otherwise. Note well that the two-variable modulus operation $x \text{ mod } y$ is not allowed, we only consider the situation where the second argument is fixed. There are no permissible operations other than composition.

What can we say about rudimentary functions? Let us show that the ternary if-then-else function $\mathsf{ifte}(x, y, z) = y$ if $x > 0$, and z otherwise, is also rudimentary:

$$\mathsf{ifte}(x, y, z) = \mathsf{ift}(x, y) + \mathsf{ift}((\mathsf{ift}(x, 1) + 1) \text{ mod } 2, z)$$

Strictly speaking, 2 here is an abbreviation for the term $1 + 1$, but we appeal to common sense to keep notation manageable. Note the use of modular arithmetic in the last definition: rudimentary functions provide very little control structure, so we need to exploit arithmetic to express a logical operation. Here is one example of a rudimentary function that arises naturally in Floyd's cycle finding algorithm. The generalized mod function is defined by

$$x \text{ mod } (t, p) = \begin{cases} x & \text{if } x < t, \\ t + (x - t) \text{ mod } p & \text{otherwise.} \end{cases}$$

where $t \geq 0$ and $p \geq 1$ are parameters. This function describes the position of a particle moving at speed 1 along an ultimately periodic, discrete orbit of transient length t and period p. Alternatively, we can think of a pointer moving down a linked list that contains a loop.

To see that $\text{mod } (t, p)$ is rudimentary, note that

$$x \text{ mod } (t, p) = \mathsf{ifte}(\mathsf{pred}^{\,t}(x), t + (\mathsf{pred}^{\,t}(x) \text{ mod } p), x)$$

The expression on the right is admissible since t and p are fixed parameters, so $\mathsf{pred}^{\,t}$ is obtained by repeated composition.

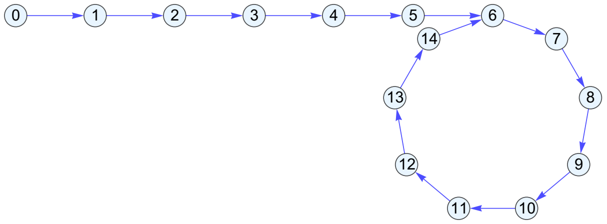


Figure 8.1: An orbit with transient 6 and period 9.

The limitations of this clone are palpable. For example, it seems that neither proper subtraction nor multiplication is rudimentary. In particular, since there is no loop construct and no recursion, there seems to be no way to repeatedly apply addition to simulate multiplication. On the other hand, note that because of the if-then-else function we can construct rudimentary functions that assume arbitrary values at finitely many given places. Thus, one has to be a bit careful when trying to pin down the limitations of rudimentary functions. A method that is often useful is to find an upper bound on the value of all functions in a clone in terms of their arguments. In this case, an affine bound works.

Lemma 8.1.1 *For every rudimentary function $f : \mathbb{N}^n \to \mathbb{N}$ there are constants $c_0, \dots, c_n$ such that*

$$f(x_1, \dots, x_n) \leq c_0 + \sum_i c_i x_i.$$

Proof. Proof is by structural induction on f. The claim is easily verified for the basic functions. If f is obtained by composition, the only interesting case is $f = g + h$. But then the affine bounds for g and h can be combined to yield an affine bound for f. □

Note that the bounding function $c_0 + \sum_i c_i x_i$ is itself rudimentary. It may, however, be much less complicated than the original function. It follows immediately that multiplication is not rudimentary. The argument for subtraction is more complicated and will be postponed to section 8.3.1. A picture of the rudimentary function $f(x, y) = (x \bmod 10) + (y \bmod 4)$ is shown in figure 8.2.

Elementary Functions

We could extend the rudimentary functions by including multiplication, exponentiation, factorials and other such functions among the basic functions. However, it is better to avoid endless proliferation of basic functions; instead we try to identify stronger function-building operations. One interesting idea is to allow summation over functions: suppose $g : \mathbb{N}^{n+1} \to \mathbb{N}$ is given. Then we can form a new function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ by

$$f(x, \boldsymbol{y}) = \sum_{z<x} g(z, \boldsymbol{y})$$

with the understanding that $f(0, \boldsymbol{y}) = 0$. This is referred to as bounded summation; bounded products are defined similarly. We will also admit one new basic function, proper subtraction: $x \mathbin{\dot{-}} y = x - y$ for $x \geq y$ and 0 otherwise. Here is a rather interesting class of functions proposed by Kalmár in 1943.

Definition 8.1.3 *The class of elementary functions is the clone containing constants 0 and 1, binary addition and proper subtraction, and closed under bounded sums and bounded products.*

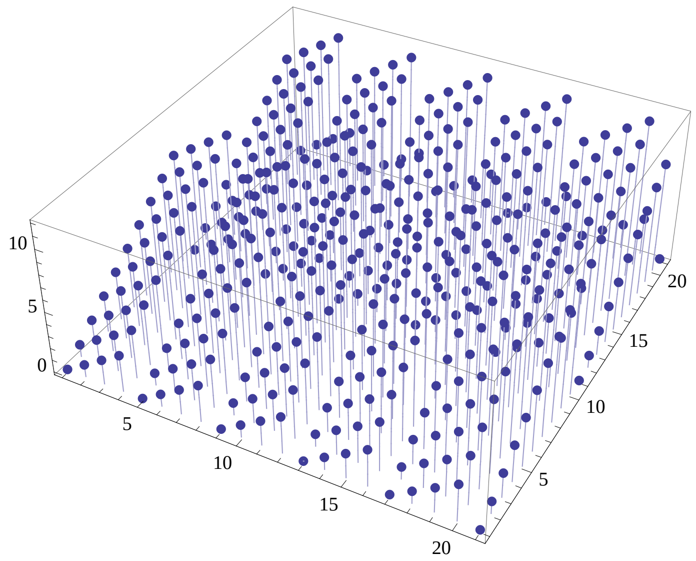


Figure 8.2: A rudimentary function of two variables.

While multiplication and exponentiation are absent from the definition, they are easily seen to be elementary:

$$x \cdot y = \sum_{z<x} y \qquad y^x = \prod_{z<x} y.$$

As a consequence, exponential polynomials are also elementary. Note, though, that it is not clear that super-exponentiation could be handled in a similar manner. For any function f define its complement function

$$\overline{f}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } f(\boldsymbol{x}) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

so that $g(\boldsymbol{x}) = 1 \dot{-} f(\boldsymbol{x})$. If f happens to be $\{0,1\}$-valued, then $g(\boldsymbol{x}) = 1 - f(\boldsymbol{x})$. As is customary in some programming languages with relaxed type structures, we can think of our constants 0 and 1 as Boolean values, false and true, respectively. We can then build support for propositional logic within the elementary functions. The sign function is defined by $\mathsf{sign}(x) = \min(1, x)$, so that $\mathsf{sign}(0) = 0$ and $\mathsf{sign}(x) = 1$ for all $x \geq 1$. Also let $\overline{\mathsf{sign}}(x) = 1 \dot{-} \mathsf{sign}(x)$. Both functions are elementary since

$$\overline{\mathsf{sign}}(x) = \prod_{z<x} 0$$

and $\overline{\overline{g}} = g$ for $\{0,1\}$-valued functions.

The interpretation of 0 and 1 as Boolean values allows us to represent relations and predicates as arithmetic functions, thus avoiding proliferation of types. We can then discuss the complexity of the relation in terms of the complexity of the corresponding function. This idea will be used over and over again.

Definition 8.1.4 *Let $R \subseteq \mathbb{N}^n$ be any relation on the natural numbers. Define its characteristic function* $\mathsf{char}_R : \mathbb{N}^n \to \mathbf{2}$ *by*

$$\mathsf{char}_R(\boldsymbol{x}) = \begin{cases} 1 & \textit{if } \boldsymbol{x} \in R, \\ 0 & \textit{otherwise.} \end{cases}$$

A relation is elementary if its characteristic function is elementary. For emphasis, we may occasionally write $\mathsf{char}(R)(\boldsymbol{x})$.

Thus, in order to express the assertion "R holds on $\boldsymbol{x}$" we can write the equation $\mathsf{char}_R(\boldsymbol{x}) = 1$. In the special case where $R \subseteq [n]$ for reasonably small n, the characteristic function of R is also known as a bitvector: the bit in position i indicates whether $i \in R$. The idea that one can think of a relation $R \subseteq \mathbb{N}$ as a function $\mathbb{N} \to \mathbf{2}$ may seem quaint and entirely obvious. However, when Cantor published his eponymous theorem in 1882, instead of saying that the powerset of a set has strictly larger cardinality than the set itself, he stated (in essence):

$$|S \to \mathbf{2}| > |S|.$$

Be warned that some older texts flip the bits and use the convention that $\mathsf{char}_R(\boldsymbol{x}) = 0$ indicates membership in R. At any rate, we can now use arithmetic to express Boolean operations on arithmetic relations.

Lemma 8.1.2 *The elementary relations are closed under intersection, union and complement.*

Proof. We express the set-theoretic operations in terms of arithmetic:

$$\begin{aligned}\mathsf{char}(R \cap S)(\boldsymbol{x}) &= \mathsf{char}(R)(\boldsymbol{x}) \cdot \mathsf{char}(S)(\boldsymbol{x})\\ \mathsf{char}(R \cup S)(\boldsymbol{x}) &= \mathsf{sign}\big(\mathsf{char}(R)(\boldsymbol{x}) + \mathsf{char}(S)(\boldsymbol{x})\big)\\ \mathsf{char}(\mathbb{N}^n - R)(\boldsymbol{x}) &= 1 \mathbin{\dot{-}} \mathsf{char}(R)(\boldsymbol{x})\end{aligned}$$

□

Note that the same argument holds for any class of functions extending the elementary ones, a fact that we will use in many places. By the lemma, the elementary relations form a Boolean algebra. Actually, slightly more is true: given an elementary function for R and S we can effectively construct an elementary function for $R \cup S$, $R \cap S$ and $\mathbb{N}^n - R$. It is not hard to see that equality and the standard order relations on the naturals are all elementary. For example, the characteristic function of equality is given by

$$E(x, y) = \overline{\mathsf{sign}}((x \mathbin{\dot{-}} y) + (y \mathbin{\dot{-}} x))$$

We can use definition-by-cases as long as the cases are given by an elementary relation.

Definition 8.1.5 *Let* $g, h : \mathbb{N}^n \to \mathbb{N}$ *and* $R \subseteq \mathbb{N}^n$. *Define* $f = \mathsf{DC}[g, h, R]$ *by*

$$f(\boldsymbol{x}) = \begin{cases} g(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in R,\\ h(\boldsymbol{x}) & \text{otherwise.}\end{cases}$$

Lemma 8.1.3 *If* g, h, R *are elementary, then* $\mathsf{DC}[g, h, R]$ *is also elementary.*

Proof. Again we express logic in terms of arithmetic. For $f = \mathsf{DC}[g, h, R]$ we have

$$f(\boldsymbol{x}) = \mathsf{char}_R(\boldsymbol{x}) \cdot g(\boldsymbol{x}) + \overline{\mathsf{char}}_R(\boldsymbol{x}) \cdot h(\boldsymbol{x})$$

Since $\mathsf{char}_R(\boldsymbol{x}) + \overline{\mathsf{char}}_R(\boldsymbol{x}) = 1$ we get the desired behavior. □

Of course, we could also have used an elementary version of the if-then-else function from the last section:

$$\mathsf{ifte}(x, y, z) = \mathsf{sign}(x) \cdot y + \overline{\mathsf{sign}}(x) \cdot z.$$

A particularly important algorithmic technique is search over some finite domain. For example, we might search for a factor of a given number. More generally, we would like to be able to find some number $z < x$ such that an elementary relation $R(z, \boldsymbol{y})$ holds. We can express search in terms of bounded existential quantification. Informally, we would like to define

$$P_\exists(x, \boldsymbol{y}) :\Leftrightarrow P(0, \boldsymbol{y}) \lor P(1, \boldsymbol{y}) \lor \ldots \lor P(x-1, \boldsymbol{y})$$

and likewise for $P_\forall$. The number of disjuncts on the right depends on x, so admissibility of this operation does not simply follow from closure under union.

Definition 8.1.6 (Bounded Quantifiers) *Define*

$$\begin{aligned} P_\forall(x, \boldsymbol{y}) &:\Leftrightarrow \forall\, z < x\, P(z, \boldsymbol{y}) \\ P_\exists(x, \boldsymbol{y}) &:\Leftrightarrow \exists\, z < x\, P(z, \boldsymbol{y}). \end{aligned}$$

We let $P_\forall(0, \boldsymbol{y}) = \mathsf{true}$ *and* $P_\exists(0, \boldsymbol{y}) = \mathsf{false}$*.*

Lemma 8.1.4 *Elementary relations are closed under bounded quantification.*

Proof.

$$\mathsf{char}(P_\forall)(x, \boldsymbol{y}) = \prod_{z<x} \mathsf{char}_P(z, \boldsymbol{y})$$

$$\mathsf{char}(P_\exists)(x, \boldsymbol{y}) = \mathsf{sign}\left(\sum_{z<x} \mathsf{char}_P(z, \boldsymbol{y})\right)$$

□

Back to search. Given a function $g : \mathbb{N}^{n+1} \to \mathbb{N}$ we call z a witness for $\boldsymbol{y} \in \mathbb{N}^n$ if $g(z, \boldsymbol{y}) = 0$. More precisely, we are looking for the least witness for each $\boldsymbol{y} \in \mathbb{N}^n$, hence this technique is also referred to as minimization. Of course, the witness may fail to exist, so we truncate our search.

Definition 8.1.7 (Bounded Search (Bounded Minimization)) *Let* $g : \mathbb{N}^{n+1} \to \mathbb{N}$*. Then* $\mathsf{BS}[g] : \mathbb{N}^{n+1} \to \mathbb{N}$ *is the function* f *defined by*

$$f(x, \boldsymbol{y}) = \begin{cases} \min\big(\, z < x \mid g(z, \boldsymbol{y}) = 0 \,\big) & \textit{if } z \textit{ exists,} \\ x & \textit{otherwise.} \end{cases}$$

Thus $f(x, \boldsymbol{u}) = x$ indicates failure of the search up to x. Since we are in a strictly arithmetical setting, we cannot return some convenient failure indicator, or throw an exception. Elementary functions are closed under bounded search as expressed in the next lemma.

Lemma 8.1.5 *If g is elementary, then so is* $\mathsf{BS}[g]$*.*

Proof. Define the elementary predicate

$$P(z, \boldsymbol{y}) :\Leftrightarrow \forall\, z' \le z\, (g(z', \boldsymbol{y}) \neq 0)$$

Thus $P(z, \boldsymbol{y})$ means that all $z' \le z$ fail to be witnesses. Note that $P(z, \boldsymbol{y})$ can hold only on an initial segment of $\mathbb{N}$. But then

$$\mathsf{BS}[g](x, \boldsymbol{y}) = \sum_{z<x} \mathsf{char}_P(z, \boldsymbol{y})$$

is elementary, as required. □

Here is a more direct, albeit slightly more cryptic way, to show that bounded search is admissible in the context of elementary functions:

$$\mathsf{BS}[g](x,\boldsymbol{y}) = \sum_{z<x}\Big(1 \dot{-} \sum_{u\leq z}(1 \dot{-} g(u,\boldsymbol{y}))\Big)$$

We can push this result a little further: the search does not have to end at x, but it can extend to the value of an elementary function h of x and $\boldsymbol{y}$. We write $\mathsf{BS}[g,h]$ for the function

$$f(x,\boldsymbol{y}) = \begin{cases}\min\big(\, z < h(x,\boldsymbol{y}) \mid g(z,\boldsymbol{y}) = 0\,\big) & \text{if } z \text{ exists,}\\ h(x,\boldsymbol{y}) & \text{otherwise.}\end{cases}$$

Note, though, that we do need to have some bound: as we will see shortly, unbounded search as in $\min\big(\, z \mid g(z,\boldsymbol{y}) = 0\,\big)$ is not an admissible operation for elementary functions; not even when the existence of a witness z for each $\boldsymbol{y}$ is guaranteed.

Minimization is useful to show that certain number theoretic functions are elementary. For example, x div $y = \min\big(\, q < x \mid x < (q+1)y\,\big)$ and $x \bmod y = x \dot{-} y \cdot (x \text{ div } y)$. As a consequence, the function $\mathsf{bit}(a,i)$ that returns the ith bit in the binary expansion of a is also elementary: $\mathsf{bit}(a,i) = (a \text{ div } 2^i) \bmod 2$.

Example 8.1.1 Here is a more interesting application of bounded search: we will show that the function that enumerates primes is elementary. First, bounded existential quantification shows that divisibility is elementary

$$x \mid y :\Leftrightarrow \exists\, z < y+1\,(x \cdot z = y)$$

Bounded universal quantification shows that primality is elementary

$$\mathsf{prime}(x) :\Leftrightarrow x \geq 2 \wedge \forall\, z < x\,(z \mid x \Rightarrow z = 1)$$

The relation "x is the nth prime" is elementary (with 2 being the 0th prime):

$$P(n,x) :\Leftrightarrow \mathsf{prime}(x) \wedge n = \sum_{z<x} \mathsf{char}_{\mathsf{prime}}(z)$$

In order to turn this relation into a function $n \mapsto p_n$ such that $P(n,p_n)$ we can use closure under bounded search. To this end we need an elementary bound on the size of the nth prime. Since we are not concerned about a tight bound 2^{2^n} will do:

$$p_n = \min\big(\, z < 2^{2^n} \mid P(n,z)\,\big)$$

Number theory provides much better bounds, but all that matters for our purposes is that the bounding function is elementary.

Note that iterated exponential functions are elementary, so an elementary function can grow very rapidly. As we will see in chapter **??**, every function that can be computed in elementary time (or, for that matter, elementary space) is already elementary. Thus, elementary functions turn out to be quite powerful enough to capture any kind of "practical" problem.

8.1.3 Elementary Coding

The only data type available so far for our computable functions is natural number. We do not have a direct way of computing with other discrete structures such as words, lists, trees, graphs and so on. One way to deal with this constraint is to code all these objects as numbers, so we can then apply, say, elementary functions, to perform operations on these objects. This is a proof-of-concept type argument, in practical computation we need to avoid all unnecessary coding for efficiency reasons. Also note that we would like to keep the coding operations as simple as possible, otherwise we can hide complexity in a way that is clearly bogus. For example, suppose we were to code natural numbers as follows: if n is composite, we code it as $2n$, written in binary; and as $2n+1$ otherwise. In this framework primality testing is trivial, clearly not a desirable situation when one is interested in computation and complexity. And, we could push this further: using the same trick we can code numbers so that membership testing in any set $A \subseteq \mathbb{N}$ becomes trivial. To avoid such pathologies we will use very basic coding functions.

Pairing and Coding Functions

More precisely, we would like to give a simple definition of an arithmetic coding function, a multiadic map

$$\langle . \rangle : \mathbb{N}^\star \to \mathbb{N}$$

that associates sequences of natural numbers with a single number, in a reversible fashion. Such functions exist for purely set-theoretic reasons: the two sets have the same cardinality $\aleph_0$. However, we want the map and its associated decoding functions to be easy to compute. By this we mean that there are "inverse" functions $\mathsf{len} : \mathbb{N} \to \mathbb{N}$ and $\mathsf{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that

$$\begin{aligned} \mathsf{len}(\langle a_0, a_1, \ldots, a_{n-1} \rangle) &= n, \\ \mathsf{dec}(\langle a_0, a_1, \ldots, a_{n-1} \rangle, i\,) &= a_i, \quad i < n \end{aligned}$$

for all sequences $a_0, a_1, \ldots, a_{n-1}$. Thus len determines the length of the sequence and dec decodes it back into its elements.

The first step in finding a coding function is to solve the easier problem of selecting a pairing function, a map $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ with unpairing functions π_1 and π_2 such that $\pi_i(\pi(x_1, x_2)) = x_i$. The existence of unpairing functions implies that π must be injective. For technical reasons, one often insists that $\pi(x, y) \geq x, y$, so that the code of a large number cannot be small. There are many choices for a pairing function, the classical one due to Cantor is the second-degree polynomial:

$$\pi(x, y) = \frac{(x+y+1)(x+y)}{2} + x = \binom{x+y+1}{2} + x$$

It is not hard to see that this particular pairing function is actually a bijection, see the exercises. A surprising theorem by Fueter and Pólya from 1923 states that, up to a swap of variables, this is the only quadratic polynomial that defines a bijection $\mathbb{N}^2 \leftrightarrow \mathbb{N}$. The proof is rather difficult and uses the fact that e^a is transcendental for algebraic $a \neq 0$. Even more surprisingly, it is an open problem whether there are other bijections for higher degree polynomials.

At any rate, for our purposes, the following exponential function turns out to be easier to deal with in the end.

$$\pi(x, y) = 2^x(2y+1)$$

Note that the binary expansion of $\pi(x,y)$, MSD first, looks like so:

$$y_k y_{k-1} \dots y_0 1 \underbrace{00\dots0}_{x}$$

where $y_k y_{k-1} \dots y_0$ is the binary expansion of y, MSD first. This makes it easy to find the corresponding unpairing functions. Figure 8.3 shows a visualization of the Cantor versus the binary pairing function.

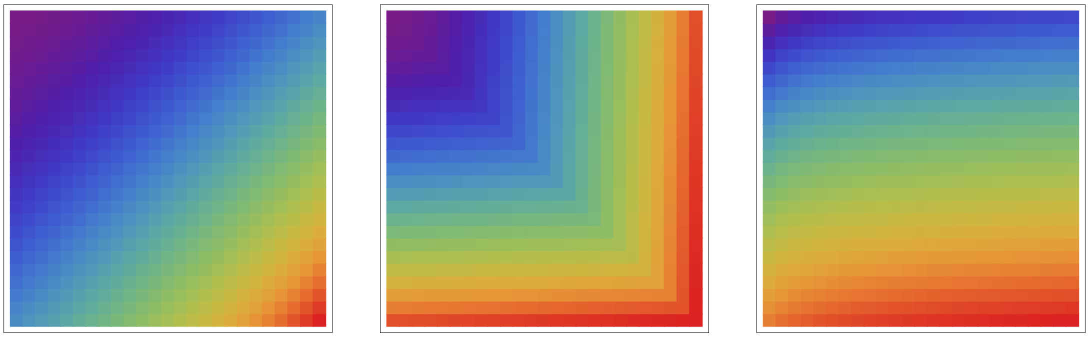


Figure 8.3: Images of three natural pairing functions on $\mathbb{N}$.

We can now iterate the pairing function to code sequences of natural numbers as follows.

$$\begin{aligned} \langle \mathsf{nil} \rangle &= 0 \\ \langle a_0, \dots, a_{n-1} \rangle &= \pi(a_0, \langle a_1, \dots, a_{n-1} \rangle) \end{aligned}$$

The numbers that appear as codes of sequences are called sequence numbers and we write

$$\mathsf{Seq} = \{ \langle \boldsymbol{a} \rangle \mid \boldsymbol{a} \in \mathbb{N}^\star \}.$$

In general, it is not immediately clear what Seq is, but due to our choice of pairing function, the binary expansion of a sequence number $\langle a_0, \dots, a_{n-1} \rangle$ has the form

$$1 \underbrace{00\dots0}_{a_{n-1}} 1 \underbrace{00\dots0}_{a_{n-2}} \dots 1 \underbrace{00\dots0}_{a_1} 1 \underbrace{00\dots0}_{a_0}$$

For example, $\langle 1,2,3 \rangle = 274 = 100010010_2$ and $\langle 0,0,0,0 \rangle = 15 = 1111_2$. It follows that $\mathsf{Seq} = \mathbb{N}$, all natural numbers appear as sequence numbers and we have actually constructed a bijection $\mathbb{N}^\star \leftrightarrow \mathbb{N}$. Note, though, that other definitions of a coding function may well produce a smaller set of sequence numbers, see below for Gödel's β function and the exercises. We will continue to write, say, $x \in \mathsf{Seq}$, whenever it is important that x is a sequence number.

It is clear that our pairing function $\pi(x,y) = 2^x(2y+1)$ and the associated unpairing functions π_1 and π_2 are all elementary. The number of elements in a list coded by sequence number a is none other than the binary digit sum of a, the number of 1's in the binary expansion of a. Since the binary digit function bit is elementary, so is $\mathsf{len}(a)$. The behavior of the length function is somewhat erratic as seen in figure 8.4. The decoding function has the form

$$\mathsf{dec}(a,i) = \pi_1(\pi_2^i(a)) \qquad \text{for } i < \mathsf{len}(a)$$

with the understanding that $\mathsf{dec}(a,i) = 0$ for $i \geq \mathsf{len}(a)$. In the older literature, it is customary to write $(a)_i$ for $\mathsf{dec}(a,i)$.

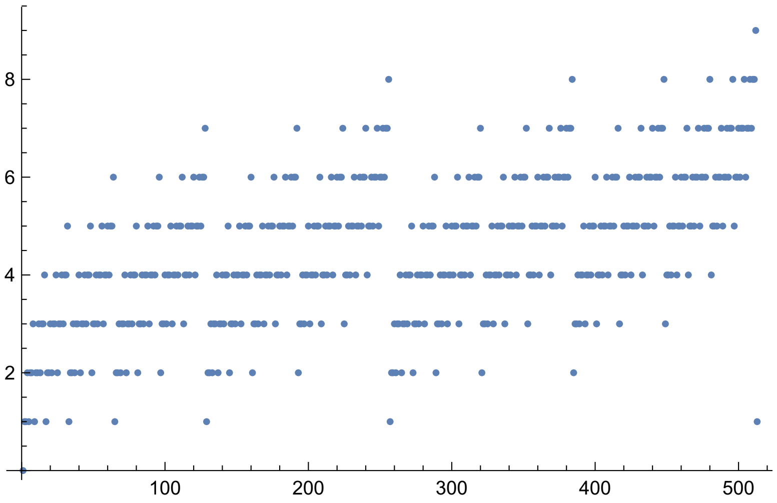


Figure 8.4: The number of 1's in the binary expansion of the numbers up to 512.

Lemma 8.1.6 *The decoding function* dec *is elementary.*

Proof. First define a function pos that determines the position of the ith 1-bit in the binary expansion of a, where $0 \leq i < \mathsf{len}(a)$.

$$\begin{aligned}
\mathsf{pos}(a,0) &= \min\big(\, k < a \mid a \bmod 2^k = 0 \wedge a \bmod 2^{k+1} \neq 0 \,\big) \\
\mathsf{pos}(a,k) &= \min\big(\, k < a \mid \sum_{i \leq k} \mathsf{bit}(a,i) = k+1 \,\big)
\end{aligned}$$

Then

$$\begin{aligned}
\mathsf{dec}(a,0) &= \mathsf{pos}(a,0) \\
\mathsf{pos}(a,k) &= \mathsf{pos}(a,k) \mathbin{\dot{-}} \mathsf{pos}(a,k \mathbin{\dot{-}} 1) \mathbin{\dot{-}} 1
\end{aligned}$$

This shows that dec is elementary. □

Note that, technically, the function $\langle . \rangle$ cannot be elementary since it has domain $\mathbb{N}^\star$. However, for every fixed n, the n-ary restriction $\langle x_1, \ldots, x_n \rangle$ is clearly elementary. Moreover, we can bound the value of our coding function in an elementary way. For $\boldsymbol{x} = x_1, \ldots, x_n$ in $\mathbb{N}^n$ let $\widehat{\boldsymbol{x}} = \max(x_1, \ldots, x_n)$.

Lemma 8.1.7 *There is an elementary function* $B(x,y)$ *such that for all* n *and* $\boldsymbol{a} = a_1, \ldots, a_n$ *we have* $\langle \boldsymbol{a} \rangle \leq B(n, \widehat{\boldsymbol{a}})$.

Proof. It is easy to see that

$$\langle a_1, a_2, \ldots, a_n \rangle < 2^{n + \sum a_i}$$

The function on the right is clearly elementary. □

∗ Gödel's β-Function

The coding function based on iterating a pairing operation from the last section is by no means the only choice. For example, we can exploit unique decomposition into prime factors to concoct a (piecewise) elementary coding function by

$$\langle a_1, \ldots, a_n \rangle = p_1^{a_1+1} p_2^{a_2+1} \ldots p_n^{a_n+1}$$

Here p_n stands for the nth prime, so that the function $n \mapsto p_n$ is elementary. Decoding here comes down to computing prime factorizations. Gödel, in the proof of his famous incompleteness theorem, used yet another divisibility-based construction that is somewhat more basic and relies on the Chinese Remainder Theorem: a single number $s < \prod m_i$ can code a whole sequence of numbers $a_i < m_i$, provided the m_i are pairwise coprime. Technically, we start with the decoding function and use bounded search to construct the corresponding coding function.

Lemma 8.1.8 *There exists an elementary function $\beta : \mathbb{N}^2 \to \mathbb{N}$ such that*

$$\forall\, a_0, \ldots, a_{n-1}\, \exists\, a\, \forall\, i < n\, (a_i = \beta(a, i)).$$

Proof. Let π be our standard pairing function with unpairing functions π_1 and π_2, all elementary. Set

$$\beta(a, i) = \pi_1(a) \bmod (i + 1)\pi_2(a) + 1$$

where $x \bmod y$ denotes the binary remainder function. The argument a is intended to be of the form $a = \pi(s, m)$. Let us write $m_i = (i + 1)m + 1$, so that $\beta(a, i) = s \bmod m_i$. We need to establish the existence of such a witness a. Set $m = \max(a_0, \ldots, a_{n-1}, n)!$ and note that for all $0 \leq i < j < n$, the numbers m_i and m_j are coprime. But then the simultaneous congruences $x = a_i \pmod{m_i}$ have a solution s which can be chosen to be less than $\prod m_i$. □

We can now define a coding function $\langle . \rangle$ in a somewhat indirect fashion by setting

$$\langle a_1, \ldots, a_n \rangle = \min\big(\, a \mid \beta(a, 0) = n \wedge \forall\, i \in [n]\, (\beta(a, i) = a_i)\,\big)$$

As before, the restriction of $\langle . \rangle$ to $\mathbb{N}^n$ is elementary for each fixed n. Likewise, the set of sequence numbers is elementary. As the proof shows, β requires only addition, multiplication and remainder.

Data Structures

Using our coding function, we can now interpret finitary data structures as natural numbers, and their associated operations as arithmetic functions. For example, suppose we wish to use lists of natural numbers, objects in $\mathbb{N}^\star$. Write nil for the empty list, and $\mathbb{N}^+$ for the collection of all non-empty list. We need the following three fundamental operations, a constructor and two destructors:

$\mathsf{cons} : A \times A^\star \to A^+$	a bijection
$\mathsf{head} : A^+ \to A$	$\mathsf{head}(\mathsf{cons}(a, x)) = a$
$\mathsf{tail} : A^+ \to A^\star$	$\mathsf{tail}(\mathsf{cons}(a, x)) = x$

Using the coding function from section 8.1.3, these are straightforward to implement: we use sequence numbers for the lists. Our coding function has the advantage that it conforms to the standard construction of lists in languages such as Lisp:

$$\mathsf{cons}(x_1, \mathsf{cons}(x_2, \ldots, \mathsf{cons}(x_n, \mathsf{nil}) \ldots)$$

As a consequence, cons is but an application of π, head is π_1 and tail is π_2. Other natural list operations are also easily seen to be elementary by applying the bounded search lemma. Consider, for example, the append operation, given by

$$(\,\mathsf{nil}, a) = \mathsf{cons}(a, \mathsf{nil}) \qquad (\,\mathsf{cons}(b, x), a) = \mathsf{cons}(b, (\,x, a))$$

Note that for any sequence number s

$$(\,a, s) = \min\bigl(\, t \in \mathsf{Seq} \mid \mathsf{len}(t) = \mathsf{len}(s) + 1 \land (t)_{\mathsf{len}(s)} = a \land \forall\, i < \mathsf{len}(s)(t)_i = (s)_i \,\bigr)$$

We can bound the search by $s \cdot 2^{a+1}$, so the append operation is also elementary by the bounded search lemma. The same holds true for the operations of reversal, join, sorting and so on.

In a similar manner, we can reconstruct the basic concepts of set theory as long as we confine ourselves to hereditarily finite sets of natural numbers. For example, a set $\{a_1, \dots, a_n\} \subseteq \mathbb{N}$ can be expressed as the sequence number $\langle a_1, \dots, a_n \rangle$. The collection of code numbers of sets is elementary, as is the membership relation:

$$\begin{aligned} \mathsf{Set}(X) &:\Leftrightarrow \mathsf{Seq}(X) \land \forall\, i < j < \mathsf{len}(X)\, ((X)_i \neq (X)_j) \\ x \in X &:\Leftrightarrow \mathsf{Set}(X) \land \exists\, i < \mathsf{len}(X)\, (x = (X)_i). \end{aligned}$$

Since we do not impose any order requirements, equality of sets differs from numerical equality.

$$\begin{aligned} X \subseteq Y &:\Leftrightarrow \mathsf{Set}(X) \land \mathsf{Set}(Y) \land \forall\, i < \mathsf{len}(X)\, ((X)_i \in Y) \\ X =_s Y &:\Leftrightarrow X \subseteq Y \land Y \subseteq X \end{aligned}$$

The power set operation is elementary:

$$\mathfrak{P}(X) = \begin{cases} \min\bigl(\, Z \mid \mathsf{Set}(Z) \land \mathsf{len}(Z) = 2^{\mathsf{len}(X)} \land \forall\, z \subseteq X\, (z \in Z)\,\bigr) & \text{if } \mathsf{Set}(X), \\ 0 & \text{otherwise.} \end{cases}$$

Again, by lemma 8.1.7, we can bound the min operator. To represent finite relations and functions we construct sets of tuples.

$$\begin{aligned} \mathsf{Rel}_n(X) &:\Leftrightarrow \mathsf{Set}(X) \land \forall\, x \in X\, (\mathsf{Seq}(x) \land \mathsf{len}(x) = n) \\ \mathsf{Func}_n(X) &:\Leftrightarrow \mathsf{Rel}_n(X) \land \forall\, x, y \in X\, (\mathsf{Init}(x, n) = \mathsf{Init}(y, n) \Rightarrow (x)_2 = (y)_2) \end{aligned}$$

Here $\mathsf{Init}(a, i) = \langle (a)_0, \dots, (a)_{i-1} \rangle$ for all $i \leq \mathsf{len}(a)$, and 0 otherwise, returns the code of an initial segment of a sequence. As a consequence, we can manipulate all finite Bourbaki-style structures in the setting of elementary computation. Natural operations on these structures, such as testing for isomorphism, are then elementary. Of course, this is no more that a proof-of-concept, any realistic implementation would have to rely on much more efficient data structures.

8.1.4 Bounded Recursion

Our elementary coding machinery shows that elementary functions are already quite powerful and cover, modulo coding, a great many operations in discrete mathematics. Some of the arguments are complicated by the fact that some functions are most easily defined by recursion, but our elementary functions lack a recursion mechanism. Of course, bounded sum and product are really defined by recursion:

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

Many familiar functions from arithmetic follow the same pattern. For example, as Dedekind recognized in the 19th century, addition can be defined in terms of the operation "add 1" by

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

To make clear that we are dealing with the successor function rather than general addition, we will often write $S(x)$ or x^+ in this context. Thus the equations for addition have the form

$$\begin{aligned} \mathsf{add}(0,y) &= y \\ \mathsf{add}(x^+,y) &= \mathsf{add}(x,y)^+ \end{aligned}$$

It thus seems reasonable to explore recursion as a function-building operation in general. We will start with a rather restricted form of recursion, bounded primitive recursion, and consider more general methods later. Suppose we have a recursion function $h : \mathbb{N}^{n+2} \to \mathbb{N}$, a default function $g : \mathbb{N}^n \to \mathbb{N}$ and a bounding function $B : \mathbb{N}^{n+1} \to \mathbb{N}$. We can use these to define a new function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ by

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

subject to the condition

$$f(x,\boldsymbol{y}) \leq B(x,\boldsymbol{y})$$

The first two equations should be familiar to anyone who has ever encountered a modern programming language. The bounding condition may seem somewhat out of place, but it will become clear from the following proof that it is important to limit the complexity of f. The key idea is that sequence numbers can express finite lists of integers and can thus be used to keep track of previous values of a function and, in a sense, can implement recursion.

Lemma 8.1.9 *The elementary functions are closed under bounded recursion: whenever h, g and B are elementary, then so is f.*

Proof. The two defining equations can be expressed in terms of sequence numbers like so:

$$\begin{aligned} f(x,\boldsymbol{y}) = z \iff \exists\, s \in \mathsf{Seq}\,\big(&\mathsf{len}(s) = x^+ \wedge (s)_0 = g(\boldsymbol{y}) \wedge \\ &\forall\, 1 < i < x\, ((s)_{i^+} = h(i,(s)_i,\boldsymbol{y})) \wedge (s)_x = z\big) \end{aligned}$$

Thus the sequence number s records all previous values of f. Since $f(x,\boldsymbol{y})$ is required to be bounded from above by $B(x,\boldsymbol{y})$, it follows that f is elementary. □

It may not be entirely clear that such a function f actually exists. But note that we can compute $f(0,\boldsymbol{y})$, $f(1,\boldsymbol{y})$, ..., $f(x-1,\boldsymbol{y})$, $f(x,\boldsymbol{y})$ in order, a standard bottom-up, dynamic programming style approach. Because of the bound B, this whole process is still elementary; without it, we still have an existence argument, but f might be much more complicated.

Given all these closure properties, how would we go about showing that some computable function fails to be elementary? Following the bounding method that worked for rudimentary functions, it is tempting to find a bound for the growth of elementary functions. We can simplify n-dimensional input a bit by considering only the largest component: $\max \boldsymbol{x} = \max\big(x_i \mid i = 1, \ldots, n \,\big)$. Define an iterated exponential function by $2{\uparrow}(0,z) = z$ and $2{\uparrow}(k+1,z) = 2^{2{\uparrow}(k,z)}$. Thus, for every fixed k the function $x \mapsto 2{\uparrow}(k,x)$, is elementary.

Lemma 8.1.10 *For every elementary function f, there is some k such that*

$$f(\boldsymbol{x}) \leq 2{\uparrow}(k, \max \boldsymbol{x}).$$

Proof. By induction on the buildup of f. □

Corollary 8.1.1 *The diagonal function $2{\uparrow}k = 2{\uparrow}(k,k)$ is not elementary.*

Note that the function $2{\uparrow}k$ is clearly computable in the intuitive sense, though it is indeed growing at an enormous rate. In a while, we will see other computable functions that dwarf $2{\uparrow}k$.

8.1.5 Primitive Recursion

One can check that a great many number theoretic functions are already elementary. Yet, the powers-of-2 function from above shows that there are fairly natural and clearly computable functions that fail to be elementary, so we still need to enlarge the clone. This raises the question as to what kind of operations should be considered admissible if we were to try to extend the elementary functions to the class of all computable functions. Our first step will be to remove the bound from the recursions in the last section.

Unbounded Recursion

This time we will have a rather anemic set of basic functions, a hard constant zero and the successor function:

Constant zero	$0 : \mathbb{N}$	
Successor function	$S : \mathbb{N} \to \mathbb{N}$	$S(x) = x + 1$

However, we require stronger closure conditions of our new clone, and in conjunction with these closure conditions our basic functions fully suffice. Our additional closure operation is motivated by the last section; we drop the boundedness condition used there and obtain full primitive recursion. Given $h : \mathbb{N}^{n+2} \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$ one can define a new function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ by

$$\begin{aligned} f(0, \boldsymbol{y}) &= g(\boldsymbol{y}) \\ f(x^+, \boldsymbol{y}) &= h(x, f(x, \boldsymbol{y}), \boldsymbol{y}) \end{aligned}$$

We will write $\mathsf{Prec}[h,g]$ for the function defined from h and g by primitive recursion. These functions have to have suitable arity, a point that will mostly ignore in the future; for us, everything type-checks.

In some modern programming languages a definition of f will look essentially just like our equations, though some minor syntactic details may differ. We could compute $f(x, \boldsymbol{y})$ in the bottom-up fashion described in the context of bounded recursion, but a compiler or interpreter is more likely to use a top-down, recursive approach. The details of how this computation is organized are a bit complicated and involve the use a of a stack of pending function calls, a technique first pioneered in the programming language ALGOL.

Definition 8.1.8 *The class of primitive recursive (p.r.) function is the clone generated from constant 0 and successor, and closed under primitive recursion.*

First note that all constant functions $\mathsf{C}^{(k)}_n : \mathbb{N}^n \to \mathbb{N}$, $n, k \geq 0$, are primitive recursive. We obtain the hard constants $\mathsf{C}^{(k)}_0$ from 0, S and composition. For arity 1, we can use primitive recursion:

$$\begin{aligned} \mathsf{C}^{(k)}_1(0) &= S^k(0) \\ \mathsf{C}^{(k)}_1(x^+) &= \mathsf{C}^{(k)}_1(x) \end{aligned}$$

More precisely, $\mathsf{C}^{(k)}_1 = \mathsf{Prec}[\mathsf{P}^{(2)}_2, S \circ S \circ \ldots \circ S(0)]$. Similarly we can define constant functions for all higher arities. Of course, we will mostly use the standard notion for constants 1, 2, 3 and so on.

The next step is to show that the clone of primitive recursive functions includes the clone of elementary ones. To begin with, the standard functions of arithmetic are all primitive recursive. For example, addition looks like

$$\begin{aligned} \mathsf{add}(0, y) &= y \\ \mathsf{add}(x^+, y) &= \mathsf{add}(x, y)^+ \end{aligned}$$

So addition can be obtained by one application of primitive recursion from projections and the successor function. Thus $\mathsf{add} = \mathsf{Prec}[S \circ \mathsf{P}^{(3)}_2, \mathsf{P}^{(1)}_1]$, corresponding to $g(y) = y$ and $h(x, z, y) = S(z)$ in the defining equations above. It is clear that the formal terms are somewhat difficult to read for humans, though it is not too hard to implement a little interpreter that evaluates these terms, given appropriate numerical arguments. Here is an example of a detailed computation for $\mathsf{add}(2, 3)$.

$$\begin{aligned} \mathsf{add}(2, 3) &= \mathsf{Prec}[S \circ \mathsf{P}^{(3)}_2, \mathsf{P}^{(1)}_1](2, 3) \\ &= S \circ \mathsf{P}^{(3)}_2(1, \mathsf{Prec}[S \circ \mathsf{P}^{(3)}_2, \mathsf{P}^{(1)}_1](1, 3), 3) \\ &= S \circ \mathsf{P}^{(3)}_2(1, S \circ \mathsf{P}^{(3)}_2(1, \mathsf{Prec}[S \circ \mathsf{P}^{(3)}_2, \mathsf{P}^{(1)}_1](0, 3), 3), 3) \\ &= S \circ \mathsf{P}^{(3)}_2(1, S \circ \mathsf{P}^{(3)}_2(1, \mathsf{P}^{(1)}_1(3), 3), 3) \\ &= S \circ \mathsf{P}^{(3)}_2(1, 4, 3) \\ &= 5 \end{aligned}$$

So we start with the term $\mathsf{add}(2, 3)$, unfold it according to the definition, and then perform step-wise reductions until the final, numerical result appears. Multiplication can be obtained by another recursion, using addition as a building block.

$$\begin{aligned} \mathsf{mult}(0, y) &= 0 \\ \mathsf{mult}(x^+, y) &= \mathsf{add}(\mathsf{mult}(x, y), y) \end{aligned}$$

which translates into $\mathsf{mult} = \mathsf{Prec}[\mathsf{add} \circ (\mathsf{P}^{(3)}_3, \mathsf{P}^{(3)}_2), \mathsf{C}^{(0)}_1]$, or, fully expanded, $\mathsf{mult} = \mathsf{Prec}[\mathsf{Prec}[S \circ \mathsf{P}^{(3)}_2, \mathsf{P}^{(1)}_1] \circ (\mathsf{P}^{(3)}_3, \mathsf{P}^{(3)}_2), \mathsf{C}^{(0)}_1]$. Similarly exponentiation can be defined via multiplication:

$$\begin{aligned} \exp(x, 0) &= 1 \\ \exp(x, y^+) &= \mathsf{mult}(\exp(x, y), x) \end{aligned}$$

A more surprising use of recursion is the following definition of the predecessor function:

$$\begin{aligned} \mathsf{pred}(0) &= 0 \\ \mathsf{pred}(x^+) &= x \end{aligned}$$

As a last example, consider the factorial function. We have

$$f(0) = 1$$
$$f(x^+) = \mathsf{mult}(x^+, f(x))$$

Or, spelled out as a formal expression, we get

$$\mathsf{Prec}[\mathsf{Prec}[\mathsf{Prec}[\mathsf{S} \circ \mathsf{P}^{(3)}_2, \mathsf{P}^{(1)}_1] \circ (\mathsf{P}^{(3)}_2, \mathsf{P}^{(3)}_3), \mathsf{C}^{(0)}_1] \circ (\mathsf{S} \circ \mathsf{P}^{(2)}_1, \mathsf{P}^{(2)}_2), \mathsf{C}^{(1)}_0]$$

It is in fact quite difficult to come up with an arithmetic function that fails to be primitive recursive, but is still computable in the intuitive sense.

Closure Properties

At this point it seems plausible that all elementary functions are primitive recursive. To prove that this is indeed the case, we need to show that bounded sums and products are admissible in the sense that they produce p.r. functions when applied to p.r. functions.

Lemma 8.1.11 *Let $g : \mathbb{N}^{n+1} \to \mathbb{N}$ be primitive recursive, and define $f(x, \boldsymbol{y}) = \Sigma_{z<x} g(z, \boldsymbol{y})$. Then $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is again primitive recursive. The same holds for products.*

Proof. Summation is naturally defined in a recursive manner, here is the corresponding formal term:

$$\mathsf{Prec}[\mathsf{add} \circ (g \circ (\mathsf{P}^{(\mathsf{n+3})}_2, \ldots, \mathsf{P}^{(\mathsf{n+2})}_{\mathsf{n+2}}), \mathsf{P}^{(\mathsf{n+2})}_2), \mathsf{C}^{(0)}_n]$$

In less cryptic notation, write $F(x, \boldsymbol{y}) = \sum_{z<x} g(x, \boldsymbol{y})$. Then

$$F(0, \boldsymbol{y}) = 0$$
$$F(x^+, \boldsymbol{y}) = F(x, \boldsymbol{y}) + g(x, \boldsymbol{y})$$

and bounded summation can be seen as a particular application of primitive recursion. □

Corollary 8.1.2 *Elementary functions are primitive recursive.*

As a consequence, we can now lift many of the results from elementary functions to primitive recursive ones. The good news is that this requires little effort, the proofs carry over verbatim.

Lemma 8.1.12 *Primitive recursive relations are closed under union, intersection and complement as well as bounded quantification.*

Proposition 8.1.1 *Primitive recursive functions are closed under definition-by-cases and bounded search.*

These results generalize to the case where the quantification is over an interval described by a p.r. function, and to search bounded by a p.r. function. Sequence numbers can also be used to enhance the technique of recursion itself: we can have the next value depend on all previous values, rather then just the last. This is usually referred to as course-of-value recursion. Define

$$\widetilde{f}(x, \boldsymbol{y}) = \langle f(0, \boldsymbol{y}), f(1, \boldsymbol{y}), \ldots, f(x, \boldsymbol{y}) \rangle$$

It is easy to see that f is elementary if, and only if, $\widetilde{f}$ is elementary. Thus, it is natural to generalize our recursion scheme slightly by defining functions so that the next value depends directly on all the previous values.

$$\begin{aligned} f(0, \boldsymbol{y}) &= g(\boldsymbol{y}) \\ f(x^+, \boldsymbol{y}) &= H(x, \widetilde{f}(x, \boldsymbol{y}), \boldsymbol{y}) \end{aligned}$$

Lemma 8.1.13 (Course-of-Value Recursion) *If g and H are primitive recursive, then f is also primitive recursive.*

Proof. Define a function F by primitive recursion as follows.

$$\begin{aligned} F(0, \boldsymbol{y}) &= \langle g(\boldsymbol{y}) \rangle \\ F(x^+, \boldsymbol{y}) &= \mathsf{app}(F(x, \boldsymbol{y}), \langle H(x, F(x, \boldsymbol{y}), \boldsymbol{y}) \rangle) \end{aligned}$$

Here app is the append operation for sequences as in section 8.1.3. Then $F(x, \boldsymbol{y}) = \widetilde{f}(x, \boldsymbol{y})$ and f is duly p.r. □

Example 8.1.2 Here is an application. We have already seen that finite sets of numbers can be expressed via coding, so consider the function

$$\begin{aligned} &H(0) = \{0\} \qquad H(1) = \{0, 1\} \\ &H(x) = H(\lfloor x/2 \rfloor) \cup H(\lceil x/2 \rceil) \cup \{x\} \end{aligned}$$

For example, $H(51) = \{0, 1, 2, 3, 4, 6, 7, 12, 13, 25, 26, 51\}$. H appears naturally in the analysis of certain recursive algorithms, for example, in the fast computation of Fibonacci numbers. By the lemma, H is primitive recursive. It is a healthy exercise to determine the cardinality of $H(n)$ as a function of n.

Evaluation and Partial Functions

By definition, primitive recursive functions can be expressed by terms in a simple programming language. These terms are just strings, syntactic objects that can easily be coded as numbers. For example, we could adopt the following convention: the first component of the sequence number $\widehat{f}$ for f determines the type of function, the second, its arity, and the remaing components, if any, provide information about how to compute f.

0	$\langle 0, 0 \rangle$	zero
S	$\langle 1, 1 \rangle$	successor
$\mathsf{P}^{(\mathsf{n})}_{\mathsf{i}}$	$\langle 2, n, i \rangle$	projections
$\mathsf{Prec}[h, g]$	$\langle 3, n, \widehat{h}, \widehat{g} \rangle$	primitive recursion
$\mathsf{Comp}[h, g_1, \ldots, g_n]$	$\langle 4, m, \widehat{h}, \widehat{g_1}, \ldots, \widehat{g_n} \rangle$	composition

In the last case, m is supposed to be the second component of $\widehat{g}$. This translation from primitive recursive functions to natural numbers suggests the following definition.

Definition 8.1.9 *Every primitive recursive function f is effectively associated with a natural number $\widehat{f}$, an index for f.*

We obtain an enumeration $(\tau_e)_{e \geq 0}$ of all unary primitive recursive functions. More precisely, if e is an index of a unary function we interpret τ_e in the obvious way; if not, we think of τ_e

as the unary constant zero function. This enumeration is the arithmetical version of listing all primitive recursive programs, say, in length-lex order. The purpose of programs is to be evaluated on certain inputs, and it is intuitively clear that the process of evaluation is itself a computable operation. In the world of digital computers evaluation is handled by compilers and runtimes, where the program and the arguments are given as strings. In our environment we can model this situation by an operation

$$\begin{aligned} \mathsf{eval} : \mathbb{N} \times \mathbb{N} &\longrightarrow \mathbb{N} \\ \mathsf{eval}(e, x) &= \tau_e(x) \end{aligned}$$

It is a good exercise to implement such an evaluation operator for primitive recursive functions.

Alas, evaluation has an uninted and somewhat ominous side-effect: it forces us to consider partial functions. As a first step leading towards this conclusion, one may ask whether eval itself is a primitive recursive function. It seems reasonable to assume that eval is indeed primitive recursive, all the coding machinery is certainly primitive recursive and even elementary, and one only needs to follow the inductive definitions to evaluate a primitive recursive program. Unfortunately Cantor's diagonal method ruins this scenario. To wit, if eval is primitive recursive, then the function

$$f(x) = \mathsf{eval}(x, x) + 1$$

is also primitive recursive, and has an index $e \in \mathbb{N}$. Evaluating f on e produces

$$f(e) = \mathsf{eval}(e, e) + 1 \quad \text{but also} \quad f(e) = \mathsf{eval}(e, e)$$

a manifest contradiction.

The only way to avoid this issue is to give up on the illusion of all computable functions being total: some of these functions will necessarily fail to be defined on some inputs. Anyone who has ever written a program of sufficient sophistication will know that this is far from artificial: programs do fail to converge, though typically as a result of an error rather than by design. We have phrased our argument in terms of primitive recursive functions, but it generalizes directly to any reasonable clone $\mathcal{C}$ of total functions that allegedly captures the notion of computability.

One might hope that, at least, our primitive recursive functions represent all total computable function, but that does not hold up either. We can modify the last argument to produce a **2**-valued, total computable function that fails to be primitive recursive:

$$g(x) = \begin{cases} 0 & \text{if } \mathsf{eval}(x, x) \neq 0, \\ 1 & \text{otherwise.} \end{cases}$$

It follows that primitive recursive functions do not capture our intuitive notion of computability, even if we tried to restrict all functions to be total. Incidentally, this is exactly what happened in the early days of computability theory, the focus was entirely on total computable functions until Kleene introduced his μ-recursive functions, see section 8.2. As always, diagonalization arguments are slightly annoying: we would prefer a concrete example of a computable function that fails to be primitive recursive. In the next section we will introduce computable functions with enormously large growth rates, so large that they fall outside of the realm of primitive recursive functions. These examples also provide some insight in the size of the gap there is between primitive recursion and computability.

Fast Growing Functions

As we will see, there is huge gap between primitive recursion and general computability, even for total functions, but it is surprisingly difficult to exhibit a concrete example of an arithmetic function that is clearly computable but fails to be describable in terms of primitive recursion. We will give two solutions to this problem.

The first concrete example was found by Hilbert's student W. Ackermann in 1928. The version presented here is due to R. Péter and differs slightly from the original, ternary function. We define a binary function $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by double recursion as follows. For emphasis we have written x^+ rather than $x + 1$ to indicate the places where recusion takes place.

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

Note the double recursion–a construct that looks more complicated than primitive recursion. In fact, it is not entirely clear that $A(x, y)$ is well-defined for all x and y. Even for small arguments, it is quite tedious to produce the function value. For example, the computation of $A(2, 1)$ looks like so:

$$\begin{aligned} A(2, 1) &= A(1, A(2, 0)) \\ &= A(1, A(1, 1)) \\ &= A(1, A(0, A(1, 0)) \\ &= A(1, A(0, A(0, 1)) \\ &= A(1, A(0, 2)) \\ &= A(1, 3) \\ &= A(0, A(1, 2)) \\ &= \ldots \\ &= 5 \end{aligned}$$

More calculations along these lines suggest that A is indeed computable, we will see a stronger justification in a moment. Informally, note that most modern programming languages consider double recursion to be a perfectly admissible construct. For example, here is a slightly cryptic bit of C code that implements the Ackermann function (at least for machine-sized integers), using ?, the conditional expression operator.

```
int acker(int x, int y) {
    return( x ? ( acker( x-1, y ? acker( x, y-1 ) : 1 )) : y+1 );
}
```

This code compiles and executes without a problem, and correctly computes at least a few small values of the Ackermann function. For larger values, limitations of machine sized integers and stack size quickly crash the computation and memoizing does little to help with resource issues.

We need to show that the recursion equations for A define a total function. In order to do so we need to argue about solutions of a system of equations and how these solutions might be obtained. We will discuss these matters more carefully in section 8.5.1, for the time being we rely on intuitive ideas.

Proposition 8.1.2 *For all $x, y \in \mathbb{N}$, there exists precisely one $z \in \mathbb{N}$ such that the equation $A(x, y) = z$ is derivable from Ackermann's equations.*

Proof. The proof is by induction on x, and subinduction on y. This comes down to induction on $\mathbb{N} \times \mathbb{N}$ with respect to the lexicographic product order

$$(x, y) \prec (x', y') \iff (x < x') \lor (x = x' \land y < y'),$$

a well-order of order type ω^2, see section 5.2. It is easy to see that the arguments appearing on the right-hand-side of the equations are always $\prec$-less than the arguments on the left-hand-side. □

So we are dealing with a total function. Let us take a closer look at how the values of the Ackermann function are computed. We claim that there is an elementary function $\varDelta$ that naturally describes each step in the computation of A. Here is the computation of $A(2, 1)$ from above again, with all the syntactic sugar stripped away:

$$(2,1) \rightsquigarrow (1,2,0) \rightsquigarrow (1,1,1) \rightsquigarrow (1,0,1,0) \rightsquigarrow (1,0,0,1) \rightsquigarrow (1,0,2) \rightsquigarrow (1,3) \rightsquigarrow \ldots \rightsquigarrow (5)$$

Figure 8.5 shows the length of the lists that arise in the computation of $A(3, 4) = 125$. Note the very regular behavior.

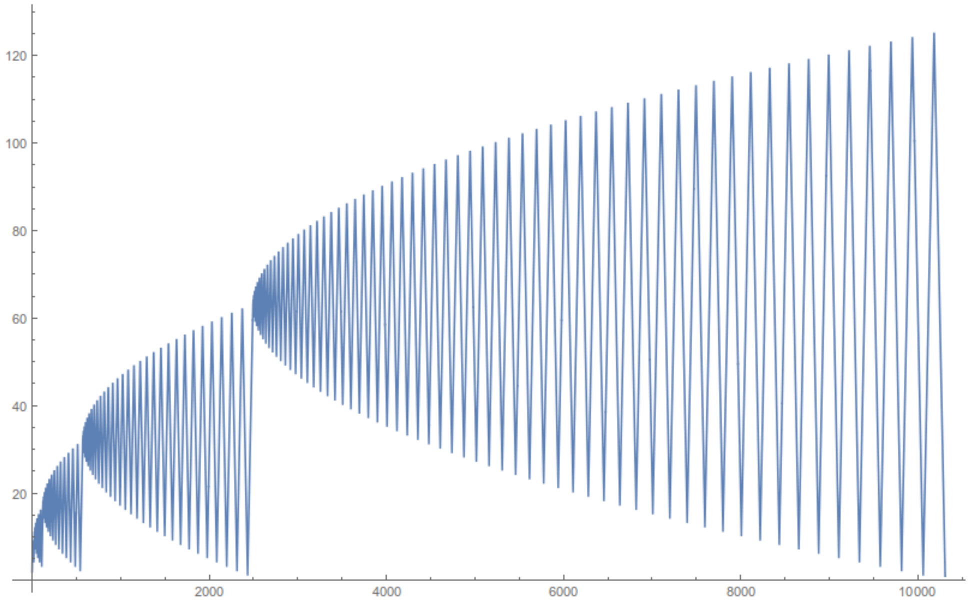


Figure 8.5: The computation of $A(3, 4)$ using the list operator $\varDelta$.

We can model this behavior in terms of an operation $\varDelta$ on lists of naturals. Strictly speaking, $\varDelta$ should be defined on sequence numbers coding these lists, but we will simply pretend that $\varDelta$ is multiadic instead. Thus $\varDelta$ operates on lists of length at least 2 as follows:

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

A well-order argument similar to the one above shows that for every a, b there is a stage σ such that $\varDelta^\sigma(a, b) = (c)$ where $A(a, b) = c$. Thus the computation of the Ackermann function can be reduced to iterating an elementary function. However, as we will see shortly, there is no hope to bound the number of iterations necessary by an elementary or even primitive recursive function.

Note our sequence-of-lists approach corresponds very closely to the recursion stack that would be used in the computation of the Ackermann function as in the snippet of C code from above. Since we can code the sequence of lists produced by Δ into a single number, we can compute the Ackermann function like so:

$$\begin{aligned}\mathsf{acker}(x,y) &= \min\big(\,\sigma \mid \mathsf{len}\Delta^{\sigma}(x,y) = 1\,\big)\\ A(x,y) &= \Big(\Delta^{\mathsf{acker}(x,y)}(x,y)\Big)_0\end{aligned}$$

This suggests that we could add unbounded search to our collection of admissible operations to capture functions defined by systems of equations such as the Ackermann function.

So, the Ackermann function is intuitively computable, but could it actually be primitive recursive? To answer this question it is useful to apply currying and think of Ackermann's function as a family of unary functions $(A_x)_{x\geq 0}$ where $A_x(y) = A(x,y)$. The critical part of the definition then looks like so:

$$A_{x^+}(y) = \begin{cases} A_x(1) & \text{if } y = 0,\\ A_x(A_{x^+}(y-1)) & \text{otherwise.}\end{cases}$$

Thus the next level is obtained by iteration: $A_{x^+}(y) = A_x^{y+1}(1)$. It follows by induction that each function A_x is primitive recursive. Alas, these functions grow more and more rapidly as x increases:

$$\begin{aligned}A(0,y) &= y^+\\ A(1,y) &= y^{++}\\ A(2,y) &= 2y+3\\ A(3,y) &= 2^{y+3}-3\\ A(4,y) &= 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}-3\\ A(5,y) &\approx \text{ super-super exponentiation}\\ A(6,y) &\approx \text{ nameless horror}\end{aligned}$$

Referring to $A(6,y)$ as a nameless horror is not quite justified, there are interesting notation system that make it possible to express these numbers such as Knuth's up-arrow notation or Conway's chain-arrow notation–but they do not really provide much more insight than our definition. The first five values of the diagonal function $A(x,x)$ are

$$1, 3, 7, 61, 2^{2^{2^{65536}}} - 3$$

From the table, one might suspect that the Ackermann function is a purely theoretical construct whose only purpose it is to refute the conjecture that computable is the same as primitive recursive. Surprisingly, this function appears in the analysis of the well-known union/find algorithm (with ranking and path-compression), a method to compute dynamic equivalence relations, see section 2.2.2. The running time of union/find differs from linear only by a minuscule amount, which amount is something like the inverse of the Ackermann function. Still, in general anything beyond level 3.5 of the Ackermann hierarchy is irrelevant for practical computation.

For a proof that Ackermann's function is not primitive recursive one exploits its rapid growth. Let us say that A dominates some arithmetic function $f : \mathbb{N}^n \to \mathbb{N}$ at level k if $f(\boldsymbol{x}) < A(k, \widehat{\boldsymbol{x}})$. Here $\widehat{\boldsymbol{x}}$ is defined as $\max(x_1, \ldots, x_n)$, $\boldsymbol{x}$ in $\mathbb{N}^n$.

Theorem 8.1.1 *The Ackermann function dominates every primitive recursive function. As a consequence, A is not primitive recursive.*

To see why the claim of the theorem shows that A is not primitive recursive, suppose otherwise. Then the diagonal function $\alpha(x) = A(x,x)$ is also primitive recursive. But then A dominates α, say, at level k. It follows that $\alpha(x) < A(k,x)$ for all x and $\alpha(k) < A(k,k) = \alpha(k)$, a contradiction. The proof of the claim is relatively simple, albeit tedious: one constructs a suitable k by induction on the build-up of f. First, we need to establish a number of monotonicity properties of A.

Lemma 8.1.14

1. $y < A(x,y)$
2. $y < y'$ *implies* $A(x,y) < A(x,y')$
3. $A(x,y^+) \leq A(x^+,y)$
4. $x < x'$ *implies* $A(x,y) < A(x',y)$
5. $A(x,2y) < A(x^{++},y)$

Proof.

(1) Induction on x, sub-induction on y, the case $x = 0$ being trivial. $x+1$: A is always positive, thus the claim holds for $y = 0$. But $A(x^+,y^+) = A(x,A(x^+,y)) >_{(1)} A(x^+,y) > y$.

(2) It suffices to show that $A(x,y) < A(x,y^+)$. This is trivial for $x = 0$, for $x+1$ the claim follows from (1) and the definition of A, case 3.

(3) Induction on y. The base case is part 2 of the definition of A. Step: $A(x^+,y^+) = A(x,A(x^+,y)) \geq_{(2)} A(x,A(x,y^+)) \geq_{(1),(2)} A(x,y^{++})$.

(4) $A(x^+,y) \geq_{(3)} A(x,y^+) >_{(2)} A(x,y)$; rest by induction on x'.

(5) Induction on y. Base case: $A(x^{++},0) >_{(4)} A(x,0) = A(x,2y)$. Step: $A(x^{++},y^+) = A(x^+,A(x^{++},y)) >_{(2),IH} A(x^+,A(x,2y)) \geq_{(3)} A(x,A(x,2y)^+) \geq_{(1)} A(x,(2y+1)^+) = A(x,2y^+)$.

□

For the induction argument, it is easy to see that A dominates all the basic primitive recursive functions. The next two lemmata handle the inductive cases: composition and primitive recursion.

Lemma 8.1.15 *If A dominates h and g_1, $\ldots$,g_m, then A also dominates $f = h \circ (g_1,\ldots,g_m)$.*

Proof. Assume $A(k_0,\widehat{y}) > h(y)$ and $A(k_i,\widehat{\boldsymbol{x}}) > g_i(\boldsymbol{x})$. Set $k = max(k_i)+2$. Then $A(k,\widehat{\boldsymbol{x}}) \geq_{(3)} A(k-1,\widehat{\boldsymbol{x}}+1) = A(k-2,A(k-1,\widehat{\boldsymbol{x}}))$. But $A(k-1,\widehat{\boldsymbol{x}}) >_{(1),(4)} A(k_i,\widehat{\boldsymbol{x}}) > g_i(\boldsymbol{x})$, whence by (2) $A(k,\widehat{\boldsymbol{x}}) > A(k-2,\max g_i(\boldsymbol{x})) \geq_{(4)} A(k_0,\max g_i(\boldsymbol{x})) > h(g_1(\boldsymbol{x}),\ldots,g_m(\boldsymbol{x})) = f(\boldsymbol{x})$. Done. □

Lemma 8.1.16 *If A dominates h and g, then A also dominates* $\mathsf{Prec}[h,g]$.

Proof. Assume $A(k_g,\widehat{y}) > g(y)$ and $A(k_h,\max(\boldsymbol{x},z,\widehat{y})) > h(\boldsymbol{x},z,y)$ and set $k = \max(k_g,k_h)+3$.

Claim: $A(k-2,\widehat{\boldsymbol{y}}+\boldsymbol{x}) > f(x,\boldsymbol{y})$ where $f = Prec[h,g]$.

The proof is by induction on x. $x = 0$: $A(k-2, \widehat{\boldsymbol{y}}) > A(k_g, \widehat{\boldsymbol{y}}) > g(\boldsymbol{y}) = f(0, \boldsymbol{y})$ x^+: $A(k-2, \widehat{\boldsymbol{y}}+x) > f(x, \boldsymbol{y})$ by IH, hence $A(k-2, \widehat{\boldsymbol{y}}+x) > \max(x, \widehat{\boldsymbol{y}}, f(x, \boldsymbol{y}))$. So $A(k-2, \widehat{\boldsymbol{y}}+x^+) = A(k-3, A(k-2, \widehat{\boldsymbol{y}}+x)) > A(k_h, \max(x, \widehat{\boldsymbol{y}}, f(x, \boldsymbol{y}))) > h(x, f(x, \boldsymbol{y}), \boldsymbol{y}) = f(x^+, \boldsymbol{y})$.

Thus $A(k, \max(x, \widehat{\boldsymbol{y}})) >_{(5)} A(k-2, 2\max(x, \widehat{\boldsymbol{y}})) \geq A(k-2, \widehat{\boldsymbol{y}}+x) > f(x, \boldsymbol{y})$, the last inequality is just our claim. □

Goodstein Sequences

There are several other well-known rapidly growing computable functions that fail to be primitive recursive. Here is an example that defies intuition. Suppose we write a number in base 2, say

$$266 = 2^8 + 2^3 + 2$$

We can turn this into the hereditary binary expansion by writing the exponents also in base 2, and so on. In the example,

$$266 = 2^{2^{2+1}} + 2^{2+1} + 2$$

where we really should write 2^0 instead of 1. Now suppose we replace 2 everywhere by 3 in the last expression:

$$3^{3^{3+1}} + 3^{3+1} + 3$$

Starting with base $k \geq 2$, we refer to this as the base bump operation $\beta_k(n)$. In our example,

$$\beta_2(266) = 443426488243037769948249630619149892887 \approx 4 \times 10^{38}$$

Unsurprisingly, the new number is much larger. Note that $\beta_k(n) = n$ for $n < k$, but otherwise the function is strictly increasing. Given a natural number m, define the Goodstein sequence $G(m)$ of m as follows. It is convenient to use 1-indexing for this sequence: $G(m)(1) = m$ and

$$G(m)(i+1) = \begin{cases} \beta_{i+1}(G(m)(i)) - 1 & \text{if } G(m)(i) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Intuition will suggest that subtracting 1 at the end is nowhere near enough to compensate for the strong growth of the base bump functions. And yet:

Theorem 8.1.2 *All Goodstein sequences converge to 0.*

Proof. Associate a sequence $(\alpha_k)_{k\geq 1}$ of ordinal numbers with $G(m)$ as follows: replace $k+1$ in the base $k+1$ representation of this number by ω. One can easily check that (α_k) is strictly decreasing before it hits the fixed point 0. At this point, the original sequence must also be 0. □

While this proof of Goodstein's theorem is in a sense quite straightforward, it uses ordinals below ε_0 and cannot be handled in Peano arithmetic. In light of the theorem, we can define the Goodstein function $G_0(m)$ to be the least position i such that $G(m)(i) = 0$. Then G_0 is total and obviously computable. Alas, it is very hard to come up with good examples. For example, $G_0(3) = 5$ since $G(3) = 3, 3, 3, 2, 1, 0$. But

$$G(4) = 4, 26, 41, 60, 83, 109, 139, 173, 211, 253, 299, \ldots$$

and $G_0(4) \approx 10^{121,210,695}$.

To obtain reasonable descriptions of the values of the Goodstein function one needs to define hierarchies of fast growing functions, using transfinite induction along the lines of

$$\begin{aligned} f_0(n) &= n^+ \\ f_{\alpha+1}(n) &= f_\alpha^{n^+}(n) \\ f_\lambda(n) &= f_{\alpha_n}(n) \end{aligned}$$

where $\alpha_n \to \lambda$. All the finite levels f_n are primitive recursive and the Ackermann function is essentially f_ω. But, f_ω is dwarfed by functions higher up in the hierarchy such as f_{ω^2}. Worse, f_{ε_0} is a total function, but this fact cannot be established in Peano arithmetic.

Self-Avoiding Words

One last example of a computable function with an absurd growth rate, due to Harvey Friedman. Recall that a finite word u is a subsequence of a word v, if $1 \leq i_1 < i_2 < \ldots i_n \leq m$ of positions in v such that $u = v_{i_1} v_{i_2} \ldots v_{i_n}$ (we assume 1-indexing in this section). In other words, we can erase some letters in v to get u. We write $u \sqsubseteq v$ to indicate subsequence order. Note that subsequence order is independent of any underlying order of the alphabet, unlike, say, lexicographic or length-lex order. The structure of the order depends entirely on the cardinality of the underlying alphabet. There is a surprising theorem concerning these orders due to Higman.

Theorem 8.1.3 (Higman 1952) *Every antichain in the subsequence order is finite.*

Proof. Here is the Nash-Williams proof (1963): assume there is an infinite antichain. Then there is a non-increasing sequence $\boldsymbol{x} = (x_n)$ in the sense that $i < j$ implies that $x_i \not\sqsubseteq x_j$.

Choose the minimal such sequence in the sense that x_n is the length-lex minimal word such that $x_0, x_1, \ldots, x_n$ starts a non-increasing sequence. There must be a letter a such that the subsequence $x_{n_j} = a\, y_j$, $j \geq 0$, of words starting with a is infinite. Let $k = n_0$ and define a new sequence

$$\boldsymbol{z} = x_0, x_1, \ldots, x_{k-1}, y_0, y_1, \ldots$$

contradicting minimality. □

For a finite or infinite word x, write $x[i]$ for the block $x_i x_{i+1} \ldots x_{2i}$. We will always assume that $i \leq |x|/2$ when x is finite to make sure that this block of length $i+1$ actually exists. Here is a somewhat strange definition due to Friedman: a word x is self-avoiding if, for $i < j$, the block $x[i]$ is not a subsequence of block $x[j]$.

Over a one-letter alphabet, the longest possible self-avoiding word clearly is aaa. For two letters, with some amount of effort, one can check that $x = abbbaaaaaaa$ is the longest self-avoiding word: if we append an a, then $x[5]$ is a subsequence of $x[6]$; if we append an b, then $x[1]$ is a subsequence of $x[6]$. The following is an easy consequence of Higman's theorem.

Corollary 8.1.3 *Every self-avoiding word is finite.*

Now consider an alphabet of size k. By the corollary and Kőnig's lemma, the set S_k of all finite self-avoiding words must itself be finite. But then we can define the following function:

$$\alpha(k) = \max\big(\, |x| \mid x \in S_k \,\big)$$

Note that α is utterly straightforward to compute: self-avoiding words are closed under prefixes, and thus form a finite subtree of $\Sigma^\star$, or a trie in computer science parlance. We can build this trie in stages, starting at the root. For example, a simple implementation of this algorithm computes the number of self-avoiding words of length up to 12, for alphabets up to size 4.

1	2	3	4	5	6	7	8	9	10	11	12
1	1	1	0	0	0	0	0	0	0	0	0
2	4	8	8	16	12	24	4	8	2	4	0
3	9	27	60	180	348	1044	1518	4554	5334	16002	16674
4	16	64	216	864	2688	10752	29376	117504	285108	1140432	2569248

We have already seen that $\alpha(1) = 3$ and $\alpha(2) = 11$. To get a lower bound on the next value $\alpha(3)$, we have to resort to an Ackermann-like function:

$$B_1(x) = 2x$$

$$B_{k^+}(x) = B_k^x(1)$$

Then, according to Friedman, we have

$$\alpha(3) > B_{7198}(158386)$$

a mind-numbingly large number. There is even an upper bound: $\alpha(3) < B(B(5))$ where $B(x) = B_x(x)$ is the diagnoal of the $B_k(x)$ functions. This upper bound is even more incomprehensible than the lower bound, there is no connection betweeen numbers of this magnitude and the physical universe, presumably the place where our intuition about computation originates. Not even the tiniest fraction of a computation of this kind could be realized physically, even if we could compute on the Planck-scale and work from the big-bang to the heat-death of the universe.

Perhaps even more disconcerting is the fact that $\alpha(3)$ is quite small compared with other fast-growing computable functions. The central idea in Friedman's argument is to construct a sequence of fairly simple objects $X_1, X_2, \ldots, X_n$ so that X_i cannot be embedded in some sense into X_j, for all $1 \leq i < j \leq n$. For the α function, these objects are the blocks in a word over an alphabet of some fixed size. Higman's theorem guarantees that there is some finite bound on the length of all such sequences, but the bound is incomprehensibly large. Another well-known example of this kind is the infamous TREE function that is based on constructing a sequence of non-embedding trees formed with nodes of a fixed number k of colors. By a famous theorem of Kruskal, for any k there is a sequence of maximum length, but the value of $\mathsf{TREE}(3)$ dwarves $\alpha(3)$.

8.2 Partial Recursive Functions

The examples of rapidly growing, yet intuitively computable functions make it clear that we need to extend our notion of primitive recursive functions if we want to capture computation in its intuitive sense. Since primitive recursive functions are quite powerful on their own, one should try to determine where exactly they fall short. We know already that we will have to introduce partial functions at some point, but in and of itself this is not a particularly helpful observation.

A closer look at the algorithm that determines the longest self-avoiding word over a k-letter alphabet reveals that all the required string operations are easily primitive recursive, really just so much wordprocessing. The part that is not covered by primitive recursion is the outer loop that searches for the first empty level in the trie of self-avoiding words. The same is true for the Ackermann function and Goodstein sequences, in both cases there is primitive recursive procedure that we have to apply repeatedly until some condition is met. In the context of programming languages this would be called a while loop, see section 8.3.2. For arithmetic functions we can obtain a similar effect using the following minimization operator.

Definition 8.2.1 (Minimization Operator)
Let $g : \mathbb{N}^{n+1} \nrightarrow \mathbb{N}$ be an arithmetic function. Define a minimization operator $\mathsf{Min}(g) = f$ as follows: $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ where

$$f(\boldsymbol{x}) = \min\big(\, z \mid g(z, \boldsymbol{x}) = 0 \land \forall\, s < z\, (g(s, \boldsymbol{x}) > 0)\,\big)$$

The μ-recursive functions or partial recursive functions or general recursive functions are the clone generated by the same basic functions and operations as the primitive recursive functions, but with the addition of the minimization operator.

Note that primitive recursion still only applies to total functions, it cannot be used on the partial functions that will generally result from an application of the Min operator. One should think of the Min operator as unbounded search, we are looking for the least number a such that some property holds of a. In order to be able to pronounce a specific a to be the correct answer, we have to establish that a has the property, but also that all smaller numbers fail to have it. This is the reason why we technically require $g(s, \boldsymbol{x})$ to have some positive value for all $s < z$. As a consequence, $\mathsf{Min}(g)$ is computable whenever g is computable. Observe that, with a view towards computability, we cannot simply change the definition to just demanding that $g(z, \boldsymbol{x}) = 0$; in this case, a function g where $g(1, \boldsymbol{x}) = 0$ but $g(0, \boldsymbol{x}) \uparrow$ would not produce a result after any finite amount of computation. An alternative way to ensure computability of the minimization is to insist that $g : \mathbb{N}^{n+1} \to \mathbb{N}$ be total; this is used in several places in the literature. However, since we are using minimization as a closure operation, the given definition is slightly more natural, in particular since totality of a computable function is undecidable.

In the presence of partial functions it is advisable to use a somewhat modified notion of equality that takes the possibility of divergence into account.

Definition 8.2.2 (Kleene Equality)
Given two partial recursive functions f and g, write

$$f(\boldsymbol{x}) \simeq g(\boldsymbol{x})$$

to mean that, either, both f and g are defined on $\boldsymbol{x}$ and return the same output, or, both diverge on $\boldsymbol{x}$. We also write $f(\boldsymbol{x}) \downarrow$ and $f(\boldsymbol{x}) \uparrow$ to indicate convergence and divergence, respectively.

Recall our coding mechanism for primitive recursive functions from section **??**. We can easily extend the code to include the Min operator.

0	$\langle 0, 0 \rangle$	zero
S	$\langle 1, 1 \rangle$	successor
$\mathsf{P}^{(\mathsf{n})}_{\mathsf{i}}$	$\langle 2, n, i \rangle$	projections
$\mathsf{Prec}[h, g]$	$\langle 3, n, \widehat{h}, \widehat{g} \rangle$	primitive recursion
$\mathsf{Comp}[h, g_1, \ldots, g_n]$	$\langle 4, m, \widehat{h}, \widehat{g_1}, \ldots, \widehat{g_n} \rangle$	composition
$\mathsf{Min}[g]$	$\langle 5, n, \widehat{g} \rangle$	minimization

In the last case, $n+1$ is the second component of $\widehat{g}$, the arity of g. Thus, we can effectively associate an index e with any partial recursive function. Following Kleene, we write

$$\{e\}(\boldsymbol{x}) \simeq y$$

to express the assertion that the partial recursive function with index e on input $\boldsymbol{x}$ produces output y.

According to our definition, the minimization operator could be used multiple times in the construction of a μ-recursive function. Assume that the computation of f involves unbounded searches over $z_1, z_2, \ldots, z_m$. It stands to reason that we could replace the m separate searches by a single search over z and m bounded sub-searches $z_i < z$. There are some technical details to fill in, see the exercises. As Kleene has shown, one can collect all the unbounded searches into just a single one, using a primitive recursive predicate to describe the property in question.

Theorem 8.2.1 *There is a primitive recursive predicate T and a primitive recursive function D so that for any partial recursive function with index e we have:*

$$\{e\}(\boldsymbol{x})\downarrow \iff \exists t\, T(e, \boldsymbol{x}, t)$$
$$\{e\}(\boldsymbol{x}) \simeq D\big(\min\big(\, t \mid T(e, \boldsymbol{x}, t)\,\big)\big)$$

Strictly speaking, we need a separate T predicate for each arity, or we could code $\boldsymbol{x}$ as a sequence number. In the next section we will discuss machine models of computation. In that context, one of Turing's breakthrough results was to show that there are universal machines, machines that can simulate all other machines of a certain type. Intuitively, we can think of T and D as being the analog of a universal machine in our program-based environment. To wit, t codes the complete computation of a universal machine on input program e and arguments $\boldsymbol{x}$, and D extracts the result of the computation. In essence, t is the number of steps of this computation, we can easily construct the actual computation from the step count in a primitive recursive fashion.

We will discuss the properties of partial recursive functions in greater detail in chapter 9.

8.3 Loop and While Programs

Primitive recursive functions can be construed as being defined by programs written in a particularly spartan programming language. Unlike actual programming language, this one has no data types other than natural number and no direct control structures. For example, it requires work to construct an if-then-else statement. In this section we will introduce two formal languages that are somewhat closer to actual programming languages, but still so primitive that their semantics is easy to describe.

8.3.1 Loop Programs

Our first programming language Loop has only one data type, namely natural numbers. Operations are limited to set-to-zero, increment and assignments. We allow sequential composition and one control structure corresponding to a for-loop. The program elements are described informally as follows:

initialize	$x := 0$
increment	$x{+}{+}$
assignments	$x := y$
sequential composition	$P; Q$
control	do x : P od

We could have avoided assignments, but that makes the classification of Loop programs slightly more tedious. The semantics are clear except for the loop construct do x : P od. This is intended to mean: "Let n be the value of x before the loop is entered; then execute P exactly n times." In other words, even if P changes the value of x, the number of executions will still be the same. We could get the same effect by not allowing x to appear in P. It follows that all loop programs terminate, regardless of the input. We will give a formal description of the semantics of a Loop program shortly, first a few typical examples. Arithmetic operations such as addition and multiplication are not built-in, but it is not hard to see how to implement them, at the cost of introducing a few loops. Here are addition and multiplication.

```
//  add : x, y --> z
        z := x;
        do y :
                z++;
        od

//  mult : x, y -> z
        z := 0;
        do x :
                do y :
                        z++;
                od
        od
```

The predecessor function is a bit more interesting; it requires a little trick, the use of an extra variable that lags behind a counter.

```
// pred : x -> z
        z := 0;
        v := 0;
        do x :
                z := v;
                v++;
        od
```

To get better control structures we first implement the sign function $\mathsf{sign}(x) = \min(x, 1)$. Here a loop is abused to perform a Boolean test.

```
// sign : x -> z
        z := 0;
        v := 0;
        v++;
        do x :
                z := v;
        od
```

Based on sign we can execute code conditionally: the following program executes P whenever x is positive (or true, in the standard identification of Boolean values and integers).

```
z = sign(x);        // macro
do z :
        P;
od
```

To explain the semantics of a Loop program in some detail, define an environment to be a map that assigns a value to all variables. Write $\mathcal{E}$ for all environments and $\mathcal{P}$ for all loop programs. Let $E[x \mapsto t]$ be the environment that is the same as E except that the value of E at x is set to t. Any single loop program P uses only finitely many variables, so all we need is a finite lookup table. We can think of the table as a collection of registers and we can construct an abstract machine that executes P as follows. We keep track of a sequence $\sigma = \sigma_1, \dots, \sigma_n$ of commands, the command stack, and the environment E. Thus, a configuration of the machine is given by the pair $(\sigma, E) \in \mathsf{List}(Cmd) \times \mathcal{E}$. The one-step function has the format

$$\mathsf{next} : \mathsf{List}(Cmd) \times \mathcal{E} \to \mathsf{List}(Cmd) \times \mathcal{E}$$

To perform one step, we pop σ_1 from the command stack and update stack and environment according to the following table. For basic commands, only the environment changes. For compound commands the environment is unaffected, but we push commands onto the stack.

σ_1	environment	stack
`x = 0`	$E[x \mapsto 0]$	$\sigma_2, \dots, \sigma_n$
`x++`	$E[x \mapsto E(x) + 1]$	$\sigma_2, \dots, \sigma_n$
`x = y`	$E[x \mapsto E(y)]$	$\sigma_2, \dots, \sigma_n$
`P; Q`	E	$P, Q, \sigma_2, \dots, \sigma_n$
`do x:  P od`	E	$\underbrace{P, \dots, P}_{E(x)}, \sigma_2, \dots, \sigma_n$

In order to compute with this machine we initialize the stack to contain the program P to be executed. We assume that P has a list of designated input variables $x_1, \dots, x_n$ and an output variable y. The initial environment E assigns the given values to the input variables and 0 to all other variables. After a sequence of moves the command stack becomes empty, at which point the execution of the program stops. $E(y)$, the final value of variable y, is the result of the computation.

The workings of our loop machine are slightly more complicated than necessary since it requires parsing of the loop program. Syntax analysis is fairly easy in this case, but we can avoid dealing with the command stack and directly define the semantics of loop programs via a function

$$[\![.]\!] : \mathcal{P} \to (\mathcal{E} \to \mathcal{E})$$

that maps environments to environments and explains precisely how the execution of a program fragment changes the contents of the variables. The definition of $[\![P]\!]$ is by induction on the buildup of P:

$$\begin{aligned}
[\![\mathtt{x} = \mathtt{0}]\!](E) &= E[x \mapsto 0] \\
[\![\mathtt{x} = \mathtt{y}]\!](E) &= E[x \mapsto y] \\
[\![\mathtt{x{+}{+}}]\!](E) &= E[x \mapsto E(x) + 1] \\
[\![\mathtt{P;Q}]\!](E) &= [\![Q]\!]([\![P]\!](E)) \\
[\![\mathtt{do\ x{:}\ P\ od}]\!](E) &= [\![P]\!]^{E(x)}(E)
\end{aligned}$$

As before, we compose $[\![P]\!]$ with injections $\iota : \mathbb{N}^n \to \mathbb{N}^k$ and projections $\pi : \mathbb{N}^k \to \mathbb{N}$ to define the usual input/output behavior. For example, if P is the program `do x:  x++ od` then $[\![P]\!](a) = 2a$. Likewise, the program Q given by

```
do x:
   do x: x++ od
od
```

has behavior $\llbracket Q \rrbracket(a) = \llbracket P \rrbracket^a(a) = a2^a$. Next we introduce the class of all functions computable by loop programs (under some reasonable input/output convention).

Definition 8.3.1 (Loop-Computability) *A function $f : \mathbb{N}^n \to \mathbb{N}$ is loop-computable if there is a Loop program P, an injection ι and a projection π such that $\pi \circ \llbracket P \rrbracket \circ \iota = f$.*

A relation $A \subseteq \mathbb{N}^n$ is loop-decidable if its characteristic function is loop-computable.

It is easy to see that loop-computable functions are closed under composition, so the question arises whether we can describe them directly as a clone by determining appropriate basic functions and closure operations. In particular, what is the relationship between loop-computable functions and our three clones of arithmetic functions: rudimentary, elementary and primitive recursive? At the top level we have the following result.

Theorem 8.3.1 *The loop-computable functions are precisely the primitive recursive functions.*

Proof. We will first show that loop-computable functions are closed under primitive recursion. For simplicity assume that there is only one parameter y. Suppose $P : y \to z$ and $Q : x, u, y \to z$ are two loop programs computing $g : \mathbb{N} \to \mathbb{N}$ and $h : \mathbb{N}^3 \to \mathbb{N}$. We show that $f = \mathsf{Prec}[g, h] : \mathbb{N}^2 \to \mathbb{N}$ is loop-computable by constructing a loop program for f directly. Here is a program `Pf` for f. Variable s is new.

```
// Pf:   x, y --> z
    s := x;
    x := 0;
    P;                 // y --> z
    do s:
           u := z;
           Q;          // x,u,y --> z
           x++;
    od
```

It is a good exercise to determine the semantics $\llbracket \texttt{Pf} \rrbracket$ in detail. Next we have to prove that application of a loop operation preserves primitive recursiveness. Consider a loop program P with variables $x_1, x_2, \ldots, x_k$ and semantics $f = \llbracket P \rrbracket : \mathbb{N}^k \to \mathbb{N}^k$. We may assume by induction that $f = (f_1, \ldots, f_k)$ is primitive recursive in the sense that each of the f_i is so p.r. Now consider program `Q:  do x: P od`. For simplicity, assume $x = x_1$. Define the vector valued function

$$\begin{aligned} F(0, \boldsymbol{x}) &= \boldsymbol{x} \\ F(n+1, \boldsymbol{x}) &= \llbracket P \rrbracket(F(n, \boldsymbol{x})) \end{aligned}$$

Using sequence numbers it is easy to show that all the component functions of F are primitive recursive. But then $\llbracket Q \rrbracket(\boldsymbol{x}) = F(x_1, \boldsymbol{x})$ is primitive recursive, as required. □

Thus, loops and primitive recursion are in a sense interchangeable. We can get even better results by distinguishing between loop programs of different complexity: each program has a particular nesting depth of loops.

Definition 8.3.2 *Let* Loop(k) *be the class of loop programs that have nesting depth at most k, the rank of the program. A function is* Loop(k)*-computable if it can be computed by a program in* Loop(k)*. We refer to the least such k as the rank of the function.*

Level 0 of this hierarchy is not very interesting: it is easy to see that any scalar function that is Loop(0)-computable is of the form

$$f(\boldsymbol{x}) = c \cdot x_i + d$$

where c and d are both constant natural numbers (and perhaps $c = 0$). The next level, Rank 1 turns out to be much more interesting. We have seen that addition and predecessor are Loop(1)-computable. It is easy to modify the argument for conditionals above to show that if-then applied to Loop(1)-computable functions produces another Loop(1)-computable function. Since Loop(1) is trivially closed under composition we are missing only remainders and integer division with fixed modulus to show that all the basic rudimentary functions are Loop(1)-computable. The following programs compute the remainder for modulus 2 and the integer quotient $x/2$. The loop bodies below implement a swap of u and v.

```
// mod2 : x --> u
        u := 0;
        v := 1;
        do x :
                t := u; u := v; v := t;
        od

// div2 : x --> u
        u := 0;
        v := 0;
        do x :
                u++;
                t := u; u := v; v := t;
        od
```

Theorem 8.3.2 *A function is* Loop(1)*-computable if, and only if, it is rudimentary.*

Proof. We have just verified that rudimentary programs are Loop(1)-computable. The other direction is more difficult: every Loop(1) program is already rudimentary. For the proof, consider a single loop `Q: do x: P od`. Suppose the variables are $\boldsymbol{x}$ where $x = x_1$, as before. The loop body `P` is Loop(0), so the effect of executing it `P` is a linear function on each of the variables:

$$x'_i = c_i \cdot x_{p(i)} + d_i$$

where $c_i \in \{0, 2\}$ and $d_i \geq 0$ constant. Here $p : [n] \to [n]$ is an arbitrary dependency map that indicates how values are passed from one variable to another in an assignment `xi = xj;`

To deal with assignments of the form `xi = 0;` it is convenient to introduce a phantom variable x_0 whose value is fixed at 0. Then we can rewrite the new values as

$$x'_i = x_{p(i)} + d_i$$

Now consider the dependency graph $G = \langle V, E \rangle$ where $V = \{x_0, x_1, \ldots, x_n\}$ and $E = \{\, (x_j, x_i) \mid j = p(i) \,\}$. Thus, there is a path in G from x_j to x_i if the value of x_j propagates to x_i, provided the loop is executed sufficiently often. Note that dependency graphs are unicyclic: each connected component contains exactly one cycle. But then the values of the registers on a cycle at time t (after t executions of the loop) are of the form

$$x = \begin{cases} a_0 & \text{if } t = 0, \\ a \cdot (t \operatorname{div} l) + \sum_{i<t \bmod l} a_i & \text{otherwise.} \end{cases}$$

where l is the length of the cycle, and the a_i are the labels, a being the sum of all these labels. Hence, the final value of a variable after execution of `P` is a rudimentary function of the initial values. Since rudimentary functions are closed under composition, the whole Loop(1) program can only compute a rudimentary function. □

8.3.2 While Programs

We saw in section 8.3.1 that the primitive recursive functions are exactly the functions computable by Loop programs. What do we have to add to Loop to obtain all partial recursive functions? The key idea is to replace the do-loop by a while-loop:

```
while x do:
        P;
od
```

The meaning of this construct is that P will be executed repeatedly until x attains the value 0, if ever. Note that for this to make sense we have to allow x to appear in P. Since we are looking for a stronger language we have to make sure we can at least capture ordinary Loop programs. To this end we add the predecessor function to the built-ins. Clearly,

```
while x do:
        P;
        x--;
od
```

behaves just like an ordinary loop, assuming again that x does not occur in P. It follows immediately that all primitive recursive functions are while-computable.

Theorem 8.3.3 *A function partial recursively enumerable if, and only if, it is while-computable.*

Sketch of proof. Accodring to Kleene's normal form result we can conduct just one unbounded search over a primitive recursive predicate. The latter can be expressed by a Loop program, so we only need to wrap that program into a single while-loop to obtain any partial recursive function.

For the opposite direction we need to simulate the evaluation of a while program in terms of partial recursive functions. This is very similar to our discussion of environments that explain the semantics of Loop programs and involves the usual coding machinery. □

8.3.3 Exercises

Exercise 8.3.1 Find another elementary pairing function with elementary unpairing functions.

Exercise 8.3.2 Show that for any prime p the function

$$\nu_p(x) = \max\big(\, e \mid p^e \text{ divides } x \,\big)$$

is elementary.

Exercise 8.3.3 Prove that the coding and decoding functions according to Gödel's are elementary.

Exercise 8.3.4 Explain how to implement search in binary search trees as a primitive recursive operation.

Exercise 8.3.5 Show that $f(x, \boldsymbol{y}) = \sum_{z<h(x)} g(z, \boldsymbol{y})$ is primitive recursive when h is primitive recursive and strictly monotonic.

Exercise 8.3.6 Prove the lemma on course-of-value recursion. You may safely assume that standard sequence operations such as append are primitive recursive.

Exercise 8.3.7 Consider the class $\mathcal{E}$ of functions defined by closing constant 0, successor, projections, addition, proper subtraction, multiplication, exponentiation under composition and bounded search. Show that $\mathcal{E}$ is exactly the class of elementary functions.

Exercise 8.3.8 Here is an example of a double recursion that produces a very simple function. Let $f(x) = x - 10$ for $x > 100$ and $f(x) = f(f(x + 11))$ otherwise. Give a simple description for f.

Exercise 8.3.9 Using only the primitive recursive definition, show that addition is associative and commutative.

Exercise 8.3.10 Using only the primitive recursive definition, show that multiplication is associative and commutative.

Exercise 8.3.11 Analyze the H function from example 8.1.2.

Exercise 8.3.12 Design a Loop program that computes the nth prime.

Exercise 8.3.13 Determine the semantics of the Loop program

```
u = 0; v = 0;
do x :
       u++; t = u; u = v; v = t;
od
```

Exercise 8.3.14 Study the equivalence relation used in the proof of theorem **??** for the particular function

$$f(x, y) = x + x \bmod 2 + y \operatorname{div} 2 + 1.$$

Exercise 8.3.15 Explain in detail why equivalence of $\mathsf{Loop}(2)$ programs is undecidable.

Exercise 8.3.16 Show that, for while programs with Booleans, a single loop indeed suffices.

8.4 Machine Models

Program-based models of computation have the great advantage that it is fairly easy to show that complicated operations, say, in number theory, are in fact computable. What is mostly missing is a simply connection to physically realizable computation: we want to think of actual devices composed of, say, a finite state control, some registers, input/output mechanisms and the like. These devices should be capable of executing our formal programs, at least in principle. The motivation in terms of real physical devices is very helpful when it comes to intuition, but we will focus on the underlying mathematical structure, not the engineering aspects. To this end we first introduce the notion of a configuration space that will serve to make our machine models precise and to provide a common framework for all of them.

8.4.1 Configuration Spaces

A general way to describe any sort of computing machine is to develop a notion of a configuration or instantaneous description, a snapshot that records all the information necessary to resume an interrupted computation.

Definition 8.4.1 *A configuration space is a discrete set $\mathcal{C}$ together with a relation $\rightarrow$, the next-step relation. We write $\mathfrak{C} = \langle \mathcal{C}, \rightarrow \rangle$ for configuration spaces.*

For the time being we are only concerned with deterministic models of computation where $\rightarrow$ is required to be a partial function, see the chapters on finite state machines and complexity theory for the general case. Hence, for any configuration C, there is at most one configuration C' such that $C \rightarrow C'$. We allow the next-step relation to be partial mostly as a matter of convenience, we could always add a "failure" configuration and extend $\rightarrow$ accordingly. Similarly, since $\mathcal{C}$ is discrete, we could use the coding machinery from section 8.1.3 to convert $\mathcal{C}$ to a set of integers, but it is far more natural to use basic data structures such as numbers, strings, lists and the like directly.

In the next few sections we will introduce several types of machines such as register machines and Turing machines. For any of these machines $\mathcal{M}$, there will be an associated configuration space $\mathfrak{C}(\mathcal{M}) = \langle \mathcal{C}_{\mathcal{M}}, \rightarrow_{\mathcal{M}} \rangle$ that explains how the machine performs computations, at least if we ignore input/output conventions. For two configurations C and D we let

$$C \mathrel{\big|\!\frac{0}{\mathcal{M}}} D \Leftrightarrow C = D$$

$$C \mathrel{\big|\!\frac{t}{\mathcal{M}}} D \Leftrightarrow \exists\, C' \left(C \mathrel{\big|\!\frac{t-1}{\mathcal{M}}} C' \wedge C' \rightarrow_{\mathcal{M}} D \right)$$

$$C \mathrel{\big|\!\frac{*}{\mathcal{M}}} D \Leftrightarrow \exists\, t\ C \mathrel{\big|\!\frac{t}{\mathcal{M}}} D$$

If one thinks of $\mathfrak{C}(\mathcal{M})$ as a directed graph, then $C \mathrel{\big|\!\frac{t}{\mathcal{M}}} D$ means there is a path from C to D of length t. Likewise, $C \mathrel{\big|\!\frac{*}{\mathcal{M}}} D$ means that D is reachable from C.

Definition 8.4.2 *A computation or run of $\mathcal{M}$ on configuration C is a maximal path in $\mathfrak{C}(\mathcal{M})$ starting at C. A configuration is terminating or halting if it has no next configuration; similarly a computation is terminating if it ends in a terminating configuration.*

Hence a computation is terminating iff it is finite as a path in $\mathfrak{C}$. We will also informally talk about convergent and divergent computations, though there is little connection between our definitions and numerical mathematics. In the following sections on register machines and

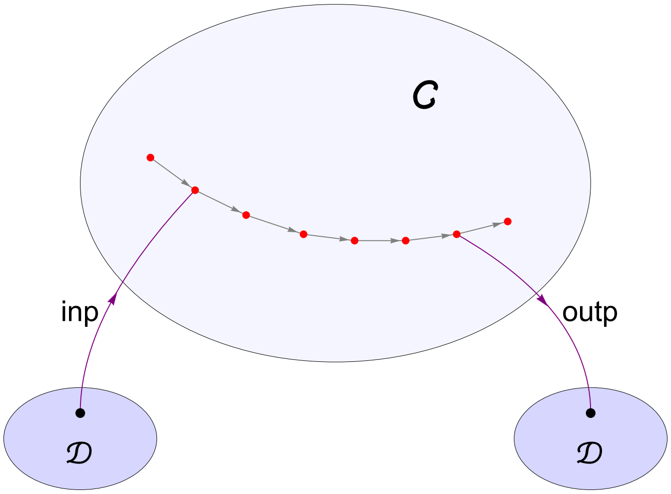
Turing machines we will see that divergent computations are impossible to avoid in the sense that any sufficiently powerful model of computation will have to allow for divergent computations.

The last ingredient required for a formal definition of computation is to mediate between the actual data one wishes to compute with, and the internals of the model. The precise nature of the data in question depends on circumstances, but we would certainly want to handle integers, strings, nested lists, graphs and so on; all sorts of discrete objects. Say we have a domain $\mathcal{D}$ of such objects, we need to determine an input map inp and an output map outp of the form

$$\begin{aligned} \mathsf{inp} &: \mathcal{D} \longrightarrow \mathcal{C} \\ \mathsf{outp} &: \mathcal{C} \longrightarrow \mathcal{D} \end{aligned}$$

To compute with a particular object $\boldsymbol{x} \in \mathcal{D}$ as input, we map $\boldsymbol{x}$ to $\mathsf{inp}(\boldsymbol{x}) = C_0$ and follow a path $C_0, C_1, \ldots, C_n$ in $\mathfrak{C}$. The range of inp is the collection of all initial configurations, the nodes in the space where meaningful computations start. The output is considered to be $\mathsf{outp}(C_n)$ where the configuration C_n is of a special kind of configuration, a terminal configuration or halting configuration, in which a result has been obtained. For example, a terminal configuration may not have a next configuration. Or the machine might be a particular state that indicates completion. If no such terminal configuration is ever reached, there simply is no output. While it makes sense to insist that inp be total, the output map outp is usually partial.

For the domain of arithmetic, the situation can be visualized like so:



In all the examples we consider below, the configuration space, the next-step relation, initial and terminal configurations will always be very simple. If we were to code all the objects as integers, primitive recursive functions and relations would suffice; in fact, elementary functions are good enough. The heavy lifting in a calculation is always handled by the machine, not by input/output conventions.

8.4.2 Register Machines

Perhaps the most straightforward way to construct a machine model for general computation is to use devices that operate directly on natural numbers, as opposed to manipulating some symbolic representation thereof. As a consequence, the input/output maps will be particularly simple. Thus, our first model will have so-called registers, each holding a natural number. Operations on these registers are limited to a few very primitive arithmetic operations, plus a control structure that combines a zero-test with a goto instruction. The operation of these machines is very natural and easy to follow, yet, as we will see, the machines are capable of implementing any intuitively computable arithmetic function whatsoever.

We enumerate the registers in our machines as R_i, $i \geq 0$, and we write $[R_i] \in \mathbb{N}$ for the content of the ith register. In any particular machine, the number of registers is fixed, but there is no universal bound on the number of registers. Plausible arithmetic and logical operations on registers are

- increment a register,
- test a register for 0,
- decrement a positive register,
- jump to an instruction,
- conditionally jump to an instruction.

There are many ways to select admissible instructions, we will here use a particularly spartan version that allows only the following three types of instructions. This choice makes programs slightly longer, but it does not reduce the power of our machines, and makes it easier to simulate them by other devices.

`inc r k`	increment register R_r, goto instruction k
`dec r k l`	if $[R_r] > 0$ decrement register R_r and goto instruction k, otherwise goto instruction l
`halt`	well ...

The goto instructions refer to line numbers in the program that provide targets for jump statements. More precisely, a register machine program (or RM-program for short) is a sequence $I_0, I_1, \ldots, I_{n-1}$ of such instructions, and goto k means that program execution should continue with instruction I_k. We refer to k as a state and to n, the number of instructions, as the size of the program. Similarly we will call r, the number of an register, an address. Note that there is no indirect addressing, instructions "increment the register with number $[R_r]$" or the like are not allowed. See exercise 8.4.10 for a model that includes indirect addressing. There is little point in trying to distinguish between a register machine and a register machine program, so we will use both terms interchangeably.

The semantics of a register machine program are fairly straightforward, but let us go through the definitions step by step. Suppose that $P = I_0, I_1, \ldots, I_{n-1}$ is a program of size n. We say that P is in standard form if the following holds: first, the states of all goto instructions are less than n; second, the registers used in the program are R_0, R_1, ..., R_{w-1} for some $w \leq n$; third, there is exactly one halt instruction, namely I_{n-1}, the last instruction in the program. exerc

Once we have given a description of the semantics of a register machine, it will be clear that we can assume without loss of generality that all machines are in standard form.

Definition 8.4.3 *A configuration of P is a pair $C = (p, \boldsymbol{x}) \in \{0, 1, \ldots, n-1\} \times \mathbb{N}^w$ indicating state and register contents. The next-step relation $(p, \boldsymbol{x}) \to_P (q, \boldsymbol{y})$ is defined as follows depending on the instruction I_p:*

I_p	*next configuration*
`inc r k`	$q = k$ *and* $\boldsymbol{y} = \boldsymbol{x}[r \mapsto x_r + 1]$
`dec r k l`	*if* $x_r > 0$*:* $q = k$ *and* $\boldsymbol{y} = \boldsymbol{x}[r \mapsto x_r - 1]$*;*
	if $x_r = 0$*:* $q = l$ *and* $\boldsymbol{y} = \boldsymbol{x}$
`halt`	*none*

By $\boldsymbol{x}[r \mapsto y]$ we mean the vector that is the same as $\boldsymbol{x}$, except that the rth component is changed to y. Since we assume P to be in standard form, only configurations with state $n-1$ fail to have a next configuration; we will consider those to be the terminal configurations in this model.

It remains to describe the input/output maps. Since registers contain natural numbers, these maps are particularly simple:

$$\begin{aligned} \mathsf{inp}(\boldsymbol{x}) &= (0; 0, \boldsymbol{x}, \boldsymbol{0}) \\ \mathsf{outp}(n-1; z, \boldsymbol{y}) &= z \end{aligned}$$

In other words, the input is presented in registers 1 through k and the result, if any, always accrues in register 0. All registers other than R_1 through R_k are initially set to 0. These conventions are somewhat arbitrary, but one can check that any reasonable modification will produce the exact same clone of register machine computable functions, see the exercises. At any rate, a program for addition now looks like so.

```
// addition    R1 R2 --> R0
  0:   dec 1  1  2
  1:   inc 0  0
  2:   dec 2  3  4
  3:   inc 0  2
  4:   halt
```

One can check that this program produces computations of the form

$$(0; 0, a, b) \vdash_{P}^{2a+1} (2; a, 0, b) \vdash_{P}^{2b+1} (4; a+b, 0, 0)$$

and duly implements addition. Some programs will fail to terminate on some inputs:

```
// P
    0:  dec 1 1 2
    1:  inc 0 1
    2:  halt
```

This program halts on input 0, with output 0, but it diverges an all positive inputs.

Definition 8.4.4 (Register Machine Computability)
A partial function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ is register machine computable or RM-computable if there exists a register machine P in standard form such that P has a terminating computation on $\mathsf{inp}(\boldsymbol{x})$ if, and only if, $f(\boldsymbol{x})$ is defined; in this case $\mathsf{outp}(C) = f(\boldsymbol{x})$ where C is the terminal configuration reached from $\mathsf{inp}(\boldsymbol{x})$.

We have used numerical values to denote program states and addresses which makes it easy to describe configurations. Alas, the programs are hard to read, it is better to use names such as X, Y, Z and so forth for the registers. We will spell out which of these symbolic registers are supposed to contain the input, and where the output will appear. Here is a program that multiplies the contents of registers X and Y, the result appears in Z; the program uses an auxiliary register U whose purpose it is to save the value for Y during the computation: we copy Y to Z a total of X times.

```
// multiplication   X Y --> Z
  0:   dec X  1  6
  1:   dec Y  2  4
  2:   inc Z  3
  3:   inc U  1
  4:   dec U  5  0
  5:   inc Y  4
  6:   halt
```

A yet better way to express register machine programs is in terms of flowgraphs. These are digraphs where the nodes are labeled $X+$ or $X-$ for each register X. The former kind has out-degree 1 and corresponds to an increment instruction. The latter has out-degree 2 and one of the edges is labeled by a 0 to indicate the second case of a decrement operation. Lastly, a node labeled H represents the halting instruction. We indicate instruction I_0 by placing the corresponding node in the top left corner of the diagram. Figure 8.6 shows the flowgraph for the multiplication machine from above. In each round, the machine destructively adds Y to Z and U, and then uses U to reconstruct Y. The number of rounds is given by X.

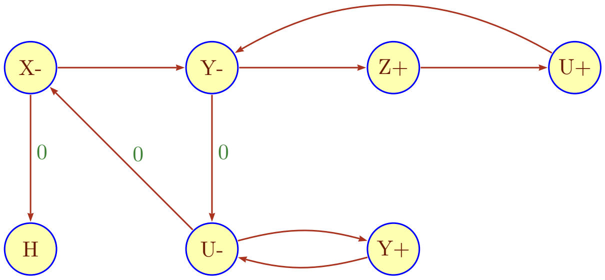


Figure 8.6: The flowgraph of a register machines that multiplies two numbers.

The state sequence of the run of this machine on inputs 3 and 6 are shown in figure 8.7. Note the cyclic behavior corresponding to loops in a programming language. A careful analysis of the image goes a long way towards determining the number of steps in the computation on general inputs n and m.

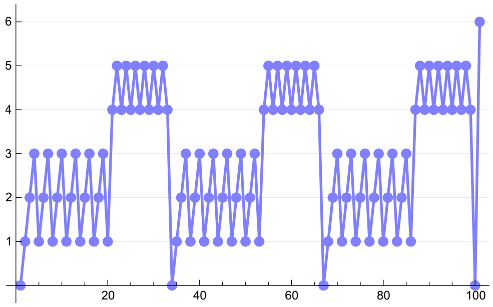


Figure 8.7: A run of the multiplication machine on inputs 3 and 6. The horizontal axis indicates time and the horizontal one state.

It is a labor of love to construct machines for other arithmetic functions such as exponentiation, factorial, square roots, gcd, nth prime and so on and so forth. Here is a slightly more ambitious example, a RM program that determines the binary digit sum of the input (the number of 1's in the binary expansion of register X). The output appears in register Z.

```
// binary digitsum of X --> Z
  0:   dec X  1  4
  1:   dec X  2  3
  2:   inc Y  0
  3:   inc Z  4
  4:   dec Y  5  8
  5:   inc Y  6
  6:   dec Y  7  0
  7:   inc X  6
  8:   halt
```

Alternatively, the flowgraph for this program is shown in figure 8.8.

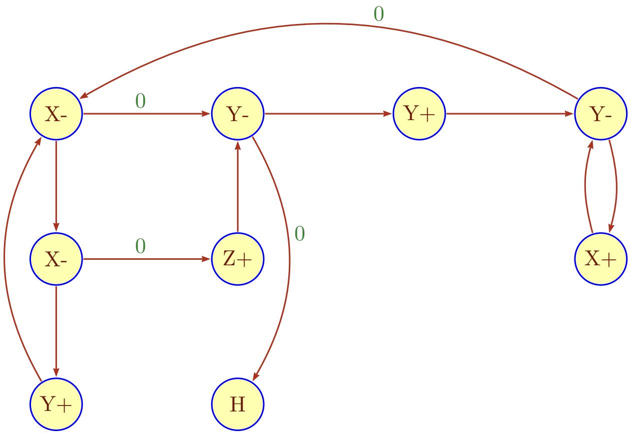


Figure 8.8: The flowgraph of a register machines that computes the binary digit sum.

As a last example, recall the coding function from section 8.1.3. Our register machines are simply not equipped to deal with multiadic functions, but we can consider slices $\mathbb{N}^k \to \mathbb{N}$. For decoding, suffice it to say that figure 8.9 shows the flowgraph of a register machines computing dec. It is a bit surprising that a register machine of size 9 already suffices.

It is a good exercise to construct more examples of register machines that implement arithmetic functions $f : \mathbb{N}^n \to \mathbb{N}$. In fact, one should be fairly hard pressed to find any arithmetical function in a number theory textbook that fails to be RM-computable.

8.4.3 Universality and Halting

In this section we will give a fairly detailed proof of one of the most fundamental results in computability theory, due to Turing: the existence of universal machines that are capable of simulating every other machine in the same class. A machine A simulates another machine B if every computation of B corresponds to a computation of A that produces the same result. Moreover, there is a uniform way in which steps of B are translated into–possibly multiple–steps of A. In the most straightforward scenario, the simulation is lockstep: every step of B corresponds to a fixed number of steps of A. We will make no attempt to formalize the concept of a simulation, it is quite difficult to do so and their is little to gain from such an effort; it will be intuitively clear in all cases what we mean.

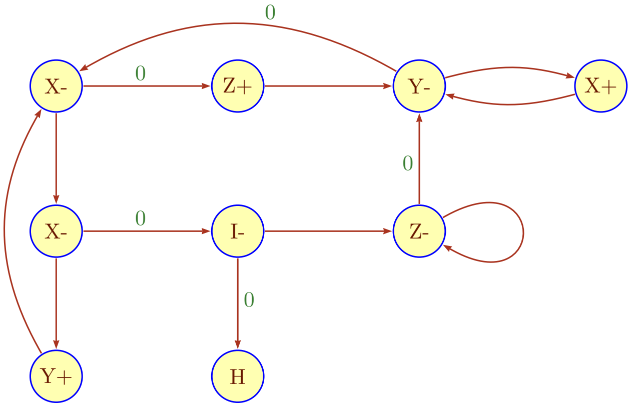


Figure 8.9: A register machine for a decoding function.

As an immediate corollary we will see that certain questions about the behavior of these machines are undecidable: there is no machine that answers these questions correctly in all cases. More precisely, we will construct a single register machine $\mathcal{U}$ that is capable of simulating every other register machine. There is a minor technical issue that we need to address: $\mathcal{U}$ is required to have a fixed number of registers, whereas the simulated machines can have arbitrarily many registers. The solution to this problem is coding: we use a sequence number to keep track of the register contents of the simulated machine. As a consequence, our simulation will be far from lockstep, but that is fine. Sequence numbers are also critical in storing and manipulating a representation of the program to be simulated. Suppose $P = I_0, I_1, \ldots, I_{n-1}$ is a RM-program. Here is one fairly natural way to encode individual instructions as integers:

instruction	code
halt	$\langle 0 \rangle$
increment (r, k)	$\langle r, k \rangle$
decrement (r, k, l)	$\langle r, k, l \rangle$

The code for P is the sequence number obtained from the codes of the instructions. For example, consider the simplified addition program:

```
// addition   R0 + R1 --> R1
  0:   dec 0  1  2
  1:   inc 1  0
  2:   halt
```

In the parity coding scheme from section 8.1.3, this program has code number

$$\langle \langle 0, 1, 2 \rangle, \langle 1, 0 \rangle, \langle 0 \rangle \rangle = 88098369175552.$$

Our universal register machine $\mathcal{U}$ has two special registers named C (for code) and R (for registers): C contains the code number of the program P to be simulated and R contains the code number of the sequence of register contents for P. Moreover, register p serves as the program counter, and register I will store individual instructions of P.

To keep the construction reasonably simple we will only consider programs that require a single input, to be stored in register R_0. The universal machine uses register x as an input register for the argument to be forwarded to P. It is straightforward to generalize this approach to the simulation of programs with several input arguments. Alternatively, we can think of the sole input x as a sequence number and obtain multiple arguments by decoding this sequence number. Of course, this requires are pre-processing phase in P; see the exercises for more details. The program counter p is initialized to 0 and updated every time an instruction of P is simulated. Informally, one cycle in the execution of $\mathcal{U}$ has the following form.

1. Extract the pth instruction from C and store it in register I. If p turns out to be out-of-range, halt.
2. Decode I to determine the type of the instruction. If the instruction is "halt," do so halt. Otherwise extract the register number r and the goto label k for increment, or labels k and l in a decrement operation.
3. Extract the content of the rth register of the simulated machine from real register R, halting if r is out-of-range. If the instruction in I is increment, change the contents of the simulated R_r correspondingly; set the value of p to k. If the instruction is decrement first check the value of R_r: if it is 0, set p to l; otherwise decrement the value and set p to k. Continue at step one.

For step (1) note that the extraction of the next command has to be non-destructive. To this end we use the push and pop programs from above: we think of C as a stack and pop the p first elements, pushing them onto an auxiliary stack C'. The last element, i.e., the code for instruction I_p, is also placed into register I. Then all elements are moved from C' back to C. The same technique is used for the extraction of the contents of a simulated register in step (3). The decoding in steps (2) and (3) can be handled by using the RM program from above that determines the number of 1's in the binary expansion of a given instruction code c and thus computes $\mathsf{ds}(c)$.

It is clear from the sample RM-programs above that a full implementation of this method would produce a mostly incomprehensible program. It is preferable to give a high level description of the machine that is fairly easy to understand using macros, pseudo instructions that are not directly available in our language, but that could be implemented by a real RM-program in a fairly straightforward manner. Of course, there is a trade-off: the closer the macros stay to a real RM-program the more complicated and opaque the universal machine becomes. Below we present what we feel is a reasonable compromise.

- `copy r s k`
  Non-destructively copy the contents of R_r to R_s and goto instruction k.
- `zero r k l`
  Test if the content of R_r is 0; if so, goto instruction k, otherwise goto instruction l.
- `pop r s k`
  Interpret the content of R_r as a sequence number $a = \langle b, c \rangle$; place b into R_s and c into R_r, continue at instruction k. If $[R_r] = 0$ both registers will be set to 0.
- `read r t s k`
  Interpret the content of R_r as a sequence number and place the $[R_t]$th component into R_s, continue at instruction k. Halt if $[R_t]$ is out of bounds.
- `write r t s k`
  Interpret the content of R_r as a sequence number and replace the $[R_t]$th component by $[R_s]$, continue at instruction k. Halt if $[R_t]$ is out of bounds.

Given these macros we can describe a universal register machine as follows. The comments in the right column provide an informal explanation for the effect of the macro on the left.

```
// universal pseudo register machine
  0:   copy   C  R  1               // R = C
  1:   write  R  p  x   2           // R[0] = x
```

```
 2:  read   C  p  I   3        // I = C[p]
 3:  pop    I  r  4            // r = pop(I)
 4:  zero   I 14  5            // if( I == 0 ) goto 13
 5:  pop    I  p  6            // p = pop(I)
 6:  read   R  r  x   7        // x = R[r]
 7:  zero   I  8  9            // if( I != 0 ) goto 9
 8:  inc    x 12               // x++; goto 12
 9:  zero   x 10 11            // if( x != 0 ) goto 11
10:  pop    I  p  2            // p = pop(I)
11:  dec    x 12 12            // x--
12:  write  R  r  x   2        // R[r] = x; goto 2
13:  halt
```

Note that this machine abuses the sequence number C as the initial values for the registers of the simulated machine, overwriting only R_0 by x. We could zero out the other components of R at the cost of either a more complicated and somewhat ad hoc macro, or by making $\mathcal{U}$ longer. At any rate, we have the following result.

Theorem 8.4.1 (Universal Register Machines) *There is a universal register machine $\mathcal{U}$ that, given any register machine program P and inputs $x_1, \ldots, x_n$ for P, simulates P on input $\langle P \rangle$ and $\langle x_1, \ldots, x_n \rangle$.*

In particular, P halts on input $\boldsymbol{x}$ if, and only if, $\mathcal{U}$ halts on $\langle P \rangle$ and $\langle \boldsymbol{x} \rangle$. Moreover, in the case of convergence, the result $P(\boldsymbol{x})$ will appear in a special register of $\mathcal{U}$. As to the correctness of the construction, we have to admit that a detailed proof seems rather daunting, albeit rather straightforward as far as the proof strategy is concerned. We take great solace from the fact that an actual implementation appears to function well. Remember Thurston's comment from section ?

One immediate consequence of the theorem is that certain questions relating to the behavior of register machines are algorithmically undecidable. Most notably, we have the following result.

Theorem 8.4.2 (Halting Problem)
It is undecidable whether a given register machine halts on a given input.

Proof. For simplicity, consider only the case of machines with a single input: given $e = \langle P \rangle$, we will show that there is no register machine that determines whether P halts after finitely many steps on input e.

Assume otherwise, and call the corresponding RM-machine H. More precisely, H returns 1 if P on e halts, and 0 otherwise; in either case, H halts. But then we can construct a new RM-machine Q that, on input e, does the following:

- Use H to check whether P on e halts,
- if so, go into an infinite loop,
- otherwise, halt and output 0.

Now let $q = \langle Q \rangle$ be the index for this program Q. By the very definition of Q, if Q halts on q, then it loops; and if it fails to halt, it halts. Contradiction. □

It might seem that this result is a bit too navel-gazing: we are interested in the limitations of machines, and, in order to exhibit one such limitation, we focus on the behavior of the machines themselves. However, it has since turned out that there are many problems in mathematics and computer science that are similarly undecidable, though many of them seem to have no direct connection to register machines or computation in general. Perhaps the most famous example is a theorem by Matiyasevic, building on work by Robinson, Davis and Putnam, who showed in 1970 that it is undecidable whether a Diophantine equation of the form $p(x_1, \ldots, x_n) = 0$ has a solution over the integers (here p is a multivariate polynomial with integer coefficients), see section 9.3. Note that the problem is "almost" solvable, though: we can systematically enumerate all possible values of the variables, and check if any particular combination evaluates to 0. Alas, this is not an algorithm: there is no way to determine when we can stop this process and conclude that no solutions whatsoever exist.

Historically, our approach is a bit out of order: Turing conceived of his eponymous machines described in the next section in the 1930s, and established the unsolvability of the Halting problem using these machines. Register machines appeared a quarter century later.

8.4.4 Turing Machines

Our next model of computation may seem to be a step back from register machines in the sense that Turing machines are even further removed from modern digital computers than register machines. However, their primary purpose was to capture any computation that could conceivable be carried out at all, and in that they succeed magnificently. At a time when a truly compelling definition of computation was not yet available, Turing's great idea was to analyze the behavior of a human computor: after all, humans, and in particular mathematicians, are certainly capable carrying out computations, and it is unclear whether any operation that cannot, in principle, be carried out by a human computor could be considered a computation. Of course, we have to disregard obvious limitations such as the limited lifespan of a human being and indeed the universe itself, never mind the lack of unbounded supplies of scratch paper. Turing accomplishes this by carefully observing and analyzing the way a human computor operates and then abstracting the key operations so they can be performed by a simple "mechanical" device.

The first important observation is that a human being typically uses paper and pencil to perform calculations (well, at the time). Writing down an expression amounts to a bookkeeping operation, we keep track of intermediate results and use them later on. Mathematicians in particular often use two-dimensional notation, but there is no loss in flattening everything out into a one-dimensional string. Hence, we can think of the paper plus the writings on it as a plain sequence of letters, a word in the sense of language theory. Note that the letters must be chosen from a finite supply: if there were infinitely many letters, it would take the computor an infinite amount of time to learn them all–not to mention obvious physical limitations such as the ability to remember all these symbols or to render them in a limited amount of physical space. In the context of Turing machines it is customary to think of this word of intermediate results as symbols being written on a tape: the tape is subdivided into cells, each cell being either blank or containing a single letter.

How does the computor perform a single step in the computation? Since humans are only capable of focusing their mind on a rather small number of objects at any particular time, we may as well assume that our computor only perceives a single cell at any particular time. Thus, the computor is aware of the contents of that one cell, but not of the neighboring cells. Moreover, the computor understands the method of the computation and knows how to proceed from input through intermediate results to the final output. The exact details of the computation method are not relevant, but we can safely assume that, at any point, the computor is in one particular internal mind state. Again for physical reasons, there are only finitely many possible states.

A single step in the computation can then be broken down into three separate sub-steps.

First, the symbol in the current tape cell may be overwritten. Second, attention may shift to one cell to the left or right. Third, the internal state may change. It is important that all these activities depend only on the symbol in the currently scanned cell and on the current internal state. The previous history of the computation, for example, is not relevant. Since there are only finitely many symbols and finitely many internal states, all possible steps can be specified in a finite lookup table. It is straightforward to formalize this description.

Definition 8.4.5 *A Turing machine is a structure* $M = \langle Q, \Sigma, \delta; q_{\text{init}}, q_{\text{halt}} \rangle$ *where*

- Q *is the state set, a finite set,*
- Σ *is the tape alphabet, a finite set including a special blank symbol* ␣*,*
- $\delta : Q \times \Sigma \nrightarrow Q \times \Sigma \times \{-1, 0, 1\}$ *is the transition function,*
- $q_{\text{init}} \in Q$ *is a special initial state,* $q_{\text{halt}} \in Q$ *is a special terminal or halting state.*

A single step by a Turing machine is determined by an instruction of the form $\delta(p, a) = (q, b, \Delta)$ where p is the current state, a the currently scanned tape symbol, q the next state, b the symbol with which a will be overwritten, and $\Delta \in \{-1, 0, 1\}$ the displacement that determines how attention shifts to the left, right or stays at the same cell. We allow the transition function δ to be partial, simply because that makes it somewhat easier to construct concrete examples of Turing machines.

It may be fairly clear how a Turing machine computes, but let us make sure to have a complete and formal definition. A tape inscription naturally is a map $\tau : \mathbb{Z} \to \Sigma$ such that all but finitely many values of τ are the special blank symbol ␣. Because of the finite support condition, it is convenient to write tape inscriptions as finite words over Σ with the understanding that the unspecified cells all carry a blank symbol. This presentation omits the actual coordinates in $\mathbb{Z}$, but this is a preferable representation: the Turing machine itself has no access to them in the first place. In fact, since we also need to keep track of the scan/read/write head of the machine, it is convenient to record the current tape inscription plus head position in a coordinate-free manner by two words $a = a_m \ldots a_2 a_1$ and $b = b_1 b_2 \ldots b_n$ over the tape alphabet where the non-blank part of the tape inscription is ab, and the head is considered to be placed at the first symbol of b. Note that a is written backwards, a minor technical convenience that will make it easier to simulate Turing machines by other devices such as register machines, see section 8.4.2. We may safely assume that the state set and the tape alphabet of a Turing machine are disjoint.

Hence we can think of a configuration of a Turing machine as a word

$$a_m a_{m-1} \ldots a_2 a_1 \, p \, b_1 b_2 \ldots b_{n-1} b_n$$

where $p \in Q$. We may safely assume that both a and b are non-empty words, otherwise we can set $a_1 =$ ␣ or $b_1 =$ ␣, as required. The space of all configurations of M is then $\mathcal{C} = \Sigma^+ \, Q \, \Sigma^+$, a simple regular language over the alphabet $Q \cup \Sigma$. To define the next-step relation on $C \to C'$, suppose $\delta(p, b_1) = (q, c, \Delta)$ and distinguish three cases according to the displacement Δ:

C'	Δ
$a_m \ldots \, q \, a_1 c \, b_2 \ldots b_n$	-1
$a_m \ldots a_1 \, q \, c \, b_2 \ldots b_n$	0
$a_m \ldots a_1 c \, q \, b_2 \ldots b_n$	$+1$

Ignoring the head position, the next tape inscription is the same in all three cases, to wit $a_m \ldots a_1 c \, b_2 \ldots b_n$.

It remains to develop suitable input/output conventions. Compared to the register machines from the last section 8.4.2, our situation is now the exact opposite: a Turing machine naturally acts on strings, a register machine naturally acts on integers. To make a Turing machine act on integers we have to encode them as strings, to make a register machine act on strings we have to encode them as integers. Let us handle Turing machine computations on strings first and deal with arithmetical functions later.

Write $\Gamma = \Sigma \cup \{\textvisiblespace\}$ for the tape alphabet of the machine, where Σ must contain at least one symbol. Given a string $x \in \Sigma^\star$ and a state $p \in Q$, define the configuration

$$C_x^p = p\textvisiblespace x_1 x_2 \ldots x_n$$

So the tape is all blank except for the letters in x, the machine is in state p and the head is positioned at the blank immediately to the left of x (note that this convention allows for x to be the empty string). The configuration $C_x^{q_{\text{init}}}$ will be referred to as the initial configuration for x, and will be starting point in $\mathfrak{C}_M$ for the computation of M on x. Similarly we declare a terminal or halting configuration to be of the form $C_z^{q_{\text{halt}}}$. Hence, our input/output maps take the form

$$\begin{aligned} \mathsf{inp}(x) &= C_x^{q_{\text{init}}} \\ \mathsf{outp}(C_z^{q_{\text{halt}}}) &= z \end{aligned}$$

Note that our output convention, while perfectly reasonable, is a bit restrictive: we require the tape head to move to the first blank cell to the left of the output, and the output has to appear in one block immediately to the right of the tape head. In particular all scratch space must have been erased before the machine halts. One of the reasons these conventions are useful is that they make it fairly easy to compose Turing machines, to apply a Turing machine $\mathcal{M}_2$ to the output generated by another machine $\mathcal{M}_1$. At any rate, we can now formally define the clone of functions that can be computed by Turing machines.

Definition 8.4.6 (Turing Machine Computability)
Turing machine M computes A partial function $f : \Sigma^\star \nrightarrow \Sigma^\star$ is Turing machine computable or TM-computable if there exists a Turing machine M such that M has a terminating computation on $\mathsf{inp}(x)$ if, and only if, $f(\boldsymbol{x})$ is defined; in this case $\mathsf{outp}(C) = f(x)$ where C is the terminal configuration reached from $\mathsf{inp}(x)$.

In this setting, failure to produce output can be caused in two different ways. First, we may have $C_x^{q_{\text{init}}} \vdash_{M} C$ where C has no next configuration but fails to be terminal. Second, the computation of M on $C_x^{q_{\text{init}}}$ may be infinite and never reach a terminal configuration.

To lift our definition of Turing computability to arithmetical functions $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ we need to resort to coding. More precisely, we need to determine an encoding map $E : \mathbb{N}^k \to \Sigma^\star$ and a decoding map $D : \Sigma^\star \to \mathbb{N}$ giving rise to input/output maps

$$\begin{aligned} \mathsf{inp}(\boldsymbol{x}) &= C_{E(\boldsymbol{x})}^{q_{\text{init}}} \\ \mathsf{outp}(C_z^{q_{\text{halt}}}) &= D(z) \end{aligned}$$

For example, we could write individual numbers in binary; vectors can be handled by using a special separator symbol. We will return to this topic in our treatment of Kolmogorov-Chaitin complexity and of complexity theory, where coding details are of some importance. In the realm of general computability any reasonable convention will produce the same clone of computable functions and we will usually avoid a detailed discussion.

Turing Machine Examples

Example 8.4.1 The following Turing machine has tape alphabet $\{\llcorner\!\lrcorner, a\}$ and states $Q = \{0, 1, 2, 3\}$, where $q_{\text{init}} = 0$ and $q_{\text{halt}} = 3$. It appends a single letter a to the end of a given block of as, returns the head to the left and halts. The transition function δ is given be the following table:

p	s	$\delta(p, s)$		
0	$\llcorner\!\lrcorner$	1	$\llcorner\!\lrcorner$	+1
1	$\llcorner\!\lrcorner$	2	a	0
1	a	1	a	+1
2	$\llcorner\!\lrcorner$	3	$\llcorner\!\lrcorner$	0
2	a	2	a	−1

Consider the configuration $q_{\text{init}}\llcorner\!\lrcorner aa \ldots aa$ where there are n many a's on the tape. It changes to $1aa \ldots aa$ in one step. The read/write head then moves to the right $a \ldots a1a \ldots a$ until we reach $a \ldots a1\llcorner\!\lrcorner$, at which point the last blank is changed into an a: $a \ldots a2a$. The head then moves back to the left until $3\llcorner\!\lrcorner a \ldots aa$ and the machine halts: there are no transitions defined out of state 3. We now have $n + 1$ letters a on the tape. It is a good exercise to repeat the construction when only displacements $\{-1, 1\}$ are allowed. As the example shows, it is a slightly easier to design a concrete Turing machine when more displacements are allowed.

The combinatorial description of the behavior of the Turing machine is quite tedious, a graphical representation of a computation is often easier to understand, see figure 8.10. If we code natural numbers in "fat unary" (n is written as a^{n+1}), the machine computes the successor function. A minor modification of the machine produces an adding machine, as shown in figure 8.11. We have adopted the convention that the two arguments are separated by a blank symbol.

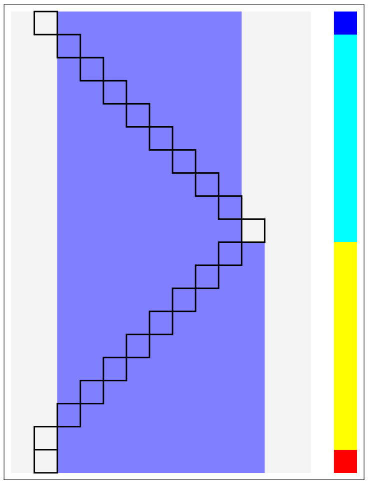


Figure 8.10: A Turing machine computing the successor function in unary. Time flows from top to bottom, and the separate column on the right indicates state.

Example 8.4.2 The following Turing machine doubles the length of a block of 1's and halts. The tape alphabet is $s = \{\llcorner\!\lrcorner, 1, 2, 3\}$ and there are 6 states. The transition function

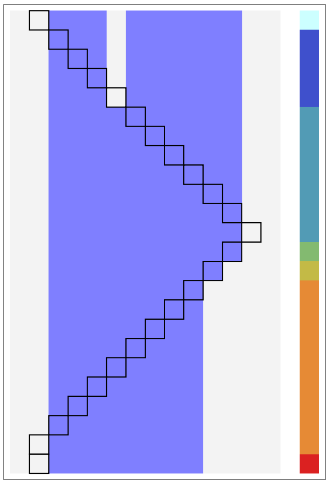


Figure 8.11: A Turing machine that adds two numbers in fat unary.

δ is given be the following table:

p	s	$\delta(p,s)$			p	s	$\delta(p,s)$		
0	␣	0	␣	1	2	3	2	3	−1
0	1	1	2	1	3	1	3	1	−1
0	3	4	3	1	3	3	3	3	−1
1	␣	2	3	−1	4	␣	5	␣	−1
1	1	1	1	1	4	3	4	3	1
1	3	1	3	1	5	␣	5	␣	0
2	1	2	1	−1	5	2	5	1	−1
2	2	0	2	1	5	3	5	1	−1

A sample computation of the doubling machine is shown in the following picture. Note the zig-zag pattern of the head movement. It is a healthy exercise to determine the running time of this Turing machine on all inputs of the form 1^n.

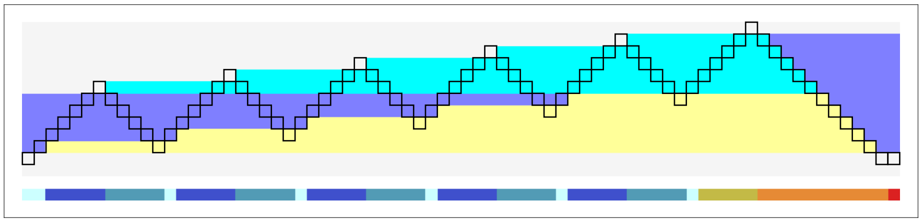


Figure 8.12: A Turing machine doubling its input. Time flows from left to right.

Example 8.4.3 A 10-state Turing machine that checks whether a given word x over the alphabet $\{a, b\}$ is a palindrome.

p	s	$\delta(p, s)$			p	s	$\delta(p, s)$		
0	␣	1	␣	+1	4	a	7	␣	−1
1	␣	8	␣	0	4	b	6	b	0
1	a	2	␣	+1	5	␣	6	␣	0
1	b	3	␣	+1	5	a	6	a	0
2	␣	4	␣	−1	5	b	7	␣	−1
2	s	2	s	+1	7	␣	8	␣	0
3	␣	5	␣	−1	7	s	9	s	−1
3	s	3	s	+1	9	␣	1	␣	+1
4	␣	6	␣	0	9	s	9	s	−1

Intuitively, the machine operates by zig-zagging back and forth over the given string. In each round, it matches the first symbol in the given string against the last. If there is a match, both are erased the process continues on the remainder of the string. If there is a mismatch, the computation fails. Strictly speaking, the machine ought to write a yes/no response on the tape before halting, but that would cost a few extra states, see the exercises. This method is clearly quadratic in the size of the input. More interestingly, one can show that this is optimal for all one-tape Turing machines.

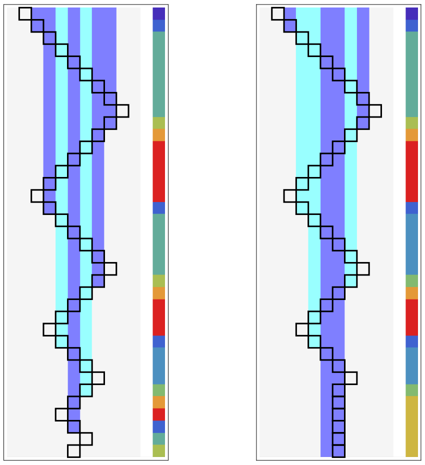


Figure 8.13: A Turing machine that checks for palindromes. The first computation succeeds, the second one fails.

Example 8.4.4 As a last example, recall the infamous Collatz function, defined on the positive integers by

$$C(x) = \begin{cases} x/2 & \text{if } x \text{ even,} \\ 3x + 1 & \text{otherwise.} \end{cases}$$

The Collatz conjecture states that the orbit of any positive integer n under C contains 1. The conjecture is infuriatingly simple and supported by a mountain of evidence, but apparently exceedingly difficult to settle. We will construct a Turing machine that simulates the Collatz function in the following sense: we adjust our input map to write a given positive integer n in reverse binary, no trailing 0s, and with the head positioned at the first digit; the machine computes all the values $C^t(n)$, without halting. Presumably, it always winds up in the loop $1 \rightsquigarrow 4 \rightsquigarrow 2 \rightsquigarrow 1$, though the tape inscription keeps shifting to the right.

p	s	$\delta(p,s)$			p	s	$\delta(p,s)$		
1	0	1	␣	1	3	␣	4	1	1
1	1	2	0	1	4	0	4	0	1
2	0	3	0	1	4	1	3	1	1
2	1	2	1	1	4	␣	5	␣	−1
2	␣	3	0	1	5	0	5	0	−1
3	0	4	1	1	5	1	5	1	−1
3	1	2	0	1	5	␣	1	␣	1

Figure 8.14 shows the simulation of C on input $n = 9$. The transient part of the orbit has length 23: $9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$. The simulation on our Turing machine takes 101 steps to reach 1.

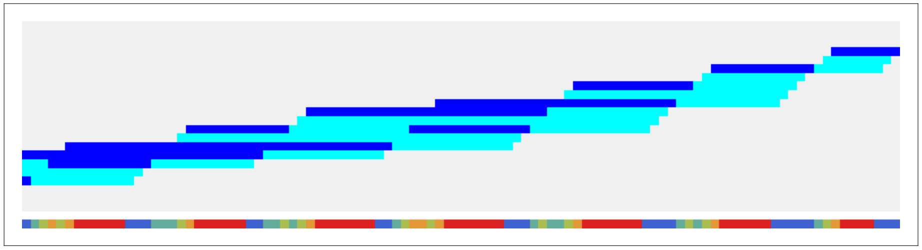


Figure 8.14: A Turing machine simulation of the orbit of $n = 9$ under the Collatz function.

The last two examples show that we need to take our conventions regarding coding, input and output with a grain of salt. It may make perfect sense to have a machine perform a divergent computation, e.g., we could envision a machine that runs forever and writes all the prime numbers on the tape. In complexity theory we will also have to modify the "hardware" of our Turing machines: there may be multiple tapes, they may be one-way infinite, read-only, write-only and so on and so forth. We adopt whatever convention is useful in a particular domain. For general computability we will stay with a single, two-way infinite read/write tape, though.

8.4.5 Turing versus Register Machines

On the face of it, Turing machine and register machines operate on different domains, to wit, strings versus natural numbers. Since there are computationally simple ways to convert between the two domains, it still makes sense to try to compare their relative computational power. For simplicity, consider only arithmetical functions $f : \mathbb{N}^k \nrightarrow \mathbb{N}$.

Theorem 8.4.3 *An arithmetical function is register machine computable if, and only if, it is Turing machine computable.*

A detailed proof is quite tedious, but not particularly challenging. Here is a sketch of the argument.

First assume we have a number-theoretic function $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ computed by some register machine P. To translate f to the realm of string functions, we code input and output in unary—after all, the register machine can only perform successor and predecessor operations at the basic level. More precisely, suppose that we have digits $0, 1$ in Γ. We code an n-tuple of natural numbers $x_1, \ldots, x_n$ as

$$X = 1^{x_1+1}01^{x_2+1}0\ldots01^{x_n+1}$$

Our simulating Turing machine $\mathcal{M}$ will maintain a block of tape of the form

$$\#1^{e_0}\#1^{e_1}\#\ldots\#1^{e_{w-1}}\#$$

where w is the number of registers in P and # is a separator symbol. It copies the given input into the register block and then simulates the operation of the register machine by adding or removing 1s from the various segments of the register block (this requires moving things around quite a bit, but it is not particularly difficult). The program of P can be stored directly in the transition function of $\mathcal{M}$. If the computation of P terminates, the Turing machine erases the whole tape except for the part that represents the output register, places the head in the proper position and halts. If P fails to terminate, the Turing machine similarly fails to halt.

For the opposite direction we have to lean heavily on the fact that register machines can handle all the coding machinery we developed in section 8.1.3. In particular, a register machine can represent a string and hence a configuration of a Turing machine as a sequence number. It can also perform all the manipulations required in the definition of the next-step relation for Turing machines. The transition function of the Turing machine can be hardwired in the program of the register machine. Moreover, we can set up a loop that repeats these steps until a terminal configuration is reached, or runs forever, depending on the behavior of the Turing machine.

Note that more is true than claimed in the theorem: there is a primitive recursive function Φ that, given a register machine with index e, determines the index $\Phi(e)$ of a corresponding Turing machine. Similarly there is a conversion function for the opposite direction. This equivalence indicates that our definitions are fairly robust, the same notion of computability arises in fairly different settings. As was shown by Turing in his seminal 1936 paper, there are universal Turing machines analogous to our universal register machines. Similarly, the Halting problem for Turing machines is unsolvable.

One might assume that the ability of a Turing machine to write freely on its tape is critical for the computational power of these machines; after all, the motivating idea was to formalize the behavior of a human computor. It comes as a bit of a surprise that even a more restricted class of machines due to Hao Wang in 1954 is already capable of modeling all partial computable functions. The tape alphabet is **2**. Informally, the instructions in a Wang machine are restricted to `move left`, `move right`, `print` 1, `goto` k and `halt`. A program is a numbered list of such instructions, and the semantics of the goto statement is: if the current symbol is 1, goto line k, otherwise continue with the next instruction. So these machines are non-erasing: a 0 can turn into a 1, but not the other way around. It takes a bit of effort to adjust the input/output conventions, e.g., we cannot clean up the tape before the computation terminates. Still, with significant effort one can show that any computable function can already be computed by a Wang machine.

8.4.6 Rado's Problem

In 1962, almost 30 years after Turing's seminal work, Tibor Rado described a computational problem that is directly related to Halting, but appears more tangible on an intuitive level. Consider only Turing machines on the minimal tape alphabet $\Sigma = \mathbf{2}$. Since the tape alphabet is fixed, the computational power of a machine depends on the number of its states. This naturally leads to the following question:

> What is the largest number of 1's any such machine can write on an initially blank tape, and then halt?

Why should this be difficult? An obvious line of attack would be to generate a list of all n-state Turing machines and then run all of them in parallel. After some time, some of the machines will halt and we can count the number of 1s left on the tape to determine the current champion. Of course, other machines will still be active, so we have to continue our simulation. The problem is that in order to terminate the experiment at some point and declare a winner, we would need to know that all the remaining active machines will never halt or at least not halt after having written more 1s than the current champion. Alas, this clashes directly with the unsolvability of the Halting problem.

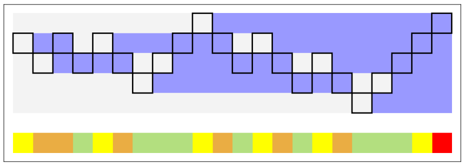
Rado's original question is actually slightly arbitrary, here are two versions more firmly rooted in computability theory.

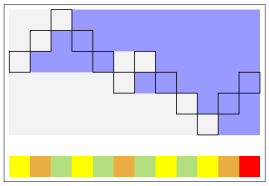
Busy Beaver Time Complexity
What is the largest number of moves a halting n-state machine can make?

Busy Beaver Space Complexity
What is the largest number of tape cells a halting n-state machine can use?

We write $\mathsf{BB}_{\mathrm{H}}(n)$ for the largest number of steps taken by any halting n-state machine. Similarly, $\mathsf{BB}_{\mathrm{W}}(n)$ denotes the largest number of 1's written by any halting n-state machine. All these kinds of functions are called Busy Beaver functions. Clearly, $\mathsf{BB}_{\mathrm{H}}(n) \geq \mathsf{BB}_{\mathrm{W}}(n)$, but the former has the advantage of relating more directly to the Halting problem. It is the standard convention to disregard the halting state in the context of busy beaver considerations, so n-state means n active states plus one halting state. Also, the head is required to move at each step.

One attractive feature of Rado's problem is that, general unsolvability notwithstanding, one can attempt to determine a few concrete function values for small n. For example, we have $\mathsf{BB}_{\mathrm{H}}(1) = \mathsf{BB}_{\mathrm{W}}(1) = 1$. Even for $n = 2$ it takes a bit of fumbling to show that $\mathsf{BB}_{\mathrm{W}}(2) = 4$ and $\mathsf{BB}_{\mathrm{H}}(2) = 6$, with the same champion. For $n = 3$ the two champions are





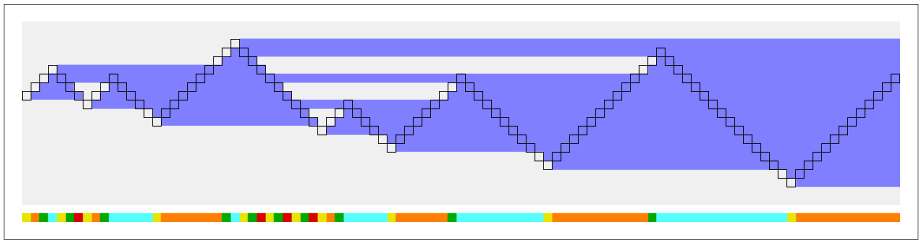
Exact values are still known for $n = 4$, but beyond that we have to make do with lower bounds. Currently, the state-of-the-art is as follows:

n	$\mathsf{BB}_\mathrm{H}(n)$	$\mathsf{BB}_\mathrm{W}(n)$
1	1	1
2	6	4
3	21	6
4	107	13
5	$\geq 47\,176\,870$	≥ 4098
6	$> 7.4 \times 10^{36\,534}$	$> 3.5 \times 10^{18\,267}$

Note, though, that it is not exactly clear how reliable some of these results are, see [**?**]. The current 5-state champion is due to Marxen and Buntrock, and behaves in a rather surprising manner. In the table below, entry (q, b, X) in position (p, a) means that $\delta(p, a) = (q, b, X)$.

	0	1
1	(2,1,R)	(3,1,L)
2	(3,1,R)	(2,1,R)
3	(4,1,R)	(5,0,L)
4	(1,1,L)	(4,1,L)
5	halt	(1,0,L)

Here is a plot of the first 100 steps of this machine on empty tape.



Looking at a run of the Marxen-Buntrock machine for a few hundred or even a few thousand steps, see figure 8.15, one invariably becomes convinced that the machine will never halt: the machine zig-zags back and forth, sometimes building solid blocks of 1's, sometimes a striped pattern. Whatever the details, the machine seems to be in a "loop." Bear in mind: there are only 5 states, there is no obvious method to code an instruction such as "do some zig-zag move 1 million times, then stop." And yet, this machine stops after an unbelievable 47,176,870 moves and leaves 4098 symbols 1 on the tape. Visual evidence can sometimes be quite misleading.

8.4.7 Exercises

Exercise 8.4.1 Write a register machine program that computes factorials. Write a register machine program that computes binomial coefficients.

Exercise 8.4.2 Write a register machine program that implements the bit function: on input i and x, return the ith bit in the binary expansion of x.

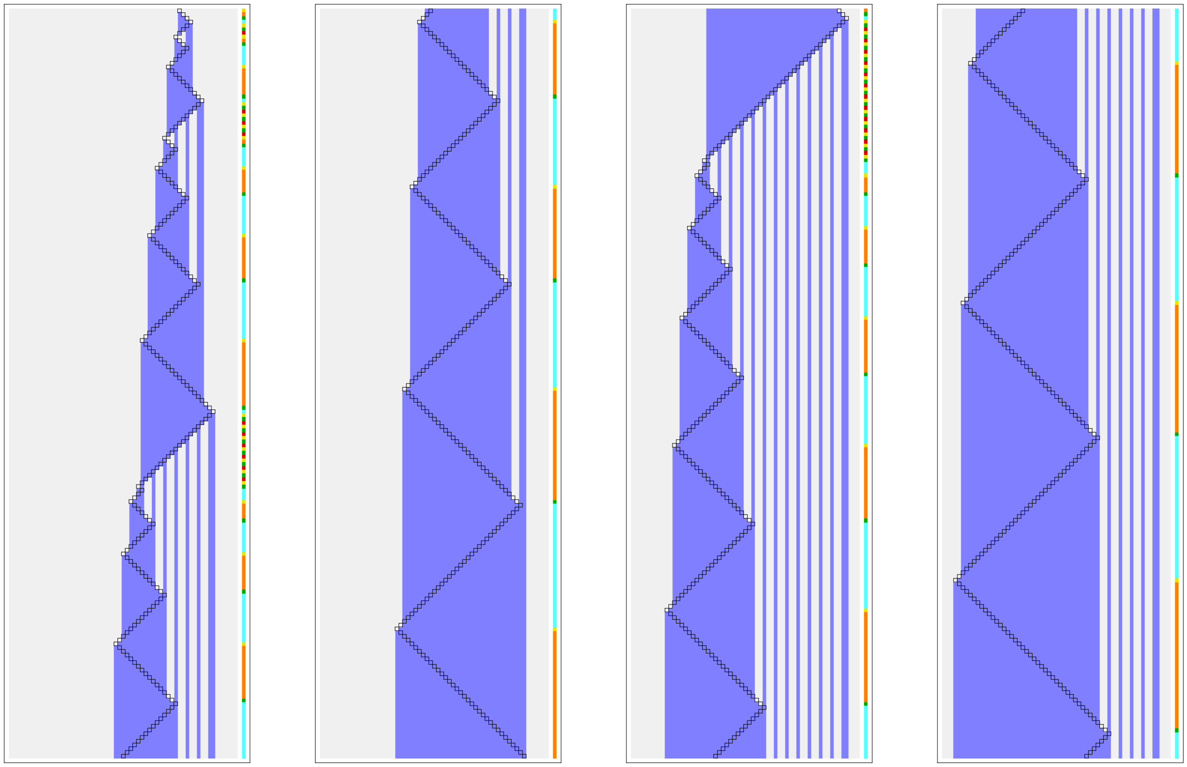


Figure 8.15: The first 800 steps of the Marxen-Buntrock machine, each block corresponding to 200 steps. Note the apparent repetitiveness, it seems that the machine is in a loop.

Exercise 8.4.3 Write a register machine program that implements the prepend function, using our coding function from above.

Exercise 8.4.4 Determine the running time of the multiplication register machine from above.

Exercise 8.4.5 Show how loop-computable functions are closed under definition by cases.

Exercise 8.4.6 Show that the following instruction set for register machines will produce the same notion of register machine computable function.

`zero r k`	set register R_r to zero, goto next instruction.
`inc r k`	increment register R_r, goto next instruction.
`comp r s k l`	if $[R_r] = [R_s]$, goto instruction k, otherwise goto l.
`halt`	well ...

Exercise 8.4.7 Prove that the register machines from figures 8.6, 8.8, 8.9 are correct.

Exercise 8.4.8 Write a simulator for register machine programs. Improve your simulator by analyzing loops in the flowgraph of the register machine and collapsing them into a single super-step if possible.

Exercise 8.4.9 Show how to expand the macros in universal register machine from section 8.4.3 into real register machines. Use these machines to turn the universal pseudo register machine into are real universal register machine. Try to minimize the size of your universal machine.

Exercise 8.4.10 Give a detailed proof that random access machines are equivalent to register machines.

Exercise 8.4.11 Establish a hierarchy of the Busy Beaver functions representing the four variants of the problem.

Exercise 8.4.12 Derive the transition table of the 3-state busy beaver from the picture of its computation. Show that this machine is indeed the champion.

Exercise 8.4.13 Formulate a plan to find the busy beaver for $n = 4$. What are the main obstructions to generalizing your approach to $n = 5$?

Exercise 8.4.14 A random access machine (RAM) is defined similarly to a register machine, but it has an additional instruction `write a b c` with the effect that the contents of register R_a are written to register R_b where $b = [R_c]$; control then moves to the next instruction. Similarly, there is a `read a b c` instruction.

Explain the semantics of RAMs in more detail and show that they generate the same clone of computable functions as ordinary register machines.

8.5 Alternative Models

8.5.1 Herbrand-Gödel Recursion

Programs and machines are popular in computer science, but in mathematics one generally prefers other descriptions of functions. Our next goal is to provide a definition of computability that stays close to standard mathematical practice and uses only equations to describe certain arithmetic functions. We have already seen the definition of addition in terms of two equations using the successor function, written informally as

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

One general issue in using equational definitions is that one has to be concerned about the existence of solutions as well as uniqueness, should a solution indeed exist. In our case, we also have to make sure that the solution is computable.

For the purpose of this discussion, it is best to be somewhat pedantic about equations as purely syntactical objects. We will allow the use of a constant $\underline{0}$ and the unary successor function S. Moreover, there will be numerical variables $x_1, \ldots, x_k$ and finitely many function symbols $f_1, \ldots, f_\ell$ of various arities. For legibility, we often write x^+ instead of $S(x)$. We can represent natural numbers via numerals: $\underline{0}$ is a numeral, and, for any numeral $\underline{n}$, the successor $\underline{n}^+$ is also a numeral. To keep notation manageable, we will write, for example, $\underline{2}$ instead of $\underline{0}^{++}$. Note that this is just syntactic sugar that improves readability, it adds nothing to the expressiveness of our equations. An equation is then a formal expression $s = t$, where both s and t are terms in our language. One important constraint is that s is restricted to be of the form

$$f(s_1, \ldots, s_r)$$

where there terms s_i contain no function symbols except perhaps S. In particular, if all the s_i are numerals, we can try to evaluate s as in the example for addition. We call f the principal function symbol of this equation.

Our addition system from above could now be written more precisely as

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

We can spot-check that this version really defines addition by deriving a chain of identities, involving only numerals and the principal function f:

$$f(\underline{3},\underline{2}) = f(\underline{2},\underline{1}^+) = f(\underline{3},\underline{1})^+ = f(\underline{3},\underline{0}^+)^+ = f(\underline{3},\underline{0})^{++} = \underline{3}^{++} = \underline{4}^+ = \underline{5}$$

What exactly does it mean that a system of equations E defines a function? We need to find actual partial functions $F_1, \ldots, F_\ell$ of appropriate arity so that, replacing f_i by F_i, $\underline{0}$ by $0 \in \mathbb{N}$ and S by the successor function, we obtain valid identities over the natural numbers. Moreover, we want the solution F_1 for f_1 to be unique, in which case we say that E defines the function F_1 (and, possibly, several others).

Here is another example, this one involving two function symbols:

$$\begin{aligned} f_1(\underline{0}) &= \underline{1} & \qquad f_2(\underline{0}) &= \underline{0} \\ f_1(x^+) &= f_2(x) & \qquad f_2(x^+) &= f_1(x) \end{aligned}$$

It is not hard to see that this system defines the characteristic functions of the even and odd numbers, respectively. As one might suspect, not every system of equations works properly, here are two standard counterexamples.

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

The problem with the first system is that it admits too many solutions, any function F such that $F(x,y) = H(x+y)$, $H(0) = 0$, will do. On the other hand,

$$\begin{aligned} g(x,\underline{0}) &= x^+ \\ g(x,y^+) &= g(x^+,y) \\ g(x^+,y^+) &= g(x,g(x,y)) \end{aligned}$$

has no solution, the equations are contradictory. Here is a much more problematic case, a system that has a unique solution, but this solution cannot be discovered by the kind of equational reasoning from above. For simplicity, let us pretend that multiplication by 2 is available, we could easily include a corresponding definition in the system.

$$f(x) = 2 * f(x^+)$$

A solution F would need to satisfy infinitely many identities

$$F(0) = 2\,F(1) = 4\,F(2) = \ldots = 2^k F(3) = \ldots$$

It follows that 2^k divides $F(0)$ for all k, hence $F(0) = 0$, and the constant zero function is the only possible solution. But there is no way to manipulate the given equation to obtain $f(\underline{0}) = \underline{0}$ and we have lost our natural approach to computing the value of a function defined by a system of equations; i.e., computability is now in question.

To fix this problem, we have to insist that only finitely many substitution instances are required to obtain $f(\underline{\boldsymbol{x}}) = \underline{F(\boldsymbol{x})}$. If this additional condition is satisfied, then we say that F is finitely definable. Note that this constraint provides a method to compute F, at least in principle: we can systematically enumerate all substitution instances until the correct one appears.

Theorem 8.5.1 *Every computable function is finitely definable.*

Proof. The claim is obvious for basic functions and those obtained by composition and primitive recursion. Suppose $f(\boldsymbol{x}) \simeq \min\big(\, y \mid g(\boldsymbol{x}, y)\,\big)$. We may safely assume that addition, multiplication and the sign functions are available. Now consider the equations

$$\begin{aligned} f_0(\boldsymbol{x}) &= f_1(\boldsymbol{x}, \underline{0}) \\ f_1(\boldsymbol{x}, y) &= \mathsf{ifte}\big(g(\boldsymbol{x}, \underline{0}), y, f_1(\boldsymbol{x}, y^+)\big) \end{aligned}$$

This defines f via f_0. □

Taking a second look at the computation of $f(\underline{3}, \underline{2})$ above, it seems that we can do better: rather than having to search blindly, we can generate all the necessary substitution instances in a natural way, using only the following rules to manipulate equations:

- Substitution:
  Replace a free variable everywhere in an equation by a numeral.
- Replacement:
  A term s on the right hand side of an equation can be replaced by numeral t if we already have an equation $s = t$.

Here is an example of a derivation following these rules.

$$\begin{array}{lll}
(1) & f(\underline{3}, \underline{0}) = \underline{3} & \text{subst. of } \mathcal{E}_1 \\
(2) & f(\underline{3}, \underline{1}) = f(\underline{3}, \underline{0})^+ & \text{subst. } \mathcal{E}_2 \\
(3) & f(\underline{3}, \underline{2}) = f(\underline{3}, \underline{1})^+ & \text{subst. } \mathcal{E}_2 \\
(4) & f(\underline{3}, \underline{1}) = \underline{3}^+ = \underline{4} & \text{repl. (1), (2)} \\
(5) & f(\underline{3}, \underline{2}) = \underline{4}^+ = \underline{5} & \text{repl. (4), (3)}
\end{array}$$

For humans this is rather too tedious, but completely mechanical and easily implemented in any language with good support for pattern matching. Note that these rules are weaker than general equational reasoning. So we have a notion of derivability (or provability), usually written

$$E \vdash f(\underline{a_1}, \ldots, \underline{a_k}) = \underline{b}$$

We can now insist that the equations we want have to be derivable in this particular sense.

Definition 8.5.1 *A partial function $F : \mathbb{N}^k \nrightarrow \mathbb{N}$ is Herbrand-Gödel computable if there is a finite system of equations E that has an k-ary function symbol f such that*

$$E \vdash f(\underline{a_1}, \ldots, \underline{a_k}) = \underline{b} \iff F(a_1, \ldots, a_k) \simeq b$$

for all $a_i, b \in \mathbb{N}$.

It follows immediately that all primitive recursive functions are also Herbrand-Gödel computable: our definitions all have the required properties. In fact, we have the following theorem.

Theorem 8.5.2 *The Herbrand-Gödel recursive functions and μ-recursive functions coincide.*

Proof. Suppose we have a finite system E of equations that defines a (partial) function F. To show that $F(x) \simeq y$ we need to construct a derivation that uses only the given equations, plus substitution and replacement. With modest effort one can show that this whole machinery is primitive recursive in the following sense: there is a p.r. relation D such that

$$D(d, x, y) \iff d \text{ is a derivation of } f(\underline{x}) \simeq \underline{y} \text{ from } E$$

The derivation needs to be coded up as a sequence number d in the standard fashion. But then we can simply search for the least such d and extract the corresponding y.

For the opposite direction, we need to show how to express the min operator in terms of equations. Here is a trick due to Kleene from 1952. Assume $F(x) \simeq \min\big(z \mid g(z, x) = 0 \big)$. Introduce three new function symbols α, β and h with equations

$$\begin{aligned}
\alpha(x, y^+) &= x \\
\beta(x, \underline{0}) &= x \\
h(\underline{0}, x) &= g(\underline{0}, x) \\
h(y^+, x) &= \alpha(g(y^+, x), h(y, x)) \\
f(x) &= \beta(z, h(z, x))
\end{aligned}$$

This system of equations provides a Herbrand-Gödel definition of F via f. □

Example 8.5.1 Suppose

$$g(0, 5) = 3, \qquad g(1, 5) = 7, \qquad g(2, 5) = 0$$

We need to derive $f(5) = 2$. Note that $\alpha(x, 0)$ and $\beta(x, y^+)$ are both undefined. Let's calculate a few values for G:

$$\begin{aligned}
&G(0, 5) = g(0, 5) = 3 \\
&G(1, 5) = \alpha(g(1, 5), G(0, 5)) = g(1, 5) = 7 \\
&G(2, 5) = \alpha(g(2, 5), G(1, 5)) = g(2, 5) = 0 \\
&G(3, 5) = \alpha(g(3, 5), G(2, 5)) = \uparrow \\
&G(4, 5) = \alpha(g(4, 5), G(3, 5)) = \uparrow
\end{aligned}$$

Now recall $f(5) = \beta(z, G(z, 5))$, which looks bad since there is a free variable on the right hand side. But substituting $z \mapsto 0$ or $z \mapsto 1$ produces a divergent term on the right. Substituting $z \mapsto 2$ produces $f(5) = \beta(2, G(2, 5)) = 2$. And substituting $z \mapsto r$ for $r > 2$ also produces a divergent term on the right.

We can interpret Herbrand-Gödel equations as a programming language for computable functions, a language with a simple evaluation mechanism. As our examples show, there are typically several equations with principal symbol f. We can combine those into a single equation if we admit a definition by cases operator such as if-then-else:

$$\mathsf{ifte}(s, u, v) \simeq \begin{cases} u & \text{if } s \simeq 0 \\ v & \text{if } s \simeq c > 0, \\ \uparrow & \text{otherwise.} \end{cases}$$

Note that ifte can naturally be evaluated in lazy fashion: first s, then either u or v (assuming s returns a value). We now have a single equation with principal symbol f of the form

$$f(\boldsymbol{x}) = t(\boldsymbol{x}, f, \boldsymbol{g})$$

where t is a term involving the displayed variables and function symbols. After replacing $\boldsymbol{x}$ by numerals, we can perform a lazy evaluation of the right hand side. To make sure that solutions are computable, we need to insist that we consider only minimal ones: no values should be added that are not already forced by the equations. In other words, we are looking for the least fixed point of the equation.

It is interesting to take a closer look at how a computation based on a fixed point equation works. Suppose we are given partial function vector $\boldsymbol{G}$ and we have the solution F. For any particular argument $\boldsymbol{a} \in \mathbb{N}^n$, the evaluation will only require finitely many values of $\boldsymbol{G}$, so we could replace these functions by suitable finite approximations $\boldsymbol{G}'$. This is referred to as the Finite Support Principle:

$$t(\boldsymbol{a}, f, \boldsymbol{G}) \simeq b \Rightarrow \exists\, \boldsymbol{G}' \sqsubseteq \boldsymbol{G} \text{ finite } \big(t(\boldsymbol{a}, f, \boldsymbol{G}') \simeq b\big)$$

Moreover, we also have a Continuity Principle: if some, possibly finite, approximation $\boldsymbol{G}' \sqsubseteq \boldsymbol{G}$ already produces convergences, then $\boldsymbol{G}$ will do the same, with the same value:

$$t(\boldsymbol{a}, f, \boldsymbol{G}') \simeq b \wedge \boldsymbol{G}' \sqsubseteq \boldsymbol{G} \Rightarrow t(\boldsymbol{a}, f, \boldsymbol{G}) \simeq b$$

Those two ideas together can be combined to produce an elegant framework for the construction of computable functions. See section 9.1.1 for other versions of Kleene's recursion theorem.

Theorem 8.5.3 (Kleene's Second Recursion Theorem) *Suppose $\boldsymbol{G}$ is a vector of given partial functions. Every recursive equation $f(\boldsymbol{x}) = t(\boldsymbol{x}, f; \boldsymbol{g})$ has a least solution F. If the $\boldsymbol{G}$ are computable, so is F.*

Proof. Consider the operator on partial functions $\Gamma(F)(\boldsymbol{x}) = t(\boldsymbol{x}, F; \boldsymbol{G})$. Define F_0 to be the totally undefined function and let $F_{n+1} = \Gamma(F_n)$. It follows by Continuity that the sequence (F_n) is increasing, and we obtain a function $\widetilde{F} = \bigcup F_n$. By Finite Support, $\widetilde{F}$ is a fixed point of our equation and indeed the least fixed point.

When the base functions $\boldsymbol{G}$ and F are computable, the effect of Γ can be described by an elementary index operation γ so that the index of $\Gamma(F)$ is $\gamma(e, \boldsymbol{e})$ where e and $\boldsymbol{e}$ are indices for F and $\boldsymbol{G}$. Hence we can effectively obtain an index e_n for each approximation F_n of $\widetilde{F}$, and hence an index for $\widetilde{F}$ by search.

□

According to the theorem, we can set up one equation for each auxiliary function, and then use them all in the final definition of the desired computable function:

$$\begin{aligned}
g_1(\boldsymbol{x}) &= t_1(\boldsymbol{x}, g_1)\\
g_2(\boldsymbol{x}) &= t_2(\boldsymbol{x}, g_2; g_1)\\
&\dots\\
g_n(\boldsymbol{x}) &= t_n(\boldsymbol{x}, g_n; g_1, \dots, g_{n-1})\\
f(\boldsymbol{x}) &= t(\boldsymbol{x}, f; \boldsymbol{G})
\end{aligned}$$

In programming parlance, we first define a number of auxiliary subroutines, possibly by recursion, and then ultimately the function that we are interested in, again possibly by recursion. Given the power of recursion, the definitions of the auxiliaries and the main function can be relatively simple.

8.5.2 λ-Calculus

Machine or programming based explanations of computability are fairly well-aligned with intuitive notions of calculating some value $f(x)$, given x. This is the intensional or "functions as rules" perspective. A slightly awkward feature of all these models is that there are many technical details that are determined in a somewhat arbitrary manner. As we have seen, reasonable choices do not affect the computational power of the model, but it is natural to ask whether there is an approach to computability that avoids these issues altogether, a kind of minimalist approach to computation. We briefly describe such a model, the λ-calculus due to Alonzo Church, arguably the first full-fledged model of computation to emerge. The λ-calculus is more abstract than our previous models, so it is somewhat surprising that it has become a standard fixture in modern programming languages, and not only the strictly functional ones.

The main idea is to axiomatize functional composition, and avoid everything else, in particular data types. In the λ-calculus not even the natural numbers are built-in and have to be constructed. At first glance this may seem absurd, but it turns out to provide a powerful framework. In the λ-calculus (more precisely, in the basic, untyped form presented here), there is only one type of variable. For the time being, it is best to avoid the question of what the intended range of these variables is supposed to be; just think of a purely syntactical system. This takes some amount of getting used to, it stands in direct contrast to the intuitively compelling idea that in a function call $f(x)$ the argument x is somehow of a lower type than the actual function f.

Formally, λ-terms are defined inductively over an alphabet that contains special symbols

$$\lambda \qquad (\qquad) \qquad \mathbf{.}$$

and a countable supply of variables x, y, x_i, ... Each variable is an atomic term and compound terms are formed using one of two constructors

Application $(M\,N)$ is a term for terms M and N;

Abstraction $(\lambda x\,\mathbf{.}\,M)$ is a term for x a variable, M a term.

These correspond to function application and abstraction, respectively. For the latter, think of x as being a variable that is free in the term M. We obtain a function by replacing x in M with some term, and evaluating the result, whatever that may mean exactly in this context.

In the customary context-free grammar form, we can roughly describe these so-called λ-terms like so:

$$\begin{aligned} \langle\mathsf{term}\rangle &::= \langle\mathsf{var}\rangle \mid \langle\mathsf{abstr}\rangle \mid \langle\mathsf{appl}\rangle \\ \langle\mathsf{abstr}\rangle &::= (\,\lambda\,\langle\mathsf{var}\rangle\mathbf{.}\langle\mathsf{term}\rangle\,) \\ \langle\mathsf{appl}\rangle &::= (\,\langle\mathsf{term}\rangle\,\langle\mathsf{term}\rangle\,) \end{aligned}$$

Note that function application is written as $(f x)$ rather than the customary $f(x)$. Incidentally, f and x are really just two indistinguishable variables, our approach to naming them suggests that f represents a function, and x an argument, but that is all in the eye of the beholder. A variable is a variable. In the interest of legibility we adopt a few conventions that allow us to omit parentheses. First, application associates to the left; second, the body of a λ abstraction extends as far to the right as possible, third, we may collapse multiple λ's into a single one. So

$$\begin{array}{rcl} MNP & \text{means} & (MN)P \\ \lambda x\,\mathbf{.}\,MN & \text{means} & \lambda x\,\mathbf{.}\,(MN) \\ \lambda xyz\,\mathbf{.}\,M & \text{means} & \lambda x\,\mathbf{.}\,(\lambda y\,\mathbf{.}\,(\lambda z\,\mathbf{.}\,M)) \end{array}$$

The last convention is just syntactic sugar, strict λ terms always represent unary functions. It is straightforward to define free and bound variables, e.g., in $\lambda x \,.\, (xy)$ variable x is bound but y is free. Note that a variable may appear both free and bound in a term and some care is needed when performing substitutions $M[x/N]$.

So far, we only have a collection of terms without any connection to a computational process of any kind. The next step is to introduce rewrite rules that modify these terms. Think of them as reductions that simplify the term in question.

α-Reduction $\lambda x \,.\, M \xrightarrow{\alpha} \lambda y \,.\, M[x/y]$

β-Reduction $(\lambda x \,.\, M)N \xrightarrow{\beta} M[x/N]$

The α-rule simply replaces a variable by another, effectively renaming it; for this to make sense, the variable y must not occur in M (this is a non-issue since we have a countable supply of variables). The β-rule is critical, we evaluate an application term by replacing the λ variable x by the term N. Here, all variables free in N must remain free after the substitution of N for x, a condition that can be ensured by using the α-rule first. As usual, repeated application of these reductions produces a notion of equality modulo α and β reductions, in symbols $t \overset{\beta}{=} s$. A term is irreducible or in normal form if it cannot be rewritten using the β-rule, and reducible otherwise. We can think of the normal form as the "value" of the original term. Hence, this we obtain a kind of computation: starting with a term, we apply the reduction rules until we reach a term in normal form. In a sense, this comes down to simplifying the term as far as ever possible.

We note in passing that there are other reductions one could consider such as η-reduction:

$$\lambda x \,.\, Mx \xrightarrow{\eta} M$$

provided that x is not free in M. This is a kind of extensionality principle, but we will not pursue this here.

At any rate, there are a few problems to deal with. First off, a term may simply fail to have a normal form. The classical example is $\Delta = \lambda x \,.\, xx$ where $\Delta\Delta \xrightarrow{\beta} \Delta\Delta$. Since we are trying to construct another model of computation, this is actually good news: we have already seen that partial functions are unavoidable. Second, even if a term has a normal form, we cannot in general apply the reductions in arbitrary order; some strategies may succeed while others may lead to loops. A standard example for this effect is $(\lambda xy \,.\, y)(\Delta\Delta)z$: the reduction fails if one tackles the λ in D. On the other hand, according to a theorem by Church and Rosser, the rewrite system is well-behaved in the sense that for terminating sequences of reductions the order in which they are applied does not matter. Even better, there are strategies that guarantee success: for example, we can always tackle the leftmost reducible λ expression.

The question arises how much computational power the λ-calculus has. As it turns out, we can even model complicated computational operations such as recursion. In fact, there is an extremely elegant solution to this.

Theorem 8.5.4 (Kleene, Turing, Curry)
There is a fixed-point operator $\mathcal{Y}$ such that, for every term M: $\mathcal{Y}M \overset{\beta}{=} M(\mathcal{Y}M)$.

Proof. Let $D = \lambda x \,.\, M(xx)$ so that $Dt \xrightarrow{\beta} M(tt)$ for any term t. Hence $DD \xrightarrow{\beta} M(DD)$ and we have a fixed point. We can abstract M from this and get

$$\mathcal{Y} = \lambda y \,.\, (\lambda x \,.\, y(xx))(\lambda x \,.\, y(xx))$$

□

Booleans and Arithmetic

Missing from our discussion so far is any mention of data types or control structures, all we have is function application and abstraction, plus two very basic rewrite rules. We start with basic logic. We need to represent the Boolean values true and false as λ expressions. A little experimentation shows that $\mathsf{tt} = \lambda xy \,.\, x$ and $\mathsf{ff} = \lambda xy \,.\, y$ work: negation is then represented by $\lambda x \,.\, x\,\mathsf{ff}\,\mathsf{tt}$, disjunction by $\lambda pq \,.\, ppq$ and conjunction by $\lambda xy \,.\, xy\,\mathsf{ff}$.

Arithmetic requires a bit more effort. The key idea towards implementing natural numbers in the λ-calculus is to exploit iteration: we can think of $f(f(f(z))) = f^3(z)$ as a representation of 3. So we think of $\lambda fz \,.\, z$ as representing 0, $\lambda fz \,.\, fz$ stands for 1, $\lambda fz \,.\, f(fz)$ for 2, and so on. More generally, define the Church numerals to be the terms

$$\underline{n} = \lambda fx \,.\, f^n(x)$$

These numerals open the door to expressing arithmetic functions in terms of the λ-calculus: we can try to find specific terms that, when a applied to a numeral or several numerals have a normal form that is yet another numeral. The latter will be interpreted as the output of the computation. For example, we have the following analogue to the successor function:

$$S = \lambda xfz \,.\, f(xfz)$$

Similarly we can represent addition and multiplication by the terms

$$\lambda mnfx \,.\, mf(nfx) \qquad \lambda mnf \,.\, m(nf)$$

Perhaps surprisingly, the first major stumbling block towards developing a library of λ terms for arithmetic functions is the predecessor function: we somehow have to remove an application of f from the Church numeral $\lambda fx \,.\, f^n(x)$. Kleene solved this problem, allegedly while waiting in a dentist's office. The idea is that one can generate a sequence of triples, written informally

$$(0, 0, 1), (0, 1, 2), (1, 2, 3), (2, 3, 4) \ldots$$

and then project away the second and third components, see the exercises.

Definition 8.5.2 *An arithmetic function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ is λ-definable if there is a term M such that $f(a_1, \ldots, a_k) \simeq b$ iff $M\underline{a}_1\underline{a}_2 \ldots \underline{a}_k$ reduces to $\underline{b}$.*

One might object that this definition is too dependent on Church numerals, but other definitions such as Scott numerals produce the same class of λ-definable functions. In fact, we obtain precisely all intuitively computable functions this way.

Theorem 8.5.5 (Church, Kleene, Rosser)
An arithmetic function is λ-definable if, and only if, it is partial recursive.

Sketch of proof. The first step is to show that all primitive recursive functions are λ-definable, so it remains to deal with the minimization operator $f = \mathsf{Min}(g)$. Recall that $f(\boldsymbol{x}) \simeq \min\big(z \mid g(z, \boldsymbol{x}) = 0 \wedge \forall\, s < z\, (g(z, \boldsymbol{x}) > 0)\big)$. We can rephrase the last description as $f(\boldsymbol{x}) \simeq F(0, \boldsymbol{x})$ where

$$F(z, \boldsymbol{x}) = \begin{cases} z & \text{if } g(z, \boldsymbol{x}) = 0, \\ F(z+1, \boldsymbol{x}) & \text{otherwise.} \end{cases}$$

Lastly, we can exploit the fixed-point theorem to show that F is λ-definable.

For the opposite direction, first note that the predicate "t codes a β-reduction from M to N" is primitive recursive: we can arithmetize everything to translate all objects to natural numbers. Each step in a β-reduction is clearly primitive recursive, as is the whole sequence. But then we need to add just one unbounded search to find the appropriate t. □

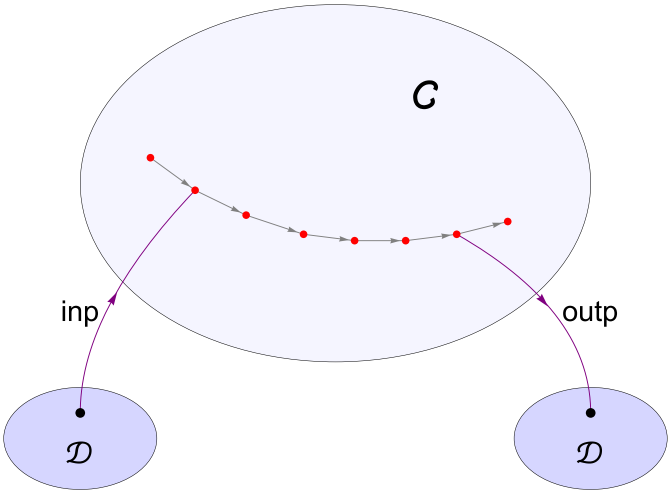
As to the question of how one should explain the domain over which the variables in the λ-calculus range, we punt: suffice it to say that simple set theoretic models fail, and considerable machinery from category theory is required to construct semantic models.

8.5.3 Church-Turing Thesis

fix conf space

At this point we have accumulated a number of models of computation that, at least on the face of it, seem rather disjointed. And yet, with some amount of effort, one can show that they all determine the same class of computable arithmetic functions. One way to explain this confluence is to recall the configuration spaces from section 8.4.1. These spaces are most natural in the context of machine models, but they can also be invoked for the other models. For example, in the λ calculus we can think of terms as the points in the space, and the "next-step" relation here is derived from α and β reductions. The input and output maps rely on Church numerals to translate natural numbers to λ terms and back.

The unifying feature is that the next-step relation is primitive recursive in all cases, and the complexity of a computation is related to the number of steps needed to produce a terminal configuration that contains the result. In particular, we cannot compute the length of a computation ahead of time, we can only attempt to carry it out in full and hope to get to a terminal configuration after finitely many steps. The technical details vary greatly, but the basic structure is always the same. Here is the picture of a configuration space again.



As an aside, the picture suggests a deterministic system where there is at most one next configuration, but it is clear from the λ calculus and Herbrand-Gödel equations that nondeterministic computation may also be of interest. In fact, it has turned out to be critical in complexity theory and the theory of algorithms, starting with the seminal 1959 paper on finite state machines by Rabin and Scott.

Church was the first to propose to take things one step further and declare that the λ-definable functions are the correct formalization of the intuitive concept of computable function. At the time, primitive recursive functions were known, as were several generalizations such as Ackermann's function, definable by double recursion or transfinite recursion; there were all captured by the λ calculus. This proposal came to be known as Church's Thesis, but it was met with significant resistance from Gödel, who believed that there simply was not enough evidence at the time for such wide-ranging claims. Based on a proposal by Herbrand, Gödel defined Herbrand-Gödel recursive functions in 1934, but these too turned out to be λ-definable. A little later, in 1936, when Alan Turing introduced his eponymous machines, Gödel responded most enthusiastically:

> This concept, ...is equivalent to the concept of a "computable function of integers" ...The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.

Gödel always gave full credit to Turing, never to Church or himself, and we now speak of the Church-Turing Thesis. At the time of this writing, there are no reasonable objections to the claim that our numerous models all exactly capture the idea of intuitive computability. It is a labor of love to check that the models we have encountered so far are in fact all equivalent. Note, though, the necessary simulations are always quite natural, adding to the appeal of the Church-Turing Thesis.

Theorem 8.5.6 *For any partial function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$, the following are equivalent:*

1. *f is Herbrand-Gödel recursive,*
2. *f is μ-recursive,*
3. *f is while-computable,*
4. *f is register machine computable, and*
5. *f is Turing-computable.*

In light of the Church-Turing Thesis it is justified to simply speak of computable functions, decidable relations and so on, without specifying a particular model of computation. For specific models of computation such as register machines or Turing machines, it seems clear that they can be realized as physical systems, rather than abstract, mathematical concepts.

It is tempting to push things even further and to contend that our definitions capture exactly all physically realizable computations. Restricting physics to just classical mechanics, G. Kreisel proposed the following in the 1960s:

> **Thesis M:** The behavior of any discrete physical system evolving according to local mechanical laws is recursive.

Since a Turing machine can compute recursive functions, this means that a Turing machine can also simulate all the physical systems in the thesis. This assertion is far less contentious than any claim along the lines of "every physical process can be simulated by a S-machine." We can conclude from Thesis M that any function that is physically computable in some way is also Turing computable.

Friedman

Tucker

Of course, this leaves out quantum physics and relativity theory. If we accept the thesis, the question is whether the opposite direction also holds: every, say, Turing machine can be implemented by a physical system. This assertion is much more problematic, one needs to hedge quite a bit. For example, no one would claim that computations associated with Goodstein sequences are realizable in any reasonable sense of the word, the required resources in terms of space, time and energy are nowhere near available in our actual universe. Instead, one typically argues that these computations can be carried out *in principle*.

More precisely, suppose we have an axiomatization of some fragment of physics, say classical mechanics. The axiomatization represent (some of) the laws of physics, but does not involve any constraints regarding total mass, energy and so forth. We could then rigorously prove that, say, arbitrary Turing machine computations can be implemented in this setting. The argument should be amenable to verification in a theorem prover. Axiomatization of physics was first proposed by Hilbert's in his sixth problem (mechanics and probability), but at present there is no system that encompasses all of physics, including in particular quantum physics and relativity. Not to mention a problem raised in a quip by M. Tegmark: "In physics it's not enough to be right; you have to be right for the right reasons."

8.5.4 Exercises

Exercise 8.5.1 Show in detail how to simulate a register machine by a system of Herbrand-Gödel equations.

Exercise 8.5.2 Explain how the approach in Kleene's Second Recursion Theorem can be adapted to handle mutual recursion as in the even/odd example above.

Exercise 8.5.3 Show that there are unary total recursive functions f_1 and f_2 such that (f_1, f_2) has range $\mathbb{N}^2$.

Exercise 8.5.4 Show that there is a strictly increasing elementary function f such that $\{e\} \simeq \{f(e)\}$.

Exercise 8.5.5 Consider a class of register machines RM' with a slightly modified instruction set:

`clear r`	set register R_r to 0
`incr r`	increment register R_r, continue with next location
`equt r s l k`	test whether $[R_r] = [R_s]$, if so, continue with location $l$else with location k
`halt`	well ...

Show that RM' defines the same class of partial functions as RM. Show that there is an elementary "compiler" that translates programs from one class to the other, in both directions.

Exercise 8.5.6 Show that f total recursive and A recursive implies $f^{-1}(A)$ recursive. Show that f partial recursive and A recursively enumerable implies $f^{-1}(A)$ recursively enumerable.

Exercise 8.5.7 Show that exponentiation 2^x and super-exponentiation $2 \uparrow x$ are loop-computable.

Exercise 8.5.8 Show that the next-prime function $n \mapsto \min\big(x > n \mid x \text{ prime}\big)$ is loop-computable.

Exercise 8.5.9 Show that $x \bmod m$ is $\texttt{Loop}(1)$-computable for ever fixed modulus m.

Exercise 8.5.10 Show that $x \operatorname{div} m$ is $\texttt{Loop}(1)$-computable for ever fixed modulus m.

Exercise 8.5.11 Give a detailed proof of theorem 8.5.2.

Exercise 8.5.12 A Markov algorithm is deterministic string rewrite system with a list (not a set) of productions. Some productions are marked as terminal and can only be applied at the last step of a derivation. To apply Markov rules, find the first applicable rule in the list, and apply it to the left-most-shortest handle. If no rule applies, the derivation also terminates.

Show that the computable string functions are exactly the functions determined by Markov systems.

Exercise 8.5.13 Show how to implement predecessor, addition and multiplication in the λ-calculus.

Exercise 8.5.14 Define addition and multiplication using the fixed-point operator $\mathcal{Y}$.

Exercise 8.5.15 Implement lists of naturals in the λ-calculus, together with operations such as head, tail, append, and join.

Exercise 8.5.16 Explain how to simulate Turing machines in the λ-calculus.

Nine

Fundamental Properties of Computation

9.1 Fundamentals

Since computable functions can be defined in many different ways, it is well worthwhile to try to isolate their key properties that are independent of any particular model. We will not push so far as to try to axiomatize computability, we will simply produce a list of fundamental results. As a matter of principle, we could establish these basic properties separately for all our models. Little would be gained from all this effort, so instead we reason less formally in terms computable functions, decidable relations, universal devices and so on. The gentle reader should verify that all our claims can be made concrete in one particular model, or maybe two structurally different ones.

9.1.1 Universality and Enumeration

Arguably the most basic and important aspect of computability is the existence of universal systems such as the universal register machine from theorem 8.4.1. From a computer science perspective this may seem fairly quaint; after all, any standard digital computer, together with the requisite operating system and a compiler, is universal, at least if we ignore physical constraints. But note the historical flow of events: Turing wrote his seminal paper in 1936, before the advent of digital computers.

In any model, there are "universal devices" $\mathcal{U}$ such that for any computable function f there is a corresponding index e such that: $f(x) \simeq \mathcal{U}(e, x)$ for all x. Here is a formulation using computable functions that avoid the issue of having to explain what exactly is meant by a device.

Theorem 9.1.1 (Enumeration Theorem) *For each arity n, there exists a partial computable function $\Phi : \mathbb{N} \times \mathbb{N}^n \nrightarrow \mathbb{N}$ such that for every partial computable function $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ there exists an index $\widehat{f}$ of f such that $f(\boldsymbol{x}) \simeq \Phi(\widehat{f}, \boldsymbol{x})$ for all $\boldsymbol{x} \in \mathbb{N}^n$.*

The device here is the computable function Φ, no less and no more. A stronger statement holds true: given any reasonable description of a computable function f, we can determine an index $\widehat{f}$ in a primitive recursive or even elementary manner. Indices are very simple data structures, and we can manipulate them effectively. Here is a first example of this idea. There are many ways to design a universal device, and each brings its own notion of index. But there is no issue, we can translate back and forth between some enumeration function Φ and another enumeration function Φ'.

Lemma 9.1.1 *There is a primitive recursive function α such that $\Phi(e, \boldsymbol{x}) \simeq \Phi'(\alpha(e), \boldsymbol{x})$.*

Note that this remains true even when we establish a link between different models, say, Turing machines and the λ-calculus. At any rate, there is no harm in simply choosing one particular enumeration once and for all. The notation $\varPhi(e, \boldsymbol{x})$ is a bit heavy-handed, so in the future will follow Kleene and write

$$\{e\}(\boldsymbol{x})$$

for the result of applying the function with index e to some argument vector $\boldsymbol{x}$. We can be more explicit about the way $\{e\}(\boldsymbol{x})$ is obtained by an unbounded search applied to a primitive recursive predicate.

Theorem 9.1.2 (Normal Form Theorem (Kleene)) *There is a primitive recursive predicate* T *and a primitive recursive function* U *such that for any computable function* $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ *there is an index* e *such that*

$$\begin{aligned} \{e\}(\boldsymbol{x})\downarrow &\iff \exists\, t\ \mathsf{T}(e, \boldsymbol{x}, t) \\ \{e\}(\boldsymbol{x}) &\simeq U(\min\big(t \mid \mathsf{T}(e, \boldsymbol{x}, t)\big)) \end{aligned}$$

Proof. Say, we use register machines to verify this claim, though the argument requires little modification for the other models. Any initial segment of a computation $C_0, C_1, \ldots, C_{N-1}$ of the register machine with index e can be coded as a sequence number of sequence numbers. The predicate $\mathsf{T}(e, \boldsymbol{x}, t)$ then checks that t is indeed the code of a computation of device number e, that the initial configuration corresponds to input $\boldsymbol{x}$, and that the last configuration is halting. The decoding function U simply extracts the result of this computation, the value of one of the registers in the final configuration coded by t. □

The decoding function U may appear to be a bit of a nuisance, but the theorem does not hold without it. Intuitively, it may be helpful to think of t simply as the length of the computation: t is bounded by an elementary function of e, $\boldsymbol{x}$ and the length. Thus, the unbounded element here is the number of steps in the computation.

Enumeration is interesting since it forces the existence of partial functions. We have seen this before, but it is worthwhile to repeat the argument in this more general setting. Let $\mathcal{F}$ be any clone that contains the successor function and an enumeration function $\varPhi$. We claim that $\mathcal{F}$ must contain partial functions. For let $f(x) \simeq S \circ \varPhi \circ (\mathsf{P}^{(1)}_1, \mathsf{P}^{(1)}_1)$. Then $f \in \mathcal{F}$ and must have an index $\widehat{f}$. But $f(\widehat{f}) \simeq f(\widehat{f}) + 1$, hence $f(\widehat{f})\uparrow$.

Also note that there has to be a partial computable function that is not the restriction of any total computable function. For let

$$g(x) = \begin{cases} \{x\}(x) + 1 & \text{if } \{x\}(x)\downarrow, \\ \uparrow & \text{otherwise.} \end{cases}$$

Another index argument shows that g cannot be the restriction of any total computable function, and g is clearly computable.

Once we have a fixed enumeration, one can compute with indices relative to that enumeration very much in the same way one manipulates programs in computer science. Here is a particularly useful, though admittedly quite unspectacular instance of such an index computation: we can fix some of the arguments of a computable function to obtain another computable function.

Theorem 9.1.3 (S-m-n Theorem) *There exists an elementary injective function* $\mathsf{S}^n_m : \mathbb{N}^{m+n} \to \mathbb{N}$ *such that for all* $e \in \mathbb{N}$, $\boldsymbol{x} \in \mathbb{N}^m$, $\boldsymbol{y} \in \mathbb{N}^n$*:* $\{e\}(\boldsymbol{x}, \boldsymbol{y}) \simeq \{\mathsf{S}^n_m(e, x)\}(\boldsymbol{y})$.

As another example of index computation, we claim that there is a primitive recursive function f such that

$$\{f(e,e')\} \simeq \{e\} \circ \{e'\}.$$

This corresponds to sequential composition of programs. Similarly we could handle definition by cases:

$$\{g(e,e')\}(x) \simeq \mathsf{ifte}\big(x, \{e\}(x), \{e'\}(x)\big).$$

The Recursion Theorem

Let us return to the topic of recursive definitions of computable functions as first encountered in theorem 8.5.3 and in section 8.5.2. This type of result often admits very elegant and concise definitions of concrete computable functions. Unfortunately, the next proof will be a bit hard to swallow. As motivation for the result from the computer science perspective, consider an interpreter F for computable functions: F takes as inputs a program e together with an argument x, and returns the result of evaluating $\{e\}$ on x. The computation may diverge, in which case the interpreter also produces no result. Hence $F(e,x) \simeq \{e\}(x)$. The next theorem says that a similar result holds true for any computable function F.

Theorem 9.1.4 (Recursion Theorem, Kleene) *Let $F : \mathbb{N}^{n+1} \nrightarrow \mathbb{N}$ be a partial recursive function. Then there exists an index $e^\star$ such that for all $x \in \mathbb{N}$: $\{e^\star\}(\boldsymbol{x}) \simeq F(e^\star, \boldsymbol{x})$.*

Proof. Define $h(e,\boldsymbol{x}) \simeq F(S^1_n(e,e),\boldsymbol{x})$ and let $\widehat{h}$ be an index for h. Set $e^\star := S^1_n(\widehat{h},\widehat{h})$. Then

$$\begin{aligned}
\{e^\star\}(\boldsymbol{x}) &\simeq \{S^1_n(\widehat{h},\widehat{h})\}(\boldsymbol{x}) \\
&\simeq \{\widehat{h}\}(\widehat{h},\boldsymbol{x}) \\
&\simeq h(\widehat{h},\boldsymbol{x}) \\
&\simeq F(S^1_n(\widehat{h},\widehat{h}),\boldsymbol{x}) \\
&\simeq F(e^\star,\boldsymbol{x})
\end{aligned}$$

□

At first glance, this seems quite impossible; e.g., what if $F(e,x) \simeq \{e\}(x) + 1$. But, all is well: recall that partial functions are unavoidable in computability; in this case any totally undefined function can serve as $\{e^\star\}$. The last proof, while formally correct, is rather frustrating since it provides little insight into the nature of the index $e^\star$. Indeed, J. C. Owings called the proof "barbarically short" and "nearly incapable of rational analysis." Owings also suggested that one could think of the proof of the recursion theorem as a diagonal argument that fails. In general, in a diagonal argument, we have an infinite matrix S over some set A:

$$S : \mathbb{N} \times \mathbb{N} \to A$$

Viewed differently, S is a one-dimensional table of infinite sequences over A. Furthermore, we have some operation α on A such that the sequence obtained by applying α to the diagonal $\big(\alpha(S(i,i))\big)_{i\geq 0}$ is not in S. Now suppose f is computable and well-behaved on indices: $\{e\} \simeq \{e'\}$ implies $\{f(e)\} \simeq \{f(e')\}$; so f preserves input/output behavior, though not necessarily the structure of the computation. Define a matrix S of computable functions by

$$S = \big(\{\{i\}(j)\}\big)_{i,j}$$

and let $\alpha(\{e\}) = \{f(e)\}$. In this case, when we are trying to diagonalize out of S, the diagonal sequence $(S(i,i))_{i \geq 0}$ as well as its image under α is still a row in S. The intersection of this row and the diagonal is the function we are looking for.

Here is a version of the recursion theorem due to H. Rogers that is perhaps a bit easier to accept intuitively. For any function $f : \mathbb{N} \to \mathbb{N}$, we refer to e as a fixed point, if $\{f(e)\} \simeq \{e\}$. This terminology clashes with the standard meaning of fixed point, but it is always clear from context which version is intended. One can think of f as a program transformation; the theorem then says that, for any such transformation, there is always a program that is not affected by it (as far as input/output behavior is concerned).

Corollary 9.1.1 (Recursion Theorem, Fixed Point Version) *Let $f : \mathbb{N} \to \mathbb{N}$ be a recursive function. Then f has a fixed point: $\{e^\star\} \simeq \{f(e^\star)\}$ for some $e^\star$.*

Proof. Again by the S-m-n theorem, there is a total recursive function h such that

$$\{h(x)\} \simeq \{\{x\}(x)\}.$$

Let e be an index for the composition $f \circ h$ and set $e^\star := h(e)$. Then

$$\{e^\star\} \simeq \{h(e)\} \simeq \{\{e\}(e)\} \simeq \{f(h(e))\} \simeq \{f(e^\star)\}$$

□

See the exercises for a proof of the fixed point theorem using the recursion theorem. There is an important theorem by Arslanov that says, in essence, that to avoid fixed points one needs an oracle at least as strong as the Halting Set.

Here is an informal way to make sense out of the fixed point result. Think of a typical program Q that defines some computable function:

Q:

```
input $x$
somehow compute $y$
return $y$
```

It is perfectly possible that program Q internally uses a subprogram P, which subprogram we can represent by its index:

Q:

```
input $x$
somehow compute $y$, using $e$
return $y$
```

This part is entirely uncontroversial. But with the recursion theorem, we can push further and use an index q for Q itself inside its very definition:

Q:

```
input $x$
somehow compute $y$, using $q$
return $y$
```

This sort of self-reference is plausible when one considers computation on a digital computer: one can imagine a scenario where a program in memory has access to its own source code and a compiler. Here are a few typical applications of the recursion theorem.

Example 9.1.1 Let $F(e,x) \simeq e$. Then $e^\star \simeq \{e^\star\}(x)$, so there is a computable function that prints its own index, regardless of the input. Note that an implementation of this result in any concrete programming language is quite a challenge: write a program in some actual programming language that, when compiled and executed, prints its own program text (these programs are called quines).

Example 9.1.2 Let $F(e,x) \simeq \{x\}(e)$ so that $\{x\}(e^\star) \simeq \{e^\star\}(x)$. Thus, there is a computable function that, on input x, runs computable function $\{x\}$ on its own index $e^\star$.

Example 9.1.3 We can define the Ackermann function without having to worry about justifying the double recursion. Of course, we still need to establish the function defined by Q is total, but there is no question that it is computable. Let

$$F(e,x,y) \simeq \begin{cases} y+1 & \text{if } x = 0, \\ \{e\}(x-1,1) & \text{if } x > 0, y = 0, \\ \{e\}(x-1,\{e\}(x,y-1)) & x,y > 0. \end{cases}$$

Then $\{e^\star\}(x,y) \simeq A(x,y)$.

9.1.2 Stages of a Computation

When we consider an actual computation on some input in any of our models, it is clear that the desired output does not appear immediately: the computation has to proceed through a number of stages before the ultimate result appears–and, of course, the computation may be divergent and no result appears ever. Stages are important since they help in organizing computations in the abstract. As an example, suppose we have two computable functions f and g and we want to determine whether at least one them converges on input x. We cannot execute f and g on x sequentially: if the first computation diverges we never get to the second, potentially convergent, computation. Instead, we have to interleave the steps of the computations: we perform one step in the computation of $f(x)$, then one step in $g(x)$, then $f(x)$, and so on. We stop if one of the two sub-computations terminates.

How can we make the notion of a stage precise? We claim that, in any concrete model, there is a canonical way of counting steps. This most obvious for register machines and Turing machines: we can count how many times the one-step relation needs to be applied to obtain the configuration in question. A similar approach also works for Herbrand-Gödel and the λ-calculus. To provide a more systematic answer, we can exploit Kleene's T predicate to define formally the stages σ, $\sigma \geq 0$, of a computation. Given a computable function $f : \mathbb{N}^n \to \mathbb{N}$, let

$$\{e\}_\sigma(\boldsymbol{x}) = \begin{cases} y & \text{if } \exists\, t < \sigma \left(\mathsf{T}(e,\boldsymbol{x},t) \wedge U(t) = y\right), \\ \sigma & \text{otherwise.} \end{cases}$$

Thus $\{e\}_\sigma(\boldsymbol{x})$ is an approximation to $\{e\}(\boldsymbol{x})$. In particular, if $\{e\}(\boldsymbol{x})\downarrow$, then this approximation converges

$$\{e\}_\sigma(x) < \sigma \wedge \sigma \leq \tau \text{ implies } \{e\}_\sigma(x) = \{e\}_\tau(x)$$

Thus, in the discrete topology, $\{e\}(x) \simeq \lim_{\sigma\to\infty}\{e\}_\sigma(x)$. It is often helpful to assume $\{e\}_\sigma(x) = y$ implies that $e,x,y < \sigma$. This is easily enforced by using appropriate coding functions. Also, we slightly abuse notation and write $\{e\}_\sigma(e)\downarrow$ instead of $\{\}_\sigma(z) < \sigma$, and we set $\mathsf{spt}\{e\}_\sigma = \{\, z < \sigma \mid \{e\}_\sigma(z)\downarrow \}$.

Lemma 9.1.2 *A function f is computable if, and only if, there exists an elementary function f' such that $f(x) \simeq \lim_{\sigma\to\infty} f'(x,\sigma)$ and*

1. $f'(x,\sigma) \leq \sigma$

2. *$f'(x, \sigma) \simeq f(x)$ for all σ such that $f'(x, \sigma) < \sigma$*
3. *$\forall\, \sigma\, (f'(x, \sigma) = \sigma)$ implies $f(x) \uparrow$*

The same holds true when f' is required to be primitive recursive instead.

Proof. We have just seen the implication from left to right. For the opposite direction, assume we have f' as in the lemma. Perform an unbounded search for σ such that $f'(x, \sigma) < \sigma$ (the computation diverges if no such σ exists). Return $f'(x, \sigma)$. □

In a situation as in the lemma we will write $f = \lim f'$. As we will see, this method of constructing computable functions is extremely useful. Here is one example: suppose we have a computable function $f : \mathbb{N} \to \mathbb{N}$ and we would like to define a new function that tests whether one of the family of functions $\{f(n)\}$ converges on a given input x:

$$g(x) \simeq \begin{cases} 0 & \text{if } \exists\, n\, (\{f(n)\}(x) \downarrow), \\ \uparrow & \text{otherwise.} \end{cases}$$

Then g is again computable. However, we cannot simply start the computation of $g(x)$ by determining $\{f(0)\}(x)$, then $\{f(1)\}(x)$, and so forth: any one of these computations might diverge and derail the whole search. But we can organize the computation in stages. At stage σ, we compute $\{f(n)\}_\sigma(x)$ for all $n < \sigma$. If one of these computations converges, we return 0; otherwise we move on to the next stage. All the activities at stage σ are primitive recursive in σ and x. Thus, in a way, we can combine all the computations $\{f(n)\}(x)$. This technique is often referred to as dovetailing.

9.1.3 Exercises

Exercise 9.1.1 Let I be the set of all indices e such that $W_e \neq \emptyset$. Show that I is r.e.-complete.

Exercise 9.1.2 Prove Rice's theorem using the recursion theorem.

Exercise 9.1.3 Provide a more formal framework to justify the "any model is fine" approach in this section. For example, you might want to consider a transition system $\langle C, \rightarrow \rangle$ where $C \subseteq \mathbb{N}$ is a set of configurations and $\rightarrow$ is a primitive recursive next-step relation.

9.2 Decidability and Semidecidability

In 1928, in the book *Grundzüge der theoretischen Logik*, D. Hilbert and W. Ackermann proposed a fundamental challenge, the so-called Entscheidungsproblem, the decision problem.

> The Entscheidungsproblem is solved when one knows a procedure by which one can decide in a finite number of operations whether a given logical expression is generally valid or is satisfiable. The solution of the Entscheidungsproblem is of fundamental importance for the theory of all fields, the theorems of which are at all capable of logical development from finitely many axioms.

Note the slight hedge: the area has to be "capable of development" from finitely many axioms. Still, as Herbrand pointed out "In a sense, the Entscheidungsproblem is the most general problem of mathematics." This will turn out to be an excellent source of difficult problems in complexity theory: instead of answering a concrete combinatorial question, we try to deal with all possible questions that can be formulated within a certain framework. If the framework is is sufficiently general, one may expect to encounter very difficult questions in this manner.

At the time when Hilbert proposed the Entscheidungsproblem, there was no generally accepted mathematical definition of what precisely a "procedure" might be. In modern parlance, we would refer to algorithms, or, more abstractly, to computability. There is strong evidence that, at the time, Hilbert considered all well-posed problems to be solvable in the sense outlined in the Entscheidungsproblem. Alas, in the 1930s, work by Gödel, Church, Turing and Post showed that algorithmic unsolvability exists in mathematics, and cannot be eliminated. In particular the Entscheidungsproblem itself turned out to be highly unsolvable.

As usual, we can use our standard approach of translating relations into functions to give a precise definition of what we mean by decidability.

Definition 9.2.1 *A set $A \subseteq \mathbb{N}^n$ is decidable if its characteristic function char_A is computable.*

Thus, informally, the decidable sets are those that have a decision algorithm, a method that determines membership in the set: for any $\boldsymbol{x} \in \mathbb{N}^n$ we can determine in finitely many steps whether $\boldsymbol{x} \in A$ or $\boldsymbol{x} \notin A$. In computer science, it is now standard to spell out a decision problem P in the format

- a set $I \subseteq \mathbb{N}^n$, the set of instances, and
- a set $Y \subseteq I$, the set of Yes-instances.

As a typical example, consider primality testing. In the standard idiom motivated by complexity theory, this decision problem would be described like so:

Problem:	**Primality**
Instance:	A natural number x.
Question:	Is x a prime number?

We specify the name of the problem, the collection of instances, and the property of these instances that one wishes to ascertain. Though prime numbers have been studied for more than two millennia, it was demonstrated only recently that a primality test can be performed in time polynomial in the size of the input, which is given in standard binary notation. Surprisingly, only basic concepts from number theory and a little polynomial arithmetic are required for the algorithm.

In general, we may safely assume the set I of all instances is rather simple; at the very least I will be elementary. Thus the set Y of Yes-instances is decidable if, and only if, the function

$$f(\boldsymbol{x}) = \begin{cases} \mathsf{char}_Y(x) & \text{if } \boldsymbol{x} \in I, \\ 0 & \text{otherwise} \end{cases}$$

is computable.

In applications, I is typically some set of (finite) data structures such as integers, trees, graphs and so on. Since coding details do not matter as this point (very much unlike the situation in complexity theory), we will speak of a set of integers or trees or graphs and so forth as being decidable, rather than the corresponding sets of words over some suitable alphabet. In fact, we may safely assume that this alphabet is $\mathbf{2} = \{0, 1\}$. There is no harm in this as long as we are interested in general computability; when it comes to efficient computation and certain low complexity classes one has to be a bit more careful, see chapter **??**.

Lemma 9.2.1 *The decidable sets are closed under union, intersection and complement. More precisely, the decidable sets form an effective Boolean algebra.*

We mention one more example of a decision problem that has a long history. It became prominent when Hilbert included it in his list of fundamental problems in Mathematics presented at the International Congress of Mathematicians in 1900 in Paris.

Problem:	**Diophantine Equations**
Instance:	A multivariate polynomial $P(x_1, \ldots, x_n)$ with integer coefficients.
Question:	Does the equation $P(x_1, \ldots, x_n) = 0$ have integral roots?

As a concrete example of such an equation, consider the following, dating back to Diophantus:

> Find three whole numbers such that the product of any two of them added to the third is a square.

In modern terminology, we are looking for positive natural numbers such that

$$(xy + z - u^2) + (xz + y - v^2) + (yz + x - w^2) = 0$$

There are quite a few solutions, for example, $(2, 3, 6)$ and $(2, 11, 18)$. Try to find a few more solutions without the help of a computer.

In the absence of any other information, one fairly obvious approach would be to systematically enumerate all n-tuples of integers and evaluate the polynomial on each of them. If there is a solution, we are guaranteed to find it. On the other hand, if there is no integral solution, the search will continue indefinitely. We refer to such a method as a semialgorithm, a broken kind of algorithm that only works properly on Yes-instances.

We could turn the semiaalgorithm into an actual decision algorithm by finding a computable bound on the size of any potential solution. Presumably, this bound would depend on the number of variables, the degree of the polynomial, the size of the coefficients and such like. For univariate polynomials this is easy, but even for quadratic polynomials in two variables there are problems. As an example, consider

$$x^2 - 991y^2 - 1 = 0.$$

Of course, $x = 1$, $y = 0$ is a trivial solution. But the smallest positive solution here is

$$\begin{aligned} x &= 379516400906811930638014896080 \\ y &= 12055735790331359447442538767 \end{aligned}$$

Number theory has unearthed a wealth of information about these types of equations, but there are no easy answers in general. Indeed, it follows from Matiyasevic's undecidability result that there is no computable bound on the size of potential solutions.

To formalize the notion of a semialgorithm, define the semi-characteristic function schar of $A \subseteq \mathbb{N}^n$ to be

$$\mathsf{schar}_A(x) = \begin{cases} 0 & \text{if } x \in A, \\ \uparrow & \text{otherwise.} \end{cases}$$

Definition 9.2.2 *A set $A \subseteq \mathbb{N}^n$ is semidecidable if its semi-characteristic function is computable.*

In other words, a set is semidecidable if, and only if, it is the support of a partial recursive function: the function converges on all Yes-instances, and diverges on all No-instances. From an algorithmic perspective, semidecidability may seem like a bizarre concept, but, in many ways, it is even more important than decidability. This can be seen in part from the next result.

Lemma 9.2.2 *A set $A \subseteq \mathbb{N}^n$ is decidable if, and only if, A itself and its complement $\overline{A} = \mathbb{N}^n - A$ are both semidecidable.*

Proof. It is easy to modify the characteristic function of A to produce semi-characteristic functions for A and $\overline{A} = \mathbb{N}^n - A$. So suppose that schar_A and $\mathsf{schar}_{\overline{A}}$ are computable. By computing both functions in stages, we can define

$$f'(x,\sigma) = \begin{cases} 1 & \text{if } \mathsf{schar}(A)_\sigma \simeq 0, \\ 0 & \text{if } \mathsf{schar}(\overline{A})_\sigma \simeq 0, \\ \sigma & \text{otherwise.} \end{cases}$$

But then $f(x) = \lim_{\sigma\to\infty} f'(x,\sigma)$ is computable and is none other than the characteristic function of A. □

For any set $A \subseteq \mathbb{N}$ define its principal function (aka Hauptfunktion) H_A by recursion as follows.

$$H_A(x) \simeq \min\big(\, z \in A \mid \forall\, x' < x\, (H_A(x') < z)\,\big)$$

Thus, H_A enumerates the elements of A in order. It is easy to see that H_A has as domain an initial segment of $\mathbb{N}$. In particular for A finite, the support of H_A has the form $\{0, 1, \ldots, n-1\}$ where n is the cardinality of A.

Lemma 9.2.3 *Let $W \subseteq \mathbb{N}$ be semidecidable. Then W is decidable if, and only if, its principal function H_W is computable.*

Proof. If W is decidable, then the definition of H_W shows that this function is computable. On the other hand, suppose the principal function is computable. Without loss of generality, we consider only the case where W is infinite, so that the domain of definition of H_W is $\mathbb{N}$. Then

$$x \notin W \iff x < H_W(0) \lor \exists\, s\, (H_W(s) < x < H_W(s+1)).$$

Hence the complement of W is also semidecidable and we are done by lemma 9.2.2. □

Recall that the graph of a partial function $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ is the set of all tuples $\{\, (\boldsymbol{x}, y) \mid f(\boldsymbol{x}) \simeq y \,\} \subseteq \mathbb{N}^{n+1}$.

Lemma 9.2.4 *A partial function $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ is computable if, and only if, its graph is semidecidable. A total function $f : \mathbb{N}^n \to \mathbb{N}$ is computable if, and only if, its graph is decidable.*

Proof. Suppose f is computable and consider a pair $(\boldsymbol{x}, y)$. To semidecide membership in the graph, we attempt to compute $f(\boldsymbol{x})$. If the computation converges, we check the result against y. For the opposite direction, given an argument $\boldsymbol{x}$, we dovetail the semidecision method on $(\boldsymbol{x}, y)$ for all y. If one of these computations converges, we output the corresponding y.

For a total function f, note that $f(\boldsymbol{x}) \neq y \iff \exists\, y'\, (y \neq y' \land f(\boldsymbol{x}) = y')$. □

In computer science it is standard practice to define collections of objects by an enumerative process: there is a way to generate all the objects in question, one after the other. The whole process may well be infinite, but every step of it is computable. For example, we could enumerate all Boolean formulae in conjunctive normal form, or all syntactically correct C++ programs (whatever that means). This leads to the following definition.

Definition 9.2.3 *A set $W \subseteq \mathbb{N}^n$ is recursively enumerable (r.e.) if it is the range of a partial computable function $f : \mathbb{N} \nrightarrow \mathbb{N}^n$.*

It is not hard to see that one can always an enumerating function f with support an initial segment of $\mathbb{N}$: the support is $\{0, 1, \dots, n{-}1\}$ if A has cardinality n, and $\mathbb{N}$ otherwise.

Lemma 9.2.5 *A set is recursively enumerable if, and only if, it is semidecidable.*

Proof. First assume $W \subseteq \mathbb{N}^n$ is recursively enumerable, say, W is the range of $f : \mathbb{N} \nrightarrow \mathbb{N}^n$. To semidecide W, given x, dovetail all computations $f(n)$ and check if any of them returns x.

For the opposite direction, suppose W is the support of $g : \mathbb{N}^n \nrightarrow \mathbb{N}$. Set a counter c to zero, and dovetail the computations $g(x)$. Whenever one of these computations converges, set $f(c) := x$ and increment c. □

We can use the enumeration theorem for computable functions to enumerate all recursively enumerable sets based on the equivalence of the last lemma. For simplicity, consider only subsets of $\mathbb{N}$:

$$W_e := \mathsf{spt}\,\{e\}$$

Also let $W_{e,\sigma} = \mathsf{spt}\{e\}_\sigma$ be the approximation to W_e at stage σ. We have $\sigma \leq \tau$ implies $W_{e,\sigma} \subseteq W_{e,\tau} \subseteq \{0, \dots, \tau - 1\}$ and $W_e = \bigcup_\sigma W_{e,\sigma}$.

We refer to e as an index for the set W_e. Note that any semidecidable set W has infinitely many indices: any register machine that computes the semi-characteristic function of W can be modified in a way that does not affect the output. For example, we could first increment and then decrement a register some number of times.

Lemma 9.2.6 *The semidecidable sets are closed under union and intersection. More precisely, the semidecidable sets form an effective lattice $\mathcal{E}$.*

Proof. For closure under intersection, we can simply run the two semidecision procedures sequentially: only if both return Yes is the input accepted.

For union, however, we need to interleave the two computations, which interleaving can formally be handled by using stages: $U_\sigma = W_\sigma \cup W'_\sigma$. □

The last lemma opens the possibility to study semidecidable sets from an algebraic perspective: what are the properties of the lattice $\mathcal{E}$?

Suppose we have a semidecidable set $W \subseteq \mathbb{N}^{n+1}$. W need not be single-valued in the sense that $(\boldsymbol{x}, y), (\boldsymbol{x}, y) \in W$ implies $y = y'$. Hence W may not be the graph of a computable function. But we can thin out W a bit so as to obtain a computable function, without losing any possible inputs.

Definition 9.2.4 *Let $R \subseteq \mathbb{N}^{n+1}$. Define the support of R to be $\mathsf{spt}R := \{\, \boldsymbol{x} \mid \exists\, y\, R(\boldsymbol{x}, y) \,\}$. A function $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ uniformizes R iff $\mathsf{spt}f = \mathsf{spt}R$ and $\forall\, x \in \mathsf{spt}f \; \big(R(x, f(x))\big)$.*

Uniformization by brute force in general drives up the level of complexity: the function $f(x) \simeq \min\bigl(y \mid R(x,y)\bigr)$ may well fail to be partial recursive even if R is recursively enumerable. To see this, consider the relation $R(x,y) \iff y = 1 \lor (y = 0 \land x \in K)$. Then f is the characteristic function of K and therefore not recursive. But we can uniformize a recursively enumerable relation by a partial recursive function if we use stages.

Theorem 9.2.1 (Uniformization Theorem) *Let $W \subseteq \mathbb{N}^{n+1}$ be a recursively enumerable relation. Then there exists a partial recursive function f that uniformizes W.*

Proof. Let W be the recursively enumerable relation. We can use the standard stages mechanism to define $f(\boldsymbol{x}) \simeq (\min\bigl(\sigma \mid (\boldsymbol{x}, (\sigma)_0) \in W_\sigma\bigr))_0$. Note that σ such that $(\sigma)_0 = y$ can be arbitrarily large, and our claim follows. □

9.2.1 Diagonalization and Undecidability

Let us briefly return to diagonalization in our abstract setting. In the context of set theory, Cantor's argument immediately implies that there are undecidable problems: there are $2^{\aleph_0}$ subsets of $\mathbb{N}$, but only countably many such subsets can be decidable. So, in a sense, almost all problems are undecidable. While entirely correct, this argument is less that satisfying in that it fails to produce a concrete example of an undecidable problem. By considering an effective version of diagonalization we can show that the Halting set

$$K = \{\, e \mid \{e\}(e)\downarrow \,\} = \{\, e \mid e \in W_e \,\}.$$

is undecidable, and, using reductions as in the next section, establish the undecidability of many concrete problems such as Diophantine equations or the word problem in group theory. Here is the basic proof again.

Lemma 9.2.7 *The Halting Set K is semidecidable, but fails to be decidable.*

Proof. To see that K is semidecidable define the approximation $K_\sigma = \{\, e < \sigma \mid \{e\}_\sigma(e)\downarrow \,\}$. Clearly, $K = \bigcup K_\sigma$ and thus semidecidable. Now assume for the sake of a contradiction that K is decidable. Then the following definition-by-cases will produce a total recursive function:

$$f(x) \simeq \begin{cases} \{x\}(x) + 1 & \text{if } x \in K, \\ 0 & \text{otherwise.} \end{cases}$$

But then f has an index e, $f = \{e\}$. If $e \in K$, we have $f(e) \simeq \{e\}(e) + 1 \simeq f(e) + 1$, a contradiction. On the other hand, if $e \notin K$, we have $\{e\}(e) \simeq f(e) \simeq 0$, again a contradiction. □

Another great source of undecidability are questions about computable functions that have the same input/output behavior. A set $I \subseteq \mathbb{N}$ is an index set if $\forall\, e, e' \, (\{e\} \simeq \{e'\} \land e \in I$ implies $e' \in I)$. Here are some interesting index sets that arise naturally when one tries to classify recursively enumerable sets:

$$\begin{aligned} \mathsf{FIN} &= \{\, e \in \mathbb{N} \mid W_e \text{ is finite} \,\} \\ \mathsf{INF} &= \{\, e \in \mathbb{N} \mid W_e \text{ is infinite} \,\} \\ \mathsf{TOT} &= \{\, e \in \mathbb{N} \mid W_e = \mathbb{N} \,\} \\ \mathsf{REC} &= \{\, e \in \mathbb{N} \mid W_e \text{ is decidable } \,\} \end{aligned}$$

By the next theorem, all these classes fail to be decidable. See section 9.3.1 for much better descriptions of the complexity of these index sets. In fact, all index set are undecidable, except in trivial cases.

Another way to express the same idea it to say that $P \subseteq \mathbb{N}$ is a non-trivial property of semidecidable sets if

- $W_e = W_{e'}$ implies $e \in P \iff e' \in P$,
- $e_0 \in P$ and $e_1 \notin P$ for some e_0 and e_1.

Rice's theorem below can then be interpreted as stating that every non-trivial property of semidecidable sets is undecidable.

Theorem 9.2.2 (Rice's Theorem) *Let I be a decidable index set. Then I is necessarily trivial: either $I = \mathbb{N}$ or $I = \emptyset$.*

Proof. Let e_0 be an index for the function that diverges everywhere. We may assume for the sake of a contradiction that $e_0 \notin I$ and $e_1 \in I$ for some e_1, otherwise we can consider $\mathbb{N} - I$. Define the partial function F by

$$F(e,x) \simeq \begin{cases} \{e_1\} & \text{if } e \in K, \\ \uparrow & \text{otherwise.} \end{cases}$$

Then F is a partial computable function. Let $\widehat{F}$ be an index for F, and set $f(e) = \mathsf{S}^1_1(\widehat{F}, e)$, so $\{f(e)\}(x) \simeq F(e,x)$. Thus $\{f(e)\} = \{e_1\}$ iff $e \in K$ and $\{f(e)\} = \{e_0\}$ otherwise. Hence $e \in K \iff f(e) \in I$. But then $\mathsf{char}_K = f \circ \mathsf{char}_I$ and I cannot be decidable, contradiction. □

The argument just given is the classical one. As it turns out, we can also exploit the recursion theorem to obtain a concise proof of Rice's theorem. To this end, assume P is decidable and define program Q by

```
input $x$
if $q \in P$
then return $\{e_1\}(x)$
else return $\{e_0\}(x)$
```

But then $q \in P$ implies $Q \simeq \{e_1\}$ and thus $q \notin P$. On the other hand, $q \notin P$ implies $Q \simeq \{e_0\}$ and thus $q \in P$, contradiction.

9.2.2 Exercises

Exercise 9.2.1 Compare the proofs of Rice's theorem.

9.3 Completeness

9.3.1 Reductions and Degrees

A standard technique in the design of algorithms is to reduce one problem to another, and bring to bear methods already known that can solve the second problem. For example, the problem of finding a matching in a bipartite graph can be reduced to solving a network flow problem. This affords a classification of sorts: if a problem can be reduced to another, it is presumably easier in some sense.

To make this idea precise, we use an idea first proposed by Turing. In modern terms, an oracle algorithm behaves like an ordinary algorithm, except that at certain places during the computation it may request information from an oracle. The oracle has the ability to compute a function Φ, perhaps in mysterious ways: given a query item z, the oracle determines some value $\Phi(z)$ free of computational charge. For our purposes, the most important case arises when the oracle is given by a set $B \subseteq \mathbb{N}$, in which case the oracle is capable of computing the characteristic function of B.

Convention In order to avoid notational clutter, from now on we will often not distinguish between a set and its characteristic function. Thus, writing $B(x) = 1$ simply means that x is a member of B.

The idea of an oracle was first introduced by Turing in connection with Turing machines. Somewhat surprisingly, it has caused a lot of confusion in some circles [**?**]. Thus it is worth while to take a closer look at how oracles might be introduced in a particular model of computation. For example, given some arithmetic function f, in the register machine model we could add one additional instruction

- `oracle`

with the following semantics: the contents r of register R_0 are passed to the oracle. The oracle then replaces r by $\Phi(r)$ in register R_0, and the computation continues with the next instruction. Alternatively, define the clone of functions partial recursive in Φ to be the least class of functions that

- contains Φ,
- contains constant zero, the successor function, and
- is closed under primitive recursion and unbounded search.

In essence, we simply pretend that Φ is one of the basic functions.

Note that this definition has two important consequences:

- Finite Support Principle: any convergent computation using oracle Φ uses only finitely many calls to Φ.
- Continuity Principle: if some, possibly finite, approximation $\Phi' \sqsubseteq \Phi$ already produces convergence, then Φ will do the same, with the same value.

See section 8.5.3 for another application of these principles.

It should be clear that as long as Φ itself is computable, the use of Φ as an oracle will not enlarge the class of computable functions (though it may greatly affect efficiency). This corresponds nicely to the notion of reduction of one computational problem to another. However, while undecidable oracles cannot appear in a practical algorithms, they are still conceptually useful since they allow one to pinpoint crucial ingredients in a computation. Moreover, they provide a mechanism to compare the relative difficulty of problems. For the time being, we will only deal with set oracles.

Definition 9.3.1 (Turing) *$A \subseteq \mathbb{N}$ is Turing reducible to $B \subseteq \mathbb{N}$ if A can be computed using B as an oracle. In symbols, $A \leq_T B$.*

It is obvious from the definitions that $\leq_T$ is reflexive. A little argument shows that this relation is also transitive, so Turing reducibility is a quasi-order. Note that complementation does not affect Turing reducibility, in either one of the two sets involved.

Proposition 9.3.1

$A \leq_T B$ if, and only if, $\mathbb{N} - A \leq_T B$.

$A \leq_T B$ if, and only if, $A \leq_T \mathbb{N} - B$.

Two mutually reducible sets contain the same amount of information in some sense and can be considered to be equivalent.

Definition 9.3.2 *Two sets A and B are Turing equivalent, whenever $A \leq_T B$ and $B \leq_T A$. This defines an equivalence relation $\equiv_T$ whose equivalence classes are called Turing degrees.*

Turing degrees are also called degrees of unsolvability since they measure the distance between a problem and solvability. We write $\deg_T(A)$ for the Turing degree of A. Note that we can extend the quasi-order $\leq_T$ to a partial order on Turing degrees. The order on these degrees is a measure of the difficulty of the set: the higher up in the order, the more difficult.

At the bottom level of this classification are the decidable sets: $A \leq_T B$ where B is decidable implies that A is also decidable. Hence

$$\deg_T(\emptyset) = \text{ decidable sets.}$$

But the degree of the Halting problem is distinct

$$\deg_T(\emptyset) <_T \deg_T(K)$$

meaning that every set in the first degree is Turing reducible to every set in the second degree, but not conversely.

Yet higher degrees are obtained by looking at yet more complicated decision problems. A good source for such problems are index sets, which must be undecidable by Rice's theorem. Recall the following basic index sets:

$$\begin{aligned}
\mathsf{FIN} &= \{\, e \in \mathbb{N} \mid W_e \text{ is finite} \,\} \\
\mathsf{INF} &= \{\, e \in \mathbb{N} \mid W_e \text{ is infinite} \,\} \\
\mathsf{TOT} &= \{\, e \in \mathbb{N} \mid W_e = \mathbb{N} \,\} \\
\mathsf{REC} &= \{\, e \in \mathbb{N} \mid W_e \text{ is decidable } \,\}
\end{aligned}$$

We claim that

$$\deg_T(\emptyset) < \deg_T(K) < \deg_T(\mathsf{FIN}) < \deg_T(\mathsf{REC})$$

and also

$$\deg_T(\mathsf{FIN}) = \deg_T(\mathsf{INF}) = \deg_T(\mathsf{TOT})$$

But note that the degree of K, a semidecidable set, contains sets that fail to semidecidable. The problem is that by the very definition of Turing reducibility we have $\mathbb{N} - A \leq_T A$, so that the complement of K is trivially in $\deg_T(K)$. This indicates that Turing reducibility is a bit too powerful in the realm of semidecidable sets. We will need to weaken our notion of reducibility to deal with this issue. Finding the appropriate notion of reduction will be critical in the context of complexity theory. Here are less powerful reductions that are better behaved.

Definition 9.3.3 *A is many-one reducible to B if there exists a recursive function $f : \mathbb{N} \to \mathbb{N}$ such that*

$$x \in A \iff f(x) \in B.$$

If function f is in addition injective then A is one-one reducible to B.
In symbols: $A \leq_m B$ and $A \leq_1 B$.

In other words: we can only ask a single question of the oracle, and whatever answer the oracle returns is also our answer. One-one reductions in addition are required to ask different questions to the oracle for different inputs. This may seem like a strange constraint, but it is often easy to achieve. The following observation is clear from the definitions.

Proposition 9.3.2

$$A \leq_1 B \textit{ implies } A \leq_m B \textit{ implies } A \leq_T B$$

The opposite implications are all false, but it requires a bit of effort to separate many-one and one-one reductions.

Proposition 9.3.3 *Let $\leq$ be one of the reductions $\leq_m$ or $\leq_1$.*

- *$A \leq_m B$ and B decidable implies that A is decidable.*
- *$A \leq_m B$ and B semidecidable implies that A is semidecidable.*

Proof. This is easy to see from the Σ_1 and Δ_1 definitions of semidecidable and decidable sets. □

The key application of reductions is to provide lower bounds: $A \leq_m B$ means that B is at least as complicated as A. Note, though, that in order for this to be true in the informal sense, we have to make sure that the reduction function itself is not too complicated. In the realm of general computability this is not too much of an issue, but can become problematic in complexity theory.

Lemma 9.3.1 *$K \leq_m \mathsf{INF}$ and $K \leq_m \mathsf{TOT}$*

Proof. There is a recursive function f such that

$$\{f(e)\}(x) \simeq \begin{cases} 0 & \text{if } e \in K, \\ \uparrow & \text{otherwise.} \end{cases}$$

But then $K \leq_m \mathsf{TOT}, \mathsf{INF}$.

□

In particular neither INF nor TOT can be decidable, a fact we already know from Rice's theorem. As before with Turing reductions one can collect mutually reducible sets into a degree. To this end let $A \equiv_m B$ if $A \leq_m B \wedge B \leq_m A$ and let $A \equiv_1 B$ if $A \leq_1 B \wedge B \leq_1 A$. Clearly, $\equiv_m$ and $\equiv_1$ are equivalence relations and the corresponding equivalence classes are called many-one degrees and one-one degrees, respectively. Clearly, many-one and one-one degrees provide a finer partition than Turing degrees.

Proposition 9.3.4 *$A \leq_m B, B$ r.e. implies A r.e..*

The most complicated recursively enumerable set we know so far is K, the Halting Set. What is its position in the partial order for many-one or one-one degrees? One might suspect that K is fairly high up. In fact, it (more precisely: its degree) might be a largest element in the order. This is captured in the next definition.

Definition 9.3.4 *A set $C \subseteq \mathbb{N}$ is Turing (many-one, one-one) complete if C is r.e. and for all A r.e.: $A \leq_T C$ ($A \leq_m C$, $A \leq_1 C$).*

Note that it is not entirely clear whether a complete set exists: in a sense, a complete set has to contain information about all recursively enumerable sets. It is straightforward to construct such a set in terms of pure set theory (a so-called hard set), but we need to keep the set itself recursively enumerable.

Lemma 9.3.2 *The Halting Set K is one-one complete.*

Proof. There is an elementary function f such that

$$\forall\, z\, \{f(e,x)\}(z) \downarrow \iff \{e\}(x) \downarrow$$

But then $x \in W_e \iff f(e,x) \in K$. We can make sure the f is also injective by exploiting the fact that every computable function has infinitely many indices (first count to e). □

One can define variants of K that are easily seen to lie in the same one-one degree–and are thus really the same as K from the point of view of complexity.

$$K_0 = \{\, \langle e, x\rangle \mid x \in W_e \,\}$$
$$K_1 = \{\, e \mid W_e \neq \emptyset \,\}$$

Here $\langle e, x\rangle$ is one of our standard pairing functions.

Proposition 9.3.5 *K, K_0 and K_1 are all one-one equivalent.*

Proof. The one-one equivalence of K and K_0 is taken care of in the last lemma. For K and K_1 essentially the same function works. □

One-one equivalent sets are also very similar from the perspective of the lattice $\mathcal{E}$: there is an automorphism that transports one to the other.

Definition 9.3.5 *Two sets $A, B \subseteq \mathbb{N}$ are recursively isomorphic if there is a recursive permutation p of $\mathbb{N}$ such that $p[A] = B$. In symbols: $A \equiv B$.*

There is an analogue to the classical Cantor-Schröder-Bernstein theorem in set theory that associates the existence of computable injections in both directions with the existence of a computable bijection. As the next result shows, the one-one degrees are simply obtained by applying a recursive permutation to the given set. Thus K, K_0 and K_1 can all be obtained from each other by recursive permutations of $\mathbb{N}$.

Theorem 9.3.1 (Myhill's Isomorphism Theorem) *For $A, B \subseteq \mathbb{N}$:*

$$A \equiv B \iff A \equiv_1 B.$$

Proof. Suppose $A \leq_1 B$ via f and $B \leq_1 A$ via g. Define a new function h in stages using a zig-zag construction: $h = \bigcup h_\sigma$ where h_σ is finite.

Stage $\sigma = 0$: h_0 is completely undefined.

Stage $\sigma > 0$, even:
Assume that h_σ is injective and $\forall\, x \in \mathsf{spt} h_\sigma\, (x \in A \iff h_\sigma(x) \in B)$. We define h on $x = (\sigma+1)/2$ as follows.

If $h_\sigma(x) \downarrow$ do nothing, otherwise compute in sequence $f(x), f \circ h_\sigma^{-1} \circ f(x), f \circ (h_\sigma^{-1} \circ f)^2(x), \ldots$ As f and h_σ are injective, there can be no repetitions. But then for some i: $y = f \circ (h_\sigma^{-1} \circ f)^i(x) \notin \mathsf{rng}\, h_\sigma$ since h_σ is finite. Set $h_{\sigma+1}(x) := y$.

$\sigma > 0$, odd: Similar, exchange f, h_σ by g, h_σ^{-1}.

It is clear that the function h is computable. □

We know that decidable sets are easy to recognize in $\mathcal{E}$. How about complete sets? More narrowly, what is the rôle of K in this lattice? By Myhill's theorem, K, K_0 and K_1 are all images of each other under automorphisms of the lattice. So we are looking for structural properties of K in terms of the lattice $\mathcal{E}$, not the fact that K has a definition that involves universal computation and the like. This turns out to be a very difficult question leading to open problems, but we still can get a little mileage out it.

K is not complemented in $\mathcal{E}$. So for any recursively enumerable set W we have

$$W \subseteq \overline{K} \quad \text{implies} \quad \exists\, x\, (x \in \overline{K} - W).$$

This property is definable over $\mathcal{E}$ and motivates the next, stronger definition (which abandons $\mathcal{E}$).

Definition 9.3.6 *A set P is productive if there is a partial recursive function p such that $W_e \subseteq P \Rightarrow p(e) \in P - W_e$.*

So p computes a witness for the fact that $W_e \neq P$. Note that $\overline{K}$ is productive with trivial witness function $f(x) = x$. Clearly, no recursively enumerable set with a productive complement can be recursive. Here is a definition due to E. Post who suggested that membership questions for recursively enumerable sets inevitably require some clever thinking.

> The conclusion is inescapable that even for such a fixed, well-defined body of mathematical propositions, mathematical thinking is, and must remain, essentially creative.

Definition 9.3.7 (E. Post, 1944) *A r.e. set C is creative if its complement is productive.*

Be that as it may, creative sets are crucial to the understanding of of one-one completeness. First, we can slightly sharpen the notion of a productive function.

Lemma 9.3.3 *Let P be productive. Then there is a productive function p for P that is total and injective.*

Proof. Let q be an arbitrary productive function for P. In the first step transform q into a total function as follows. Define h by

$$W_{h(x)} = \begin{cases} W_x & \text{if } q(x)\downarrow, \\ \emptyset & \text{otherwise.} \end{cases}$$

Then define q' by

$$q'(x) \simeq \begin{cases} q(x) & \text{if } q(x)\downarrow \text{ before } q(h(x)), \\ q(h(x)) & \text{otherwise.} \end{cases}$$

Note that $q(x)$ or $q(h(x))$ must converge since $q(x)\uparrow \Rightarrow W_{h(x)} = \emptyset \subseteq P$. Hence q' is total. Now let $W_{g(x)} = W_x \cup \{q'(x)\}$ and define $p(0) = q'(0)$.

For $x > 0$, determine $p(x)$ by enumerating in sequence $q'(x)$, $q'(g(x))$, $q'(g^2(x)) \dots$ If a repetition occurs, i.e., $q'(g^i(x)) = q'(g^j(x))$, then W_x is not a subset of P and we may set $p(x) = \min\big(z \mid z \notin \{p(0), \ldots, p(x-1)\big)$. Otherwise, for some i we have $q'(g^i(x)) \notin \{p(0), \ldots, p(x-1)\}$ and we may set $p(x) = q'(g^i(x))$ where i is minimal such.

□

So we may safely assume that the witness function of any productive set is total and injective. This will be used in the next theorem.

Theorem 9.3.2 *If P is productive, then $\mathbb{N} - K \leq_1 P$. Hence any creative set is $\leq_1$-complete.*

Proof. Let p be a total injective productive function for P. Define f recursive by

$$W_{f(x,e)} \begin{cases} \{p(x)\} & \text{if } e \in K, \\ \emptyset & \text{otherwise.} \end{cases}$$

By the recursion theorem, there is a recursive injective function ω such that $W_{\omega(e)} = W_{f(\omega(e),e)}$.

To see this let $F(e, e_0, z) = \{f(e, e_0)\}(z)$. By the recursion theorem for any fixed e_0 there exists some e^* such that $F(e^*, e_0, -) = \{f(e^*, e_0)\} = \{e^*\}$. In fact, our proof shows that e^* can be computed from e_0, i.e., $\omega(e_0) = e^*$ is a recursive function and furthermore injective.

Hence $W_{\omega(e)} = \{p(\omega(x))\}$ if $e \in K$ and $W_{\omega(e)} = \emptyset$ otherwise. Now suppose $e \in K$. Then $W_{\omega(e)} \not\subseteq P$ because otherwise the witness constructed by p would be in $W_{\omega(e)}$. Hence $p(\omega(x)) \notin P$.

On the other hand if $e \notin K$ then $\emptyset = W_{\omega(e)} \subseteq P$, whence $p(\omega(x)) \in P$. Therefore $e \in K \Leftrightarrow p(\omega(x)) \notin P$. □

Corollary 9.3.1 *Let $C \subseteq \mathbb{N}$ be recursively enumerable. The following are equivalent:*

1. *C is many-one-complete,*
2. *C is one-one-complete,*
3. *C is creative.*

9.3.2 The Turing Jump

It is a labor of love to check that all our definitions are quite robust with respect to the addition of an oracle: all the basic results are the with or without an oracle. For example, a universal register machine turns into a universal register machine plus oracle. In particular our enumeration results generalize without any effort, and we can write

$$\begin{array}{rl} \{e\}^A & e\text{th function computable with } A \\ W_e^A = \mathsf{dom}\{e\}^A & e\text{th r.e. set with } A \end{array}$$

So we have the notion of a set being recursively enumerable in A, recursive in A, a function being computable in A, and so on. How can we describe these computations with oracle A a bit more precisely? Recall that we often identify a set $A \subseteq \mathbb{N}$ with its characteristic function. In the same spirit, for any n, write $A \restriction n$ for the following finite approximation to the characteristic function:

$$(A \restriction n)(z) \simeq \begin{cases} A(z) & \text{if } z < n, \\ \uparrow & \text{otherwise.} \end{cases}$$

Computations relative to A can be expressed in terms of approximations $A \restriction n$ for some n and we write $\alpha \sqsubset A$ to mean that $\alpha = A \restriction n$ for some n. By the finite support principle,

$$\{e\}^A(x) \simeq y \iff \exists\, \alpha \sqsubset A \left(\{e\}^\alpha(x) \simeq y\right)$$

with the understanding that the computation on the right never asks the oracle α any questions outside of its domain. This rules out immediately any divergent computation that uses the oracle infinitely often. The standard Halting Set is none other than the Turing jump of $\emptyset$, so it is natural to generalize to arbitrary oracles.

Definition 9.3.8 *Let $A \subseteq \mathbb{N}$. The (Turing) jump of A is defined as*

$$A' = K^A = \{\, e \mid \{e\}^A(e) \downarrow \}$$

So $\emptyset' = K^\emptyset = K$. But note when can iterate the jump to obtain presumably more complicated sets. What are the basic properties of the jump?

Lemma 9.3.4 *The jump A' is recursively enumerable in A but not Turing reducible to A.*

Proof. This is a verbatim repetition of the oracle-free argument. □

Lemma 9.3.5 *A is recursively enumerable in B if, and only if, $A \leq_1 B'$.*

Proof. Suppose A is r.e. in B. Hence there is a primitive recursive function f such that

$$\{f(x)\}^B(z) \simeq \begin{cases} 0 & \text{if } x \in A, \\ \uparrow & \text{otherwise.} \end{cases}$$

This function is B-computable since we can replace A on the right by W_e^B. But then $x \in A \iff f(x) \in B'$. As usual, we can force f to be injective by choosing a different index for each x.

Now suppose $A \leq_1 B'$, say, $x \in A \iff f(x) \in B'$. To enumerate A given B as oracle, proceed in stages. At stage σ, compute $f(x)$ for all $x = 0, 1, \ldots, \sigma - 1$. Enumerate B'_σ using oracle B. If any of the $f(x)$ appear in B'_s, enumerate the corresponding x's into A. □

Theorem 9.3.3 *$A \leq_T B$ if, and only if, $A' \leq_1 B'$.*

In particular, A and B are Turing equivalent if, and only if, A' and B' are one-one-equivalent.

Proof. A' is r.e. in $A \leq_T B$, so A' is r.e. in B and we are through by lemma 9.3.5.

Now assume $A' \leq_1 B'$, say, $e \in A' \iff f(e) \in B'$. There are elementary functions g_1, g_2 such that

$$\{g_1(e)\}^A(z) \simeq \begin{cases} 0 & \text{if } e \in A, \\ \uparrow & \text{otherwise.} \end{cases}$$

$$\{g_2(e)\}^A(z) \simeq \begin{cases} 0 & \text{if } e \notin A, \\ \uparrow & \text{otherwise.} \end{cases}$$

Now we combine f and the g_i to show how oracle B allows us to determine membership in A.

$$\begin{aligned} e \in A &\iff g_1(e) \in A' \\ &\iff f(g_1(e)) \in B' \\ &\iff \{f(g_1(e))\}^B(f(g_1(e)))\downarrow \\ &\iff \exists\, \beta \sqsubset B\, \{f(g_1(e))\}^\beta(f(g_1(e)))\downarrow \end{aligned}$$

Likewise

$$e \notin A \iff \exists\, \beta \sqsubset B\, \{f(g_2(e))\}^\beta(f(g_2(e)))\downarrow$$

Since one of the two computations must converge, we can decide A which with oracle B. □

9.3.3 Exercises

Exercise 9.3.1 Prove that the index sets FIN, INF and TOT are all Turing-equivalent.

Exercise 9.3.2 Verify that the construction in the maximal set theorem really defines a r.e. set.

Exercise 9.3.3 Give a proof of corollary 9.3.1.

9.4 Definability and Incompleteness

9.4.1 The Arithmetical Hierarchy

The Turing jump shows that there are many levels of undecidability. One may wonder whether there are ways to describe complicated sets that do not use the machinery of universality and enumeration that is built into the definition of the jump. As it turns out, we can construct sets of high complexity also by using purely set-theoretic operations, which, at first glance, appear to have nothing much to do with computability.

Definition 9.4.1 *Let $A \subseteq \mathbb{N}^{1+n}$ be a relation. The projection of A is the set*

$$\mathsf{proj}(A) = \{\, \boldsymbol{x} \in \mathbb{N}^n \mid \exists\, z\, (z, \boldsymbol{x}) \in A \,\} \subseteq \mathbb{N}^n.$$

The complement of $A \subseteq \mathbb{N}^n$ is always understood to be $\mathbb{N}^n - A$. For any collection $\mathcal{C}$ of subsets of $\mathbb{N}^n$, define $\mathsf{proj}(\mathcal{C})$ to be the collection of all projections of elements in $\mathcal{C}$. Likewise, define $\mathsf{compl}(\mathcal{C})$ to be the collection of all complements of sets in $\mathcal{C}$.

From a logical perspective, we can think of a projection as an unbounded search: to establish $\boldsymbol{x} \in \mathsf{proj}(A)$ we have to find a witness z such that $(z, \boldsymbol{x}) \in A$. Hence, by the Kleene Normal Form theorem, every semidecidable set is a projection of a decidable set. Complementation of a semidecidable set produces a co-semidecidable set, which, as we have seen, may well fail to be semidecidable. What happens if we keep projecting and taking complements?

Here is a typical example. Define the relation $R \subseteq \mathbb{N}^3$ by

$$R(s, t, e) \iff \{e\}(t) \text{ halts after at most } s \text{ steps.}$$

We know that R is elementary and thus in particular decidable. If we project, complement, project and complement again we get

$$\mathsf{TOT} = \{\, e \in \mathbb{N} \mid \{e\} \text{ is a total function, } \}$$

the collection of all (indices of) total recursive functions. Rewritten with arithmetic quantifiers, we have

$$e \in \mathsf{TOT} \iff \forall\, t\, \exists\, s\, R(s, t, e).$$

Thus, to decide membership in TOT we would have to conduct an unbounded search for each value of t, a kind of nested infinite computation. We already know that TOT is neither semidecidable nor co-semidecidable.

Definition 9.4.2 (Arithmetical Hierarchy) *Define classes of subsets of $\mathbb{N}^n$:*

$$\begin{aligned} \Sigma_0 = \Pi_0 &= \textit{ all decidable sets} \\ \Sigma_{k+1} &= \mathsf{proj}(\Pi_k) \\ \Pi_k &= \mathsf{compl}(\Sigma_k) \\ \Delta_k &= \Sigma_k \cap \Pi_k \end{aligned}$$

A set $A \subseteq \mathbb{N}^n$ is arithmetical if it belongs to some class Σ_k.

Thus, Δ_1 is the class of all decidable sets, Σ_1 is the class of all semidecidable sets, and Π_1 is the class of all co-semidecidable sets.

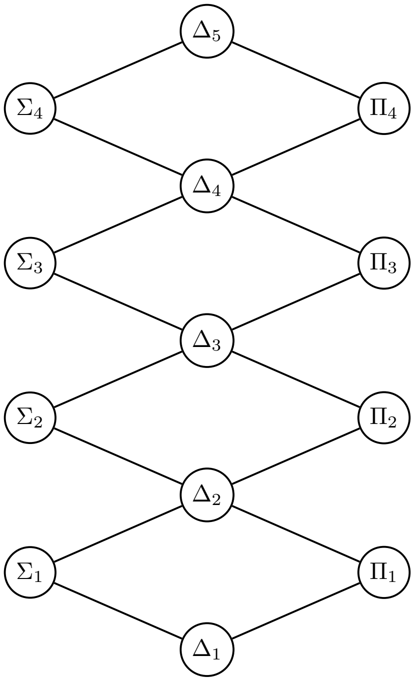


Figure 9.1: The first few levels of the arithmetical hierarchy.

One can show that this hierarchy is proper as suggested in figure 9.1.

Theorem 9.4.1 (Hierarchy Theorem)

$$\Sigma_k, \Pi_k \subsetneqq \Delta_{k+1} \subsetneqq \Sigma_{k+1}, \Pi_{k+1}.$$

The argument is quite similar to the one used to show that the Halting Problem is semidecidable (at level Σ_1 in the hierarchy) but not decidable (at level Δ_1 in the hierarchy). At any rate, we now have a classification problem: given an arithmetical set, determine where in the hierarchy it occurs. Note that the class of arithmetical sets is certainly countable, so it follows by Cantor-style counting argument that there are lots of subsets of $\mathbb{N}$ that fail to be arithmetical. Following Hilbert's idea to investigate the Entscheidungsproblem, it is natural to consider arithmetical truth:

$$\mathsf{Th}(\mathfrak{N}) = \{ \varphi \text{ sentence} \mid \mathfrak{N} \models \varphi \}$$

As it turns out, $\mathsf{Th}(\mathfrak{N})$ is not an element in this hierarchy: truth in arithmetic has no arithmetical definition. Intuitively, the problem is that that to handle the truth of Σ_k sentences one already needs a Σ_k definition. So slices of $\mathsf{Th}(\mathfrak{N})$ are naturally located in the hierarchy, but the whole truth set is not.

Of practical relevance are mostly the first few levels of the arithmetical hierarchy. More pointedly: there is no natural example of a $\Sigma_{17} - \Delta_{17}$ set. Similar effects also exist in other hierarchies: a natural polynomial time decidable set has complexity $O(n^k)$ for rather small values of k, say, $k = 10$. It is currently not well understood why this should be the case. By quantifier counting we can easily see that TOT is Π_2, FIN is Σ_2, COF and REC are Σ_3, and COMP, the collection of all complete r.e. sets, is Σ_4. For example, for REC consider an r.e. set W_e.

$$\begin{aligned} e \in \mathsf{REC} &\iff \exists\, e' \,(W_e = \Sigma^\star - W_{e'}) \\ &\iff \exists\, e' \,\forall\, x, \sigma \,\exists\, \tau \,(W_{e,\sigma} \cap W_{e',\sigma} = \emptyset \wedge x \in W_{e,\tau} \cup W_{e',\tau}) \end{aligned}$$

Counting quantifiers, this presentation shows that REC appears at level Σ_3 of the hierarchy. As a practical matter, if one is careful about avoiding unnecessary quantifiers, one usually obtains the correct result by quantifier counting. None of these sets belong to the next lower Δ_k level of the hierarchy, but that requires a separate hardness argument that is usually more complicated.

For example, it is easy to see that FIN is in Σ_2.

Proposition 9.4.1 FIN *is Σ_2-complete.*

Proof. Membership is just quantifier counting. To establish Σ_2-completeness, we show that $A \leq_m \mathsf{FIN}$ for a generic $A \in \Sigma_2$. There is a decidable relation R such that

$$x \notin A \iff \forall\, s \,\exists\, t \, R(s,t,x)$$

Then there is a computable function f such that

$$\{f(x)\}(u) \simeq \begin{cases} 0 & \text{if } \forall\, s < u \,\exists\, t \, R(s,t,x), \\ \uparrow & \text{otherwise.} \end{cases}$$

Now consider $\varphi \equiv \forall\, s < u \,\exists\, t \, R(s,t,x)$. Note that $\varphi \iff \exists\, T \,\forall\, s < u \,\exists\, t < T \, R(s,t,x)$, so we really need only one unbounded search. Now if $x \in A$, then by construction $W_{f(x)}$ is finite. But then $f(x) \in \mathsf{FIN}$. On the other hand, if $x \notin A$, then $W_{f(x)} = \Sigma^\star$, and in particular infinite. But then $f(x) \notin \mathsf{FIN}$. Hence $x \in A \iff f(x) \in \mathsf{FIN}$ and we are done. □

Proposition 9.4.2 $\mathsf{INF} \equiv \mathsf{TOT}$.

Proof. To show $\mathsf{INF} \leq_m \mathsf{TOT}$, assume we are given e: we construct a r.e. set $U = W_{f(e)}$ in stages.

```
**Stage** $\sigma$:
    Put all $x < \max W_{e,\sigma}$ into $U$.
```

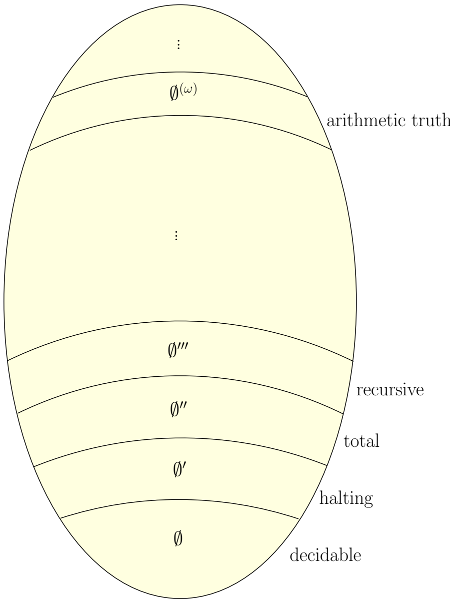
Since $W_{e,\sigma}$ is a finite subset of σ, this is perfectly effective. Hence we have $e \in \mathsf{INF} \Leftrightarrow f(e) \in \mathsf{TOT}$, and with the usual trick we get $\mathsf{INF} \leq_1 \mathsf{TOT}$.

For $\mathsf{TOT} \leq_m \mathsf{INF}$, first recall that we write $I_x = \{\, z \in \Sigma^\star \mid z < x \,\}$ for the initial segment determined by x. Given e, we construct a r.e. set $U = W_{f(e)}$ in stages.

```
**Stage** $\sigma$:
    **if** $I_x \subseteq W_{e,\sigma}$
    **then** put all $z < x$ into $U$.
```

Again, this is all perfectly effective. We conclude that $e \in \mathsf{TOT} \Leftrightarrow f(e) \in \mathsf{INF}$, and with the usual trick we get $\mathsf{TOT} \leq_1 \mathsf{INF}$. Together, we have $\mathsf{INF} \equiv \mathsf{TOT}$. □

The bottom part of the computational universe can be visualized like so:



Of course, the top level should is nowhere near drawn to scale.

9.4.2 Definability

In the last section, we have tacitly used definitions of relations over the structure of the natural numbers

$$\mathfrak{N} = \langle \mathbb{N}, +, *, 0, 1; < \rangle$$

where we have two binary operations addition and multiplication, constants zero and one, and the less-than relation. All these basic operations are computable, and even elementary, so it is natural to ask how other computable functions can be represented over $\mathfrak{N}$. We think of $\mathfrak{N}$ as a first-order structure and assume the usual language $\mathcal{L}(+, *, 0, 1; <)$ of arithmetic. So the question is: how can computable functions be defined in terms of first-order formulae over this structure, and what is the complexity of these definitions? As usual, we write $\underline{n}$ for the numeral representing the natural number n and use them as constants over $\mathfrak{N}$.

Definition 9.4.3 *A first-order formula $\varphi(\boldsymbol{x})$ defines a subset $\varphi^{\mathfrak{N}} \subseteq \mathbb{N}^n$ as follows:*

$$\varphi^{\mathfrak{N}} = \{\, \boldsymbol{x} \in \mathbb{N}^n \mid \mathfrak{N} \models \varphi(\underline{\boldsymbol{x}}) \,\}$$

A relation R is said to be first-order definable if there is some formula φ such that $R = \varphi^{\mathfrak{N}}$. Definability for a function $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ means that the graph of the function is definable.

Formulae are syntactic objects, and thus provide various measure of complexity. As in the last section, the measure we are interested in here is quantifier complexity. At the bottom of the hierarchy, we have formulae that have no full existential or universal quantifiers.

Definition 9.4.4 *Let t be a term not involving the variable x. A quantifier of the form $\exists\, x < t\, \varphi$ is interpreted to abbreviate $\exists\, x\, (x < t \wedge \varphi)$ and is called a bounded existential quantifier. Similarly, a quantifier of the form $\forall\, x < t\, \varphi$ is interpreted to abbreviate $\forall\, x\, (x < t \Rightarrow \varphi)$ and is called a bounded universal quantifier.*

A formula φ that contains only bounded quantifiers is said to be Δ_0. Functions and relations that admit a definition using a Δ_0 formula are also referred to as Δ_0.

Note that our definition is a bit sloppy, we should be more careful about which variables may occur in the bounding term t; see the exercises. As an example, primality is Δ_0:

$$\begin{aligned} x \text{ is prime} \quad &\Longleftrightarrow \quad x \geq 2 \wedge \forall\, z < x\, (z \mid x \Rightarrow z = 1) \\ x \mid y \quad &\Longleftrightarrow \quad \exists\, z \leq y\, (x * z = y) \end{aligned}$$

It should be clear that every Δ_0 relation is easily decidable: we can cope with all the bounded quantifiers by bounded search. In fact, we do not need the full power of computability.

Lemma 9.4.1 *Every Δ_0 relation is primitive recursive.*

Proof. By structural induction on the defining formula, using the closure properties of primitive recursive relations established in section 8.1.5. □

We note in passing that the converse implication fails: there are primitive recursive relations that are not Δ_0, but the proof is somewhat complicated. But how about general recursive functions? Are they definable, and what is the complexity of a defining formula, should one exist?

Definition 9.4.5 *A formula $\varphi(\boldsymbol{x})$ is Σ_1 if there is a Δ_0 formula ψ such that $\varphi(\boldsymbol{x}) = \exists\, \boldsymbol{y}\, \psi(\boldsymbol{x}, \boldsymbol{y})$.*

A relation or function is called Σ_1 if can be defined by a Σ_1 formula.

Unbounded existential quantifiers over $\mathfrak{N}$ correspond to unbounded search, so one might conjecture that computable functions and relations are all Σ_1. As a first step in this direction, we will show that $\varSigma_1$ certainly captures primitive recursive functions. Consider a $\varSigma_1$ formula

$$\varphi(\boldsymbol{x}) = \exists\, y_1 \exists\, y_2 \ldots \exists\, y_n\, \psi(\boldsymbol{x}, \boldsymbol{y})$$

where ψ is $\varDelta_0$. We can rewrite φ equivalently as

$$\exists\, z \exists\, y_1 < z \ldots \exists\, y_n < z\ \psi(\boldsymbol{x}, \boldsymbol{y})$$

so we may safely assume that there is only one unbounded quantifier.

Lemma 9.4.2 *Let $\varphi(z, \boldsymbol{x})$ be a $\varSigma_1$ formula. Then $\forall\, z < t\ \varphi(z, \boldsymbol{x})$ is equivalent to a $\varSigma_1$ formula.*

Proof. We may assume that $\varphi(z, \boldsymbol{x}) = \exists\, u\, \psi(u, z, \boldsymbol{x})$ with ψ $\varDelta_0$. We can use sequence numbers to push the bounded universal quantifier past the unbounded existential one:

$$\forall\, z < t\ \exists\, u\, \psi(u, z, \boldsymbol{x}) \iff \exists\, s\, \big(s \in \mathsf{Seq} \land \mathsf{len}(s) = t \land \forall\, z < t\, \psi((s)_z, z, \boldsymbol{x})\big)$$

One can check that all the requisite coding machinery can be expressed in a $\varDelta_0$ fashion. □

According to the lemma, up to equivalence, $\varSigma_1$ formulae are closed under all propositional connectives, bounded quantifiers and existential quantifiers. As a consequence, all primitive recursive functions are $\varSigma_1$-definable.

Lemma 9.4.3 *Every primitive recursive function is $\varSigma_1$-definable.*

Proof. We use induction on the definition of $f : \mathbb{N}^n \to \mathbb{N}$. The atomic cases are all obvious. For closure under composition, suppose $f = h \circ \boldsymbol{g}$. There are $\varSigma_1$ formulae ψ_h and ψ_{g_i} that define h and the g_i, respectively. Then

$$\exists\, z_1 \ldots \exists\, z_m\, \big(\psi_{g_1}(\boldsymbol{x}, z_1) \land \ldots \land \psi_{g_m}(\boldsymbol{x}, z_m) \land \psi_h(\boldsymbol{z}, y)\big)$$

is a $\varSigma_1$ definition of f. It remains to establish closure under primitive recursion. Let

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

Given arguments x and $\boldsymbol{y}$, we can think of f as being computed bottom-up, starting at $f(0, \boldsymbol{y}) = g(\boldsymbol{y})$, then $f(1, \boldsymbol{y}) = h(0, g(\boldsymbol{y}), \boldsymbol{y})$ and so on, all the way up to $f(x, \boldsymbol{y})$. This process can be captured using sequence numbers to keep track of intermediate values and we obtain the following definition of f:

$$\exists\, s \in \mathsf{Seq}\, \big(\mathsf{len}(s) = x \land (s)_0 = g(\boldsymbol{y}) \land \forall\, z < x\, ((s)_{z^+} = h(z, (s)_z, \boldsymbol{y}))\big)$$

Replacing g and h by their $\varSigma_1$ definitions, and using our closure properties, this produces a $\varSigma_1$ definition of f. □

Recall the characterization of partial recursive functions in Kleene's normal form theorem: we need one unbounded existential quantifier over a primitive recursive matrix. This produces the following corollary.

Corollary 9.4.1 *A function is partial recursive if, and only if, it is $\varSigma_1$-definable over $\mathfrak{N}$. Similarly a relation R is semidecidable if, and only if, it is $\varSigma_1$-definable over $\mathfrak{N}$.*

A total function is recursive function if, and only if, it is $\varDelta_1$-definable over $\mathfrak{N}$. A relation R is decidable if, and only if, it is $\varDelta_1$-definable over $\mathfrak{N}$.

The corollary provides a surprising powerful method to verify the computability of functions, first-order logic is quite expressive and arguably a more accommodating environment than machines, programs or equations. How about the remainder of the arithmetical hierarchy? Projections correspond directly to existential quantification, and complementation corresponds to logical not, so the whole hierarchy is just another way to organize definable sets.

Proposition 9.4.3 *The arithmetical sets are exactly the sets first-order definable over* $\mathfrak{N}$.

Since the sets at level Σ_k are precisely the sets definable by Σ_k-formulae, there is no harm in using the same notation for both contexts (set theory versus syntax).

9.4.3 Representability

In the last two sections, we have assumed that truth over $\mathfrak{N}$ is a concept that requires no further elaboration, some sentence of first-order logic simply holds or fails to hold. This position was popular up until the first part of the 19th century but started to crumble with the discovery of non-Euclidean geometries and assorted anomalies in analysis and set theory. A more modern approach is to consider truth relative to a particular structure, a model of the underlying theory. Unfortunately, the definition of these structures requires a relatively strong background theory such as ZFC and leaves the ontological status of the structures unsettled.

Alternatively, we can attempt to argue within a suitable formal system and effectively replace truth by provability. We will only consider axiomatizations $\mathcal{F}$ in first-order logic here. The question arises how much strength we need of a formal system $\mathcal{F}$ so we can reason about computable functions. The standard axiomatization of arithmetic was introduced by G. Peano in 1889, building on earlier work by Grassmann and Dedekind, and is now referred to as Peano Arithmetic (PA). In modern parlance, the main idea behind Peano Arithmetic is to use a Dedekind chain: we have a pointed set A with a special element $a_0 \in A$ and a function $s : A \to A$. The requirements are:

- the function s is injective, and a_0 is not in the range of of s;
- for any set $X \subseteq A$, if $a_0 \in X$ and X is closed under s, then $X = A$.

The intent is that a_0 should represent zero, and s the successor function. The second condition establishes induction: any property that holds of a_0 and is inherited by applying s must already hold everywhere. It is not hard to see that these conditions determine the natural numbers uniquely. Unfortunately, as stated they require second-order logic, which is a bit unwieldy (or, as Quine might say, not even a logic). We can translate things into first-order, and throw in addition and multiplication for good measure, as follows. Induction now is handled by an axiom schema, where the set X is replaced by an arbitrary formula $\varphi(x)$. It is this version that is now generally referred to as Peano Arithmetic.

successor	$S(x) \neq 0$	$S(x) = S(y) \Rightarrow x = y$
addition	$x + 0 = x$	$x + S(y) = S(x + y)$
multiplication	$x \cdot 0 = 0$	$x \cdot S(y) = (x \cdot y) + x$
order	$\neg(x < 0)$	$x < S(y) \Leftrightarrow x = y \lor x < y$
induction	$\varphi(0) \land \forall x \left(\varphi(x) \Rightarrow \varphi(S(x))\right) \Rightarrow \forall x \, \varphi(x)$	

A critical difference between this and the earlier version is that induction only holds for arithmetical sets, not all sets in general. As a consequence, there are non-isomorphic models, the standard one, plus unintended so-called non-standard models of arithmetic. It is a matter of experience that all proofs in elementary number theory can be naturally formalized in (PA). Even complicated results such as the prime number theorem, which is usually established using analytic means, can be reconstructed in (PA). It is currently unclear whether more complicated arguments such as Wiles's proof of the Fermat Conjecture require more powerful frameworks. We can now define what it means that a relation or function is expressible in a formal system such as (PA).

Definition 9.4.6 *A relation R is representable in $\mathcal{F}$ if for some formula φ*

$$\begin{array}{rll} R(\boldsymbol{x}) & \textit{implies} & \mathcal{F} \vdash \varphi(\underline{\boldsymbol{x}}) \\ \neg R(\boldsymbol{x}) & \textit{implies} & \mathcal{F} \vdash \neg\varphi(\underline{\boldsymbol{x}}) \end{array}$$

As always, $\underline{\boldsymbol{x}}$ stands for the vector of numerals representing $\boldsymbol{x} \in \mathbb{N}^n$. A function f is representable in $\mathcal{F}$ if for some formula φ

$$\begin{array}{rll} f(\boldsymbol{x}) = y & \textit{implies} & \mathcal{F} \vdash \varphi(\underline{\boldsymbol{x}}, \underline{y}) \\ f(\boldsymbol{x}) \neq y & \textit{implies} & \mathcal{F} \vdash \neg\varphi(\underline{\boldsymbol{x}}, \underline{y}) \end{array}$$

This is similar to the Herbrand-Gödel approach where $f(\boldsymbol{x}) \simeq y$ means that an equation $f(\underline{\boldsymbol{x}}) = \underline{y}$ had to be derivable using only substitution and replacement. Here the derivation can use the full power of first-order logic, not just restricted equational reasoning. A relation is representable iff its characteristic function is so representable, provided that $\mathcal{F}$ recognizes distinct numerals as such: $\mathcal{F} \vdash \underline{x} \neq \underline{y}$ for all $x \neq y$. This notion of representability is quite spartan, often one would like a stronger concept: the system should prove the functionality of f. More precisely, f is strongly representable in $\mathcal{F}$ if f is representable by for some formula φ and, in addition,

$$\mathcal{F} \vdash \varphi(\underline{\boldsymbol{x}}, y) \Leftrightarrow y = \underline{f(\boldsymbol{x})}$$

for all $\boldsymbol{x}$. Very minor requirements for $\mathcal{F}$ will guarantee that a representable function is already strongly representable.

Lemma 9.4.4 *Suppose that $\mathcal{F}$ is consistent and can prove the following:*

$$\begin{array}{l} \neg(x < \underline{0}) \\ x < \underline{n+1} \Rightarrow x = \underline{0} \vee x = \underline{1} \vee \ldots \vee x = \underline{n} \\ x < \underline{n} \vee x = \underline{n} \vee \underline{n} < x \end{array}$$

Then any function that is representable in $\mathcal{F}$ is already strongly representable.

Proof. Given a representation φ for f, consider

$$\psi(\boldsymbol{x}, y) \iff \phi(\boldsymbol{x}, y) \wedge \forall\, z < y\, \neg\varphi(\boldsymbol{x}, z)$$

Then ψ strongly represents f. □

It is easy to check that (PA) proves the three conditions in the lemma. We can now reexamine the definitions of, say, μ-recursive functions in terms of representability in (PA). A straightforward induction on the definition of such a function shows that every recursive function is (strongly) representable in (PA). A greater challenge is to show that representability translates into computability.

Theorem 9.4.2 *Exactly all recursive functions are representable in* (PA).

A satisfactory proof of this theorem requires a interesting technique called arithmetization: we need to translate the formal theory into an object of number theory. To this end, we need to code formulae in terms of natural numbers, and we have to make sure that all the relevant operations are easily computable. For example, we want to be able to say that such-and-such is a proof of some formula, using only arithmetic. Today, in the age of digital computers, this idea may seem fairly quaint; in 1931 when Gödel first proposed arithmetization it was a genuine breakthrough and one of the fundamental ideas behind his incompleteness theorem. To begin, we associate every formula φ with a number, the so-called Gödel number of φ, and traditionally written $\ulcorner\phi\urcorner$. Since formulae are simply strings over a finite alphabet, our elementary coding machinery from section 8.1.3 easily suffices to do so. Moreover, we can check that any kind of syntactic operation of formulae naturally translates into an elementary function on the corresponding Gödel numbers. For example, there is an elementary function Sub such that

$$\mathsf{Sub}(\ulcorner\varphi\urcorner, \ulcorner t\urcorner, \ulcorner x\urcorner)$$

is the Gödel number of the formula obtained by substituting term t for variable x in φ. If one of the arguments fails to be a suitable Gödel number, Sub returns 0. Similarly we can define an elementary predicate Pr such that

$$\mathsf{Pr}(\ulcorner\varphi\urcorner, \ulcorner\pi\urcorner)$$

holds whenever π is a proof of φ in $\mathcal{F}$. By proof we simply mean a finite sequence of formulae, $\pi = \psi_1, \psi_2, \ldots, \psi_m$, with appropriate conditions. We can set $\ulcorner\pi\urcorner = \langle\ulcorner\psi_1\urcorner, \ldots, \ulcorner\psi_m\urcorner\rangle$.

We can now establish the difficult part of theorem 9.4.2 as follows. Suppose f is represented in $\mathcal{F}$ by formula $\varphi(\boldsymbol{x}, y)$. To determine that $f(\boldsymbol{a}) = b$ for $\boldsymbol{a} \in \mathbb{N}^n$, $b \in \mathbb{N}$, we systematically enumerate all possible proofs π of $\varphi(\underline{\boldsymbol{a}}, \underline{b})$. Since we are dealing with Gödel numbers, this is just an unbounded search over $\mathbb{N}$. When we find a proof, we return the corresponding b.

At this point we can pinpoint a representability property that forces a formal system like (PA) to be undecidable. Fix a special variable x in our language once and for all, and define the operation

$$\varDelta(s) = \mathsf{Sub}(s, \underline{s}, x)$$

Thus $\varDelta(\ulcorner\varphi\urcorner)$ is the result of substituting the numeral $\underline{\ulcorner\varphi\urcorner}$ into formula φ for variable x. This is another example of self-reference, in a sense we are substituting (a code for) the formula into itself.

Theorem 9.4.3 *Let $\mathcal{T}$ be a consistent system of number theory for which theorem 9.4.2 holds and assume that $\varDelta$ is representable in $\mathcal{T}$. Then $\mathsf{Th}(\mathcal{T})$ is undecidable.*

Proof. Suppose $\varDelta$ is represented by $\varphi(x, y)$ where x is our special variable, and $\mathsf{Th}(\mathcal{T})$ is represented by $\psi(y)$. Consider the formula

$$\varPhi(x) = \forall\, y\, (\varphi(x, y) \Rightarrow \neg\psi(y))$$

which asserts roughly that the result of a $\varDelta$ substitution is not a theorem. Let $p = \ulcorner\phi\urcorner$ and set $q = \varDelta(p)$, so q is the Gödel number of $\phi(p)$. Since $\varDelta$ is represented in $\mathcal{T}$, we have $\mathcal{T} \vdash \varphi(p, q)$.

On the other hand, we must have $\mathcal{T} \vdash \neg\psi(\underline{q})$: assume either $\mathcal{T} \;/\!\vdash \varPhi(\underline{p})$ or $\mathcal{T} proves \varPhi(\underline{p})$, the conclusion is always valid.

But then $\mathcal{T} \vdash \varPhi(\underline{p})$, and $\mathcal{T} \vdash \psi(\underline{q})$. This contradicts consistency. □

So any axiomatization like (PA) where all recursive functions are representable is automatically undecidable; assuming, of course, that (PA) is indeed consistent. As always, it is a good idea to take a closer look and try to determine how much of a formal system is really needed in order for some argument to succeed. For example, what part of Peano Arithmetic is really required to handle recursive functions? As it turns out, much weaker systems suffice. We assume the usual axiomatization of identity, so that, say, $x = y \land \varphi(x) \Rightarrow \varphi(y)$ is provable.

Theorem 9.4.4 *Suppose that $\mathcal{F}$ is a consistent system containing function symbols $+$ and $\cdot$ and a relation symbol $<$. Assume that $\mathcal{F}$ has axiom schemata*

$$\begin{array}{ll}
\underline{x} \neq \underline{y} & \textit{provided that } x \neq y \\
\neg(x < \underline{0}) & \\
x < \underline{n+1} \Rightarrow x = \underline{0} \lor \ldots \lor x = \underline{n} & \\
x < \underline{n} \lor x = \underline{n} \lor \underline{n} < x & \\
\underline{x} + \underline{y} = \underline{x+y} & \\
\underline{x} \cdot \underline{y} = \underline{x \cdot y} &
\end{array}$$

Then any recursive function is already strongly representable in $\mathcal{F}$.

The system $\mathcal{F}$ fails to be finitely axiomatizable, but there are others that are finitely axiomatized and still represent all recursive functions. A famous example is R. Robinson's system Q, which removes induction from (PA) and replaces it by a much weaker axiom about the range of the successor function being all non-zero elements (the third axiom below):

$$\begin{array}{ll}
\underline{0} \neq S(x) & S(x) = S(y) \Rightarrow x = y \\
x \neq \underline{0} \Rightarrow \exists\, y\, x = S(y) & \\
x + \underline{0} = x & x + S(y) = S(x+y) \\
x \cdot \underline{0} = \underline{0} & x \cdot S(y) = x \cdot y + x
\end{array}$$

Note that induction is conspicuously absent from these axioms. As a consequence, none of the standard properties such as associativity or commutativity are derivable from them. In fact, one cannot even show that $S(x) \neq x$: just misinterpret the carrier to be all cardinals rather than just the finite ones. On the other hand, these axioms do suffice to prove equalities and inequalities between closed terms. Order can be defined by $x < y :\Leftrightarrow \exists\, z\, (x + S(z) = y)$. In fact, if one includes order as a primitive, one can give a quantifier-free version of Robinson's axioms. It is critical for undecidability, though, that both addition and multiplication are present: removing either one (yielding Presburger arithmetic or Skolem arithmetic, respectively) results in decidability.

9.4.4 Exercises

Exercise 9.4.1 Give a precise definition of a Δ_0 formula.

Exercise 9.4.2 Let I be the index set of all total recursive functions that are non-increasing. What is the complexity of I?

Exercise 9.4.3 Let $\mathsf{Comp} = \{\, e \in \mathbb{N} \mid W_i \text{ is complete} \,\}$ be the index set of all r.e. sets that are complete. What is the complexity of Comp?

9.5 Classical Recursion Theory

In this section we briefly mention a few traditional topics in classical recursion theory, in particular the lattice of semidecidable sets, and a few results about the structure of Turing degrees of semidecidable sets. It is entirely safe to skip this chapter unless you are interested in the historical aspects of the field, and some exposure to advanced techniques such as finite injury priority arguments. Of course, your intellectual development will be forever stunted if you do.

9.5.1 The Lattice $\mathcal{E}$

As Bourbaki has shown, it can be extremely fruitful to organize the study of mathematics in terms of *structures*, and in particular first-order structures. In the case of the collection of semidecidable sets this idea leads naturally to algebra, more precisely to lattice theory: semidecidable sets are closed under union and intersection,so we can think of this collection as an algebraic structure. As it turns out, we obtain a distributive lattice $\mathcal{E}$

$$\mathcal{E} = \langle \text{semidecidable}; \cup, \cap, 0, 1 \rangle.$$

In this setting, the decidable sets form a Boolean sublattice of $\mathcal{E}$, consisting of all the elements that have a complement, a definable property over $\mathcal{E}$:

$$A \text{ decidable} \iff \mathcal{E} \models \exists\, X\, (X \cap A = 0 \wedge X \cup A = 1).$$

Somewhat less obvious is that we can also define finiteness over $\mathcal{E}$:

$$A \text{ finite} \iff \mathcal{E} \models \forall\, X\, (X \subseteq A \Rightarrow X \text{ decidable }).$$

Also note that the finite sets form an ideal over $\mathcal{E}$. Since the lattice operations are defined by set-inclusion it is a natural first question to consider the subsets of an semidecidable set.

Lemma 9.5.1 *Let A be an infinite semidecidable set. Then A contains a semidecidable set that fails to be decidable. On the other hand, A also contains an infinite decidable subset.*

Proof. Suppose $f : \mathbb{N} \to A$ is a bijective enumeration of A and set $A_0 := f[K] \subseteq A$ where K is the Halting Set. Then $x \in K \iff f(x) \in A_0$, and A_0 cannot be decidable.

For the second claim, define $g(0)$ to be any element of A, and $g(n+1) := \min\big(x \mid f(x) > g(n)\big)$. Let A_0 be the range of g. Since g is monotonic, A_0 is decidable. □

Theorem 9.5.1 (Reduction Theorem) *Let A and B two semidecidable sets. Then there exist semidecidable sets $A' \subseteq A$ and $B' \subseteq B$ such that: $A' \cap B' = \emptyset$ and $A' \cup B' = A \cup B$.*

Proof. Let $(A_\sigma)_\sigma$, $(B_\sigma)_\sigma$ be non-decreasing approximations to A and B. Define $A' := \bigcup_{\sigma \geq 0} A_{\sigma+1} - B_\sigma$ and $B' := \bigcup_{\sigma \geq 0} B_\sigma - A_\sigma$. That is, at stage σ enumerate $A_{\sigma+1} - B_\sigma$ into A'. □

Theorem 9.5.2 (Separation Theorem) *Let A and B two disjoint co-semidecidable sets. Then there exists a decidable set R that separates A and B: $A \subseteq R$ and $R \cap B = \emptyset$.*

Proof. Apply the reduction theorem to $\mathbb{N} - A$ and $\mathbb{N} - B$, yielding A' and B'. Set $R := B'$. As $A' \cup B' = \mathbb{N} - A \cup \mathbb{N} - B = \mathbb{N} - (A \cap B) = \mathbb{N}$, and $A' \cap B' = \emptyset$ the set R must be decidable. $A \subseteq R$ as $x \in A \Rightarrow x \in B'$. Similarly $x \in B \Rightarrow x \in A' = \mathbb{N} - R$. □

The last two proofs are very much standard computability theory, but the motivation for the results comes directly from the study of semidecidable sets as a lattice. The next definition is similarly motivated.

Definition 9.5.1 *Two disjoint sets A and B are recursively inseparable if there is no decidable set that separates them.*

The last theorem shows that co-semidecidable sets are never recursively inseparable. But we can construct semidecidable sets that are recursively inseparable using the enumeration theorem:

$$A := \{ e \mid \{e\}(e) \simeq 0 \}$$
$$B := \{ e \mid \{e\}(e) \simeq 1 \}$$

A and B are obviously semidecidable and disjoint. Now suppose C separates them: $A \subseteq C$ and $C \cap B = \emptyset$. If C is semidecidable it has some index e. But then $e \in C$ implies $\{e\}(e) = C(e) = 1$, so $e \in B$ and we have a contradiction. On the other hand, $e \notin C$ implies $\{e\}(e) = C(e) = 0$, so $e \in A \subseteq C$ and again there is a contradiction.

Overall, the structure of $\mathcal{E}$ is rather complicated: the first-order theory is undecidable, though Π_2 sentences are decidable.

Simple Sets

From the perspective of the lattice $\mathcal{E}$, a way to ensure that a semidecidable set is not decidable, is to make sure that its complement is not semidecidable. Here is a strong assertion that ensures the complement cannot be semidecidable.

Definition 9.5.2 *A set $A \subseteq \mathbb{N}$ is immune if A is infinite but any semidecidable subset of A is already finite. A semidecidable set $S \subseteq \mathbb{N}$ is simple if its complement $\overline{S}$ is immune.*

Hence a semidecidable set S with infinite complement is simple iff

$$\forall W \in \mathcal{E} \, (W \text{ infinite } \Rightarrow S \cap W \neq \emptyset)$$

Just like creative sets, simple sets are clearly not decidable. However, they are somewhat more difficult to construct; in particular the Halting Set is not simple.

As a warm-up exercise for the construction of a simple set, let us disregard the constraint that S has to be semidecidable. Then we can construct a set S such that $\overline{S}$ is immune in stages $S = \bigcup S_\sigma$ as follows. At state 0, set $S_0 := \emptyset$. For all stages $\sigma > 0$, check if W_e is infinite and $W_e \cap S_{<\sigma} = \emptyset$. If so, pick x in W_e, $x \geq 2\sigma$, and add x to S_σ. To see that $\mathbb{N} - S$ is infinite, notice that $|S \cap [0, 2n]| \leq n + 1$. Hence, the complement of S is immune.

However, the condition "W_e is infinite" is not decidable, and in fact not even semidecidable, so S is too complicated for our purposes. Here is an argument that fixes this problem. Arguably, there are more direct constructions, but this one introduces a number of key ideas that help to solve much more complicated problems.

We retain the idea of constructing S in stages. For the construction to succeed, we need to satisfy a number of requirements, in this case the requirements

$$(P_e): \qquad S \cap W_e \neq \emptyset$$

Note that there are infinitely many requirements, we will need to figure out a way to address them all. Since our construction needs to be effective, we can only work on finitely many requirements at any particular time during the construction. In the end, though, all requirements will be met.

At stage σ one cannot tell whether (P_e) is satisfied, however, we can test an effective version of the requirement:

$$S_{<\sigma} \cap W_{e,\sigma} \neq \emptyset$$

Here $S_{<\sigma}$ is the part of S constructed prior to stage σ, with the understanding that $S_{<0} = \emptyset$. Note that if (P_e) is satisfied at stage σ, we are done with (P_e) forever! We need some bookkeeping machinery to decide which requirement to work on at any particular stage. Let us say that requirement e has higher priority than $\bar{e}$ if $e < \bar{e}$. Requirement e requires attention at stage σ if

- $S_{<\sigma} \cap W_{e,\sigma} = \emptyset$,
- $\exists\, x \in W_{e,\sigma}(2e \leq x)$.

We can then try to place x into S in attempt to satisfy the requirement. This process is referred to as the requirement receiving attention. Overall we wind up with the following construction:

Stage σ

Let (P_e) be the requirement of highest priority that requires attention at stage σ.
Let x be minimal with $x \in W_{e,\sigma} \wedge 2e \leq x$ and put x into S_σ.

If no such requirement exists, do nothing.

Claim: All requirements (P_e) are eventually satisfied.

For a proof, we proceed by induction on e. We may safely assume that W_e is infinite, otherwise (P_e) is trivially satisfied. For all $i < e$ pick a stage σ_i such that (P_i) is satisfied at stage σ_i. Let $\sigma = \max \sigma_i + 1$. Then at some stage $\tau \geq \sigma$, requirement (P_e) will require attention. It will also receive attention, as no requirements of higher priority are active at time τ. Thus we have the following theorem.

Theorem 9.5.3 *There exists a simple set.*

Maximal Sets

Here is another property of semidecidable sets that is far from obvious, and, arguably, somewhat counterintuitive. Write $A \subseteq^\star B$ to mean that $A - B$ is finite.

Definition 9.5.3 *A set $C \subseteq \mathbb{N}$ is cohesive if C is infinite but for all semidecidable W, $C \subseteq^\star W$ or $C \subseteq^\star \overline{W}$. A semidecidable set M is maximal if $\mathbb{N} - M$ is cohesive.*

Thus a cohesive set is infinite, but cannot be split by any semidecidable set into two infinite parts. Another way to express this is to insist that for all semidecidable sets W

$$C \cap W \text{ is finite or } C - W \text{ is finite.}$$

A maximal set is a co-atom in the quotient lattice $\mathcal{E}/\mathsf{FIN}$, and it is far from clear that such an object should exist.

Again, as a warm-up exercise, let us construct a cohesive set ignoring the effectiveness constraint for maximal sets. We will use a collection (c_e) of markers that will indicate the elements in $\overline{C}$. Clearly, at least one of W_0 or $\mathbb{N} - W_0$ must be infinite, say, W_0 is infinite. Define a working set D to be W_0, and place marker c_0 in W. Now one of $W \cap W_0$ or $W - W_1$ must be infinite, say, $W - W_1$ is infinite. Set D to $W_0 - W_1$ and place marker $c_1 > c_0$ in W (so D is a difference of r.e. sets). In the end, let $C := \{\, c_i \mid i \geq 0 \,\}$. Then C is indeed cohesive: for all e, we have by construction $\{ c_i \mid i \geq e \} \subseteq W_e$ or $\{ c_i \mid i \geq e \} \cap W_e = \emptyset$. Of course, this construction is far from effective.

Let us rephrase this construction in a way that is analogous to the simple set construction from above. This time, there are positive and negative requirements to contend with: the first kind tries to place elements into M, the second kind tries to keep them out.

$$\begin{array}{ll} (P_e) & \overline{M} \subseteq^\star W_e \text{ or } \overline{M} \subseteq^\star \overline{W_e} \\ (N_e) & \overline{M} \text{ has cardinality at least } e \end{array}$$

Obviously these requirements clash and we have to define a protocol that resolves these clashes and makes sure that ultimately all requirements are satisfied. In order to keep track of membership in r.e. sets, we use the following device. To keep notation manageable, we will use a standard convention in what follows: for any set $A \subseteq \mathbb{N}^n$, we write A for the characteristic function of A. Thus $A(\boldsymbol{x}) = 1$ means that $\boldsymbol{x} \in A$. Now define the e-state of x at stage σ, as

$$s(e, x, \sigma) := (W_{0,\sigma}(x), \ldots, W_{e,\sigma}(x)) \in \{0, 1\}^e$$

Thus $s(e, x, \sigma)$ is a bit-vector of length $e+1$ indicating membership in the first $e+1$ semidecidable sets, with the enumeration truncated to level σ. Clearly, s is primitive recursive and the limit $s(e, x) = \lim_{\sigma \to \infty} s(e, x, \sigma)$ always exists. If we order these bit-vectors lexicographically, then $\sigma \leq \tau$ implies that $s(e, x, \sigma) \leq s(e, x, \tau)$. Hence $s(e, x, \sigma)$ can change value only finitely often.

The strategy for an effective construction of M is now this: try to maximize the e-state of the eth element of $\overline{M}$, using markers as above. For the construction we initialize with $M_0 = \emptyset$ and $c_{e,0} = e$.

Stage σ:

For any positive stage σ, find the least $e < e' < \sigma$ such that $s(e, z_{e',<\sigma}, {<}\sigma) > s(e, z_{e,<\sigma}, {<}\sigma)$.

Set $z_{e,\sigma} := z_{e',<\sigma}$ and move all the markers $z_{i,\sigma}$ forward by one for $e' < i$.

Moving the markers should be interpreted as placing the elements $z_{e,<\sigma}$, $z_{e+1,<\sigma}$, $\ldots$, $z_{e'-1,<\sigma}$ into M_σ. As stated, the construction is infinitary, but it is possible to modify the description so that at level σ only numbers strictly less than σ are handled. The modified construction is then primitive recursive in σ.

We need to show that M has infinite complement. It suffices to show that $\lim_{\sigma \to \infty} c_{e,\sigma}$ exists. By induction, wait, until at state σ, all the witnesses $c_{i,\sigma}$ for $i < e$ have settled down. But then, at any later stage τ, $c_{e,\tau}$ can only move to increase its e-state. That can happen only finitely often.

To see that the positive requirements are also satisfied, assume by induction that everything is well for $i < e$, but (P_e) fails; thus $\overline{M} \cap W_e$ and $\overline{M} \cap \overline{W_e}$ are both infinite. Since there are only finitely many bit-vectors of length e we may choose $n \geq e$ such that for all $m \geq n$: $s(e-1, z_m) = s(e-1, z_n)$. Pick $n < i < j$ such that $z_i \notin W_e$ but $z_j \in W_e$. Wait till some stage σ when the markers for z_i and z_j have converged to their final positions. But then $s(e, z_i) > s(e, z_j)$ and marker z_i will move after stage σ, a contradiction. □

9.5.2 Intermediate Degrees

It is a peculiar empirical fact that most problems known to be semidecidable turn out to be either decidable or complete in the end, though pinning down the complexity of any particular problem may be quite difficult. Perhaps the best example is Hilbert's 10th problem, where the completeness proof required some seven decades of work. At the other end of the spectrum, Presburger's proof of the decidability of additive arithmetic is quite intricate. Note that this is a strictly empirical observation; we have not encountered any result that would indicate that there are no semidecidable Turing degrees other than $\emptyset$ and $\emptyset'$. In this section we will show that there are in fact other degrees, the so-called intermediate degrees. Alas, the proofs leave a bitter aftertaste: the sets constructed there are utterly artificial, they have no counterpart in the world of natural computation (whatever that may be). In particular, no example is known of a semidecidable set that has been studied independently in some other branch of mathematics (say, number theory) that turns out to be of intermediate degree.

Another natural question is whether Turing reducibility is a linear order: is it always true that, given two semidecidable sets A and B, either A contains at least as much information as B or the other way around? Using the jump operation, we can generate a sequence of sets of strictly increasing Turing degree. In fact, we can extend this operation to transfinite levels:

$$\emptyset <_T \emptyset' <_T \emptyset'' <_T \emptyset^{(3)} <_T \ldots <_T \emptyset^{(\omega)} <_T \emptyset^{(\omega+1)} <_T \ldots$$

Here $\emptyset^{(\omega)} = \{ \langle n, x \rangle \mid x \in \emptyset^{(n)} \}$ is the disjoint union of all the finite jumps $\emptyset^{(n)}$, $\emptyset^{(\omega+1)} = (\emptyset^{(\omega)})'$ and so forth. Thus $\emptyset^{(n)} <_T \emptyset^{(\omega)}$ for all $n \geq 0$. While this construction produces sets of enormous complexity it only gives us a chain of increasing Turing degrees, all the sets so obtained are comparable.

In this section we will show that the actual picture is quite a bit more complicated: there are incomparable degrees, i.e., there are sets A and B such that neither can be used as an oracle to compute the other. Moreover, one can construct such sets of very low complexity: both can be made to be semidecidable. The first proof below gives a weaker result but is less complicated and outlines the technique used in the main argument. Note that this also settles the question of whether there are any intermediate degrees: we must have $\emptyset <_T A <_T \emptyset'$.

The question whether there are any intermediate c.e. degrees was first posed by E. Post in 1944 in a seminal paper. No progress was made for a decade till two people found the solution almost simultaneously: R. M. Friedberg, then an undergraduate, in the US and A. A. Muchnik in Russia. They published their results in 1957 and 1956, respectively.

All the constructions of c.e. sets here are based on enumeration in stages: to define set A we explain, for each stage σ, which elements are to be placed into the set at stage σ. The mechanism that determines these elements can be quite complicated but it is clearly decidable whether x enters the set at stage σ. Hence $A_{<\sigma}$, the part of A constructed up to and excluding stage σ is decidable (uniformly in σ) and the whole set $A = \bigcup A_\sigma$ is c.e. The reason the constructions become complicated is that we are trying to control the Turing degree of A in a careful manner by making sure that special constraints are met – and there are infinitely many such constraints.

The Friedberg/Muchnik Construction

Recall the Continuity Principle from section 8.5.1: if a computation $\{e\}^A(x)$ converges, then there is a finite approximation $\alpha \sqsubset A$ with domain $\{0, 1, \ldots, s-1\}$ such that $\{e\}^A(x) \simeq \{e\}^\alpha(x)$. We refer to s as the length of α, in symbols $\mathsf{len}(\alpha)$. For the construction below it will be convenient to have a notation for the size of the initial segment of A we have to know in order to compute the eth recursive function on input x with oracle A. Let

$$\mathsf{use}(e, x, A) \simeq \min\big(\mathsf{len}(\alpha) \mid \alpha \sqsubset A \wedge \{e\}^\alpha(x)\downarrow\big).$$

Thus use is is partial recursive in A and the corresponding A-recursive approximation is

$$\mathsf{use}(e, x, A, \sigma) \simeq \min\big(\mathsf{len}(\alpha) \mid \alpha \sqsubset A \wedge \{e\}^\alpha_\sigma(x)\downarrow\big).$$

As usual, $\lim_\sigma \mathsf{use}(e, x, A, \sigma) = \mathsf{use}(e, x, A)$.

Two Incomparable Sets below $\emptyset'$

We will now build two incomparable sets that narrowly miss being semidecidable. The two sets are constructed in stages $\sigma < \omega$. At each stage, simple rules determine which elements should be added to A or B. No elements are ever removed from either set. The construction rules are simple, but fail to be decidable, so the resulting sets are not quite semidecidable. During the construction, we have to satisfy the following requirements:

(R_e) $A \not\simeq \{e\}^B$
(R'_e) $B \not\simeq \{e\}^A$.

If all requirements are indeed satisfied, then neither set can be used to compute the other one. The principal problem in the construction is that we have to deal with infinitely many requirements, and the individual requirements may well clash with each other. For example, we may wish to add some element x to A at stage σ to make sure that requirement R_e is satisfied: adding x would cause $A(x) = 1 \neq 0 = \{e\}^B(x)$. But adding x to A might change the value of some computation $\{g\}^A(y)$, and thereby affect the requirement R'_e. We resolve these clashes by dealing with the requirements in order, and preserving computations by changing the oracles only outside of a fixed initial segment.

Theorem 9.5.4 *There exist two incomparable Turing degrees below $\emptyset'$: there are sets A, $B \leq_T \emptyset'$ such that neither $A \leq_T B$ nor $B \leq_T A$.*

Proof. To construct A and B in stages we will use finite functions of the form $A_\sigma, B_\sigma : \{0, \ldots, s-1\} \to \{0, 1\}$ as approximations to A and B: $A_\sigma \sqsubset A$, $B_\sigma \sqsubset B$ and $A = \lim A_\sigma$, $B = \lim B_\sigma$. We let $A_{<\sigma}$ be the part of A constructed prior to stage σ (and likewise for B). The construction proceeds in stages as follows.

Stage $\sigma = 0$: Let $A_0 = B_0 = \emptyset$.

Stage $\sigma = 2e > 0$:

We work on requirement (R_e): $A \neq \{e\}^B$. Let $n = \mathsf{len}(A_{<\sigma})$, $n' = \mathsf{len}(B_{<\sigma})$ be the lengths of the parts of the sets constructed so far. At stage σ, we will determine whether n and n' are placed into A and B, respectively. Our actions will depend on whether some finite extension of $B_{<\sigma}$ produces a computation of $\{e\}$ on n, the least number for which membership in A is as yet undetermined, with a Boolean value. Note that if no such extension exists, then the requirement is satisfied no matter whether we place n into A or not: $\{e\}^B$ cannot be a characteristic function. In this case we simply set $A_\sigma(n) = 0$, $B_\sigma(n') = 0$. So suppose that

$$\exists\, \alpha, t\ (n' < \mathsf{len}(\alpha) \wedge B_{<\sigma} \sqsubseteq \alpha \wedge \{e\}^\alpha_t(n) \in \mathbf{2})$$

Pick α and t minimal such and set $B_\sigma = \alpha$, $A_\sigma(n) = 1 - \{e\}^\alpha_t(n)$.

Stage $\sigma = 2e + 1$: Exchange A, B, (R_e) and (R'_e).

Note that by construction $A = \lim A_\sigma$, $B = \lim B_\sigma$ are indeed total, so that membership is settled for each natural number.

Claim: Every requirement is satisfied.

Let us consider only (R_e), the other case is entirely similar. Suppose for the sake of a contradiction that $A = \{e\}^B$. Note that the function $\{e\}^B$ is then necessarily total and Boolean valued. Then, at stage $\sigma = 2e$, the condition in the construction must have been satisfied. But then our construction makes sure that $\{e\}^B(n) = \{e\}^{B_\sigma}(n)$: the part of B below $\mathsf{len}(B_\sigma)$ will not be changed. It follows from continuity that

$$n \in A \iff 0 = \{e\}_t^{B_\sigma}(n) = \{e\}^B(n) = A(n) \iff n \notin A,$$

a contradiction.

Lastly note that to determine the existence of α and t requires no more than $\emptyset'$ as an oracle: we can find α and t by using unbounded search with a primitive recursive predicate. □

The two sets in the last construction fail to be semidecidable, but they are not terribly far removed from being r.e.

Lemma 9.5.2 *$A \leq_T \emptyset'$ if, and only if, A is Δ_2-definable.*

Proof. Suppose that $A = \{e\}^K$ for some index e where K is a complete semidecidable set. Then $x \in A \Rightarrow A(x) = 1 \Rightarrow \{e\}^K = 1 \Rightarrow \exists\, \sigma, \alpha \sqsubset K\, \big(\{e\}_\sigma^\alpha(x) \simeq 1\big)$. The predicate $\{e\}_\sigma^\alpha(x) \simeq 0$ is primitive recursive and therefore certainly Δ_1-definable. Now $\alpha \sqsubset K \iff \forall\, x < \mathsf{len}(\alpha)\ (x \in \alpha \Rightarrow x \in K) \wedge \forall\, x < \mathsf{len}(\alpha)\ (x \in K \Rightarrow x \in \alpha)$. Replacing K by a Σ_1 definition we see that $\alpha \sqsubset K$ is Δ_2-definable and so A is Σ_2-definable. But the same argument also holds for $\mathbb{N} - A$, thus A is Δ_2-definable.

On the other hand suppose A is Δ_2-definable, say $x \in A \Leftrightarrow \exists\, s\, \forall\, t\, \phi(x, s, t)$. Then there is a $\emptyset'$-computable function f such that

$$f(x, s) = \begin{cases} 0 & \text{if } \forall t \phi(x, s, t), \\ 1 & \text{otherwise.} \end{cases}$$

For $f(x, s) = 1 \iff \exists\, t\, \neg\phi(x, s, t)$ which is a r.e. question and thus trivial for oracle $\emptyset'$. But then A is r.e. in K: $A_\sigma = \{\, x \mid \exists\, s < \sigma\, (f_\sigma(x, s) = 0) \,\}$. □

Two Incomparable Semidecidable Sets

Now for the main goal in this section: the construction of incomparable sets that are still semidecidable. The construction is quite similar to the previous one, but this time we cannot afford to assume knowledge about the existence of the finite extension F in the last proof. Instead, at stage σ, we only consider computations of length at most σ using whatever part of the oracle has already been constructed: $\{e\}_\sigma^{B_{<\sigma}}(x)$ is all the information we have available. For small values of σ this computation will not converge, and we have no resulting value, but for sufficiently large values of σ we will obtain a value (provided the computation converges at all). Accordingly we can then either place x into A, or try to prevent x from entering A so as to guarantee $A(x) \neq \{e\}^B(x)$.

The crucial problem is that the value of the computation may well change since other requirements will place elements into B at later stages. But since we are trying to construct semidecidable sets, we cannot remove x from A in order to respond to a changed value: once an element has entered the set, it has to stay forever. The construction below resolves this conflict by linearly ordering the requirements and giving preference to requirements of higher priority. As it turns out, each requirement will ultimately be satisfied after finitely many steps, so that requirements of lower priority also have a chance to be satisfied.

Theorem 9.5.5 (Friedberg-Muchnik) *There exist two incomparable semidecidable Turing degrees: there are two r.e. sets A, B such that neither $A \leq_T B$ nor $B \leq_T A$.*

Proof. Again we try to satisfy the requirements

$$\begin{array}{rl} (R_{2e}) & A \neq \{e\}^B \\ (R_{2e+1}) & B \neq \{e\}^A. \end{array}$$

By symmetry, it suffices to consider the even-numbered requirements. To satisfy (R_{2e}) we will try to find a witness x such that $A(x) = 1$ but $\{e\}^B(x) \simeq 0$.

The case where $\{e\}_\sigma^{B_\sigma}(x) \not\simeq 0$ for all stages σ is easy: R_{2e} is satisfied as long as we keep x out of A.

Now suppose at some stage σ we find that $\{e\}_\sigma^{B_{<\sigma}}(x) \simeq 0$. Then we throw x into A_σ and try to preserve the part of B that is used in the computation of $\{e\}_\sigma^{B_{<\sigma}}(x) \simeq 0$ so as to make sure that the value of the computation does not change in the future. This is done by means of a restraint function:

$$r(2e, \sigma) = \mathsf{use}(e, x, \sigma; B_{<\sigma}) + 1$$

If we succeed in keeping elements from entering this part of B we are done since $A(x) = 1 \neq 0 = \{e\}_\tau^{B_\tau}(x) = \{e\}^B(x)$, for all $\tau \geq \sigma$.

Note that we have to find a way to work on all requirements simultaneously. We will say that (R_i) has higher priority than (R_j) iff $i < j$. Working on a requirement of higher priority may destroy a witness of a requirement of lower priority, hence we may have to change the witness on occasion. To this end we will pick a witness for (R_{2e}) in a special, reserved set of potential witnesses: $\mathbb{N}_e = \{ \langle x, e \rangle \mid x \geq 0 \}$. Let $x(e, \sigma)$ be the witness for (R_e) at (the end of) stage σ. Note that a witness $\langle x, e \rangle$ is put into A only in an attempt to satisfy R_{2e}.

(R_{2e}) is said to require attention at stage σ if

$$\{e\}_\sigma^{B_{<\sigma}}(x(2e, {<}\sigma)) \simeq 0 \text{ and } r(2e, {<}\sigma) = 0.$$

Construction

Stage 0: Let $A_0 = B_0 = \emptyset$, $x(e, 0) = \langle 0, e \rangle$, $r(e, \sigma) = 0$.

Stage $\sigma > 0$:
Pick the requirement of highest priority that requires attention, say, (R_{2e}). Set $x = x(2e, \sigma) = x(2e, {<}\sigma)$ and put x into A_σ, Define $r(2e, \sigma) = \mathsf{use}(e, x, B_{<\sigma}, \sigma)$. For $i < 2e$ set $x(i, \sigma) = x(i, {<}\sigma)$ and $r(i, \sigma) = r(i, {<}\sigma)$. For $i >$ 2e set $x(i, \sigma) = \min\big(x \in \mathbb{N}_i \mid x \notin A_\sigma \cup B_\sigma \wedge x > \max\big(r(j, \sigma) \mid j \leq 2e \big) \big)$ and $r(i, \sigma) = 0$.

If no such requirement exists do nothing.

The requirement (R_{2e}) as above is said to receive attention at stage σ. Note that the witnesses and restraints of all requirements of higher priority are preserved, but all requirements of lower priority are clobbered: a new potential witness is selected, and the restraint function is set to 0.

Claim: Every requirement receives attention at most finitely often. Furthermore, all requirements are eventually satisfied.

Proof. By induction on e'. By IH we may pick a stage σ such that no requirement (R_i), $i < e'$, receives attention at $\tau \geq \sigma$. Let us assume $e' = 2e$.

Case 1: R_{2e} never receives attention after stage σ.

Since all higher order requirements are already satisfied, that can happen only because R_{2e} never requires attention after σ. Then $x = x(2e, \sigma) = x(2e, \tau)$, $\tau \geq \sigma$, is not in A: since the sets $\mathbb{N}_i$ are all disjoint, only R_{2e} could put x into A, and only if R_{2e} received attention. But $\{e\}^{B_\tau}(x) \not\simeq 1$, done.

Case 2: R_{2e} receives attention at some stage $\tau \geq \sigma$.

Then $x(2e, \tau)$ is put into A and a restraint is established for B at the maximum number queried in $\{e\}_\tau^{B_{<\tau}}(x(2e, {<}\tau)) \simeq 0$. But no requirements of priority $i \leq$ 2e ever become active after τ, hence $x = x(2e, \tau) = x(2e, \tau')$, $\tau' \geq \tau$, and $\{e\}^{B_{\tau'}}(x) \simeq 0 \neq A(x) = 1$ for all $\tau' \geq \tau$. Hence R_{2e} is satisfied and we are done. □

The construction of the last theorem is called a finite-injury argument, since each of the requirements is violated at most finitely often (because a requirement of higher priority receives attention). There are analogous infinite-injury arguments that can be used to establish more complicated results. For example, there is a density theorem due G. Sacks that says that between any two semidecidable sets there is a third:

Theorem 9.5.6 *Let A and B semidecidable such that $A <_T B$. Then there exists semidecidable C such that $A <_T C <_T B$.*

Also, all intermediate c.e. sets are incomparable to some other c.e. set.

Theorem 9.5.7 *For all semidecidable A such that $\emptyset <_T A <_T \emptyset'$, there is another semidecidable set B such that A and B are incomparable.*

Priority arguments are undoubtedly correct, and provide for the existence of certain semidecidable sets. Alas, the sets constructed in this manner are supremely artificial, they do not correspond to "natural" problems. For example, we know that the problem of solving Diophantine equations is r.e.-complete. It follows from the existence of intermediate r.e. degrees that there is a class of Diophantine equations for which the existence of a solution is undecidable, yet not as difficult as the Halting Problem. But no such class is known that has a reasonable algebraic definition. In fact, as of the time of this writing, not a single problem is known that is well-established in its field and is discovered to be of intermediate difficulty later on. It stands to reason that Turing degrees are not a particularly good measure of complexity after all.

9.5.3 Exercises

Exercise 9.5.1 Verify that the construction in the maximal set theorem really defines a r.e. set.

Exercise 9.5.2 What would a translation from Turing machines to Herbrand-Gödel equations look like? How about the opposite direction?

Exercise 9.5.3 Let I be the set of all indices e such that $W_e \neq \emptyset$. Show that I is r.e.-complete.

Exercise 9.5.4 Show that TOT is Π_2 by constructing this set using projections and complementation.

Exercise 9.5.5 Show that COF is Σ_3 by constructing this set using projections and complementation.

Exercise 9.5.6 Determine the position of INF in the arithmetical hierarchy.

Exercise 9.5.7 Carefully check that COMP is indeed Σ_4.

Exercise 9.5.8 Let I be the index set of all total recursive functions that are non-decreasing. What is the complexity of I?