1 What is a Proof?

1.1 Propositions

Definition. A *proposition* is a statement (communication) that is either true or false.

For example, both of the following statements are propositions. The first is true, and the second is false.

Proposition 1.1.1. *2 + 3 = 5.*

Proposition 1.1.2. *1 + 1 = 3.*

Being true or false doesn't sound like much of a limitation, but it does exclude statements such as "Wherefore art thou Romeo?" and "Give me an A!" It also excludes statements whose truth varies with circumstance such as, "It's five o'clock," or "the stock market will rise tomorrow."

Unfortunately it is not always easy to decide if a proposition is true or false:

Proposition 1.1.3. *For every nonnegative integer, n, the value of $n^2 + n + 41$ is prime.*

(A *prime* is an integer greater than 1 that is not divisible by any other integer greater than 1. For example, 2, 3, 5, 7, 11, are the first five primes.) Let's try some numerical experimentation to check this proposition. Let

$$p(n) ::= n^2 + n + 41.^{1} \tag{1.1}$$

We begin with $p(0) = 41$, which is prime; then

$$p(1) = 43,\ p(2) = 47,\ p(3) = 53, \ldots,\ p(20) = 461$$

are each prime. Hmmm, starts to look like a plausible claim. In fact we can keep checking through $n = 39$ and confirm that $p(39) = 1601$ is prime.

But $p(40) = 40^2 + 40 + 41 = 41 \cdot 41$, which is not prime. So it's not true that the expression is prime *for all* nonnegative integers. In fact, it's not hard to show that *no* polynomial with integer coefficients can map all nonnegative numbers into

[1] The symbol ::= means "equal by definition." It's always ok simply to write "=" instead of ::=, but reminding the reader that an equality holds by definition can be helpful.

prime numbers, unless it's a constant (see Problem 1.17). But the real point of this example is to show that in general, you can't check a claim about an infinite set by checking a finite set of its elements, no matter how large the finite set.

By the way, propositions like this about *all* numbers or all items of some kind are so common that there is a special notation for them. With this notation, Proposition 1.1.3 would be

$$\forall n \in \mathbb{N}.\ p(n) \text{ is prime.} \tag{1.2}$$

Here the symbol $\forall$ is read "for all." The symbol $\mathbb{N}$ stands for the set of *nonnegative integers*: 0, 1, 2, 3, ... (ask your instructor for the complete list). The symbol "$\in$" is read as "is a member of," or "belongs to," or simply as "is in." The period after the $\mathbb{N}$ is just a separator between phrases.

Here are two even more extreme examples:

Proposition 1.1.4. *[Euler's Conjecture] The equation*

$$a^4 + b^4 + c^4 = d^4$$

has no solution when a, b, c, d are positive integers.

Euler (pronounced "oiler") conjectured this in 1769. But the proposition was proved false 218 years later by Noam Elkies at a liberal arts school up Mass Ave. The solution he found was $a = 95800, b = 217519, c = 414560, d = 422481$.

In logical notation, Euler's Conjecture could be written,

$$\forall a \in \mathbb{Z}^+ \, \forall b \in \mathbb{Z}^+ \, \forall c \in \mathbb{Z}^+ \, \forall d \in \mathbb{Z}^+ .\ a^4 + b^4 + c^4 \neq d^4.$$

Here, $\mathbb{Z}^+$ is a symbol for the positive integers. Strings of $\forall$'s like this are usually abbreviated for easier reading:

$$\forall a, b, c, d \in \mathbb{Z}^+ .\ a^4 + b^4 + c^4 \neq d^4.$$

Proposition 1.1.5. *$313(x^3 + y^3) = z^3$ has no solution when $x, y, z \in \mathbb{Z}^+$.*

This proposition is also false, but the smallest counterexample has more than 1000 digits!

It's worth mentioning a couple of further famous propositions whose proofs were sought for centuries before finally being discovered:

Proposition 1.1.6 (Four Color Theorem)**.** *Every map can be colored with 4 colors so that adjacent[2] regions have different colors.*

[2]Two regions are adjacent only when they share a boundary segment of positive length. They are not considered to be adjacent if their boundaries meet only at a few points.

Several incorrect proofs of this theorem have been published, including one that stood for 10 years in the late 19th century before its mistake was found. A laborious proof was finally found in 1976 by mathematicians Appel and Haken, who used a complex computer program to categorize the four-colorable maps. The program left a few thousand maps uncategorized, which were checked by hand by Haken and his assistants—among them his 15-year-old daughter.

There was reason to doubt whether this was a legitimate proof: the proof was too big to be checked without a computer. No one could guarantee that the computer calculated correctly, nor was anyone enthusiastic about exerting the effort to recheck the four-colorings of thousands of maps that were done by hand. Two decades later a mostly intelligible proof of the Four Color Theorem was found, though a computer is still needed to check four-colorability of several hundred special maps.[3]

Proposition 1.1.7 (Fermat's Last Theorem)**.** *There are no positive integers x, y, and z such that*

$$x^n + y^n = z^n$$

for some integer $n > 2$.

In a book he was reading around 1630, Fermat claimed to have a proof for this proposition, but not enough space in the margin to write it down. Over the years, the Theorem was proved to hold for all n up to 4,000,000, but we've seen that this shouldn't necessarily inspire confidence that it holds for *all* n. There is, after all, a clear resemblance between Fermat's Last Theorem and Euler's false Conjecture. Finally, in 1994, British mathematician Andrew Wiles gave a proof, after seven years of working in secrecy and isolation in his attic. His proof did not fit in any margin.[4]

Finally, let's mention another simply stated proposition whose truth remains unknown.

Proposition 1.1.8 (*Goldbach's Conjecture*)**.** *Every even integer greater than 2 is the sum of two primes.*

Goldbach's Conjecture dates back to 1742. It is known to hold for all numbers up to 10^{18}, but to this day, no one knows whether it's true or false.

[3]The story of the proof of the Four Color Theorem is told in a well-reviewed popular (non-technical) book: "Four Colors Suffice. How the Map Problem was Solved." *Robin Wilson*. Princeton Univ. Press, 2003, 276pp. ISBN 0-691-11533-8.

[4]In fact, Wiles' original proof was wrong, but he and several collaborators used his ideas to arrive at a correct proof a year later. This story is the subject of the popular book, *Fermat's Enigma* by Simon Singh, Walker & Company, November, 1997.

For a computer scientist, some of the most important things to prove are the correctness of programs and systems—whether a program or system does what it's supposed to. Programs are notoriously buggy, and there's a growing community of researchers and practitioners trying to find ways to prove program correctness. These efforts have been successful enough in the case of CPU chips that they are now routinely used by leading chip manufacturers to prove chip correctness and avoid mistakes like the notorious Intel division bug in the 1990's.

Developing mathematical methods to verify programs and systems remains an active research area. We'll illustrate some of these methods in Chapter 5.

1.2 Predicates

A *predicate* can be understood as a proposition whose truth depends on the value of one or more variables. So "n is a perfect square" describes a predicate, since you can't say if it's true or false until you know what the value of the variable n happens to be. Once you know, for example, that n equals 4, the predicate becomes the true proposition "4 is a perfect square". Remember, nothing says that the proposition has to be true: if the value of n were 5, you would get the false proposition "5 is a perfect square."

Like other propositions, predicates are often named with a letter. Furthermore, a function-like notation is used to denote a predicate supplied with specific variable values. For example, we might use the name "P" for predicate above:

$$P(n) ::= \text{“}n \text{ is a perfect square”},$$

and repeat the remarks above by asserting that $P(4)$ is true, and $P(5)$ is false.

This notation for predicates is confusingly similar to ordinary function notation. If P is a predicate, then $P(n)$ is either *true* or *false*, depending on the value of n. On the other hand, if p is an ordinary function, like n^2+1, then $p(n)$ is a *numerical quantity*. **Don't confuse these two!**

1.3 The Axiomatic Method

The standard procedure for establishing truth in mathematics was invented by Euclid, a mathematician working in Alexandria, Egypt around 300 BC. His idea was to begin with five *assumptions* about geometry, which seemed undeniable based on direct experience. (For example, "There is a straight line segment between every

pair of points".) Propositions like these that are simply accepted as true are called *axioms*.

Starting from these axioms, Euclid established the truth of many additional propositions by providing "proofs." A *proof* is a sequence of logical deductions from axioms and previously proved statements that concludes with the proposition in question. You probably wrote many proofs in high school geometry class, and you'll see a lot more in this text.

There are several common terms for a proposition that has been proved. The different terms hint at the role of the proposition within a larger body of work.

- Important true propositions are called *theorems*.
- A *lemma* is a preliminary proposition useful for proving later propositions.
- A *corollary* is a proposition that follows in just a few logical steps from a theorem.

These definitions are not precise. In fact, sometimes a good lemma turns out to be far more important than the theorem it was originally used to prove.

Euclid's axiom-and-proof approach, now called the *axiomatic method*, remains the foundation for mathematics today. In fact, just a handful of axioms, called the Zermelo-Fraenkel with Choice axioms (ZFC), together with a few logical deduction rules, appear to be sufficient to derive essentially all of mathematics. We'll examine these in Chapter 7.

1.4 Our Axioms

The ZFC axioms are important in studying and justifying the foundations of mathematics, but for practical purposes, they are much too primitive. Proving theorems in ZFC is a little like writing programs in byte code instead of a full-fledged programming language—by one reckoning, a formal proof in ZFC that $2 + 2 = 4$ requires more than 20,000 steps! So instead of starting with ZFC, we're going to take a *huge* set of axioms as our foundation: we'll accept all familiar facts from high school math.

This will give us a quick launch, but you may find this imprecise specification of the axioms troubling at times. For example, in the midst of a proof, you may start to wonder, "Must I prove this little fact or can I take it as an axiom?" There really is no absolute answer, since what's reasonable to assume and what requires proof depends on the circumstances and the audience. A good general guideline is simply to be up front about what you're assuming.

1.4.1 Logical Deductions

Logical deductions, or *inference rules*, are used to prove new propositions using previously proved ones.

A fundamental inference rule is *modus ponens*. This rule says that a proof of P together with a proof that P IMPLIES Q is a proof of Q.

Inference rules are sometimes written in a funny notation. For example, *modus ponens* is written:

Rule.

$$\frac{P, \quad P \text{ IMPLIES } Q}{Q}$$

When the statements above the line, called the *antecedents*, are proved, then we can consider the statement below the line, called the *conclusion* or *consequent*, to also be proved.

A key requirement of an inference rule is that it must be *sound*: an assignment of truth values to the letters, P, Q, ..., that makes all the antecedents true must also make the consequent true. So if we start off with true axioms and apply sound inference rules, everything we prove will also be true.

There are many other natural, sound inference rules, for example:

Rule.

$$\frac{P \text{ IMPLIES } Q, \quad Q \text{ IMPLIES } R}{P \text{ IMPLIES } R}$$

Rule.

$$\frac{\text{NOT}(P) \text{ IMPLIES NOT}(Q)}{Q \text{ IMPLIES } P}$$

On the other hand,

Non-Rule.

$$\frac{\text{NOT}(P) \text{ IMPLIES NOT}(Q)}{P \text{ IMPLIES } Q}$$

is not sound: if P is assigned **T** and Q is assigned **F**, then the antecedent is true and the consequent is not.

As with axioms, we will not be too formal about the set of legal inference rules. Each step in a proof should be clear and "logical"; in particular, you should state what previously proved facts are used to derive each new conclusion.

1.4.2 Patterns of Proof

In principle, a proof can be *any* sequence of logical deductions from axioms and previously proved statements that concludes with the proposition in question. This freedom in constructing a proof can seem overwhelming at first. How do you even *start* a proof?

Here's the good news: many proofs follow one of a handful of standard templates. Each proof has it own details, of course, but these templates at least provide you with an outline to fill in. We'll go through several of these standard patterns, pointing out the basic idea and common pitfalls and giving some examples. Many of these templates fit together; one may give you a top-level outline while others help you at the next level of detail. And we'll show you other, more sophisticated proof techniques later on.

The recipes below are very specific at times, telling you exactly which words to write down on your piece of paper. You're certainly free to say things your own way instead; we're just giving you something you *could* say so that you're never at a complete loss.

1.5 Proving an Implication

Propositions of the form "If P, then Q" are called *implications*. This implication is often rephrased as "P IMPLIES Q."

Here are some examples:

- (Quadratic Formula) If $ax^2 + bx + c = 0$ and $a \neq 0$, then
$$x = \left(-b \pm \sqrt{b^2 - 4ac}\right)/2a.$$
- (Goldbach's Conjecture 1.1.8 rephrased) If n is an even integer greater than 2, then n is a sum of two primes.
- If $0 \leq x \leq 2$, then $-x^3 + 4x + 1 > 0$.

There are a couple of standard methods for proving an implication.

1.5.1 Method #1

In order to prove that P IMPLIES Q:

1. Write, "Assume P."
2. Show that Q logically follows.

Example

Theorem 1.5.1. *If* $0 \le x \le 2$*, then* $-x^3 + 4x + 1 > 0$*.*

Before we write a proof of this theorem, we have to do some scratchwork to figure out why it is true.

The inequality certainly holds for $x = 0$; then the left side is equal to 1 and $1 > 0$. As x grows, the $4x$ term (which is positive) initially seems to have greater magnitude than $-x^3$ (which is negative). For example, when $x = 1$, we have $4x = 4$, but $-x^3 = -1$ only. In fact, it looks like $-x^3$ doesn't begin to dominate until $x > 2$. So it seems the $-x^3 + 4x$ part should be nonnegative for all x between 0 and 2, which would imply that $-x^3 + 4x + 1$ is positive.

So far, so good. But we still have to replace all those "seems like" phrases with solid, logical arguments. We can get a better handle on the critical $-x^3 + 4x$ part by factoring it, which is not too hard:

$$-x^3 + 4x = x(2 - x)(2 + x)$$

Aha! For x between 0 and 2, all of the terms on the right side are nonnegative. And a product of nonnegative terms is also nonnegative. Let's organize this blizzard of observations into a clean proof.

Proof. Assume $0 \le x \le 2$. Then x, $2-x$, and $2+x$ are all nonnegative. Therefore, the product of these terms is also nonnegative. Adding 1 to this product gives a positive number, so:

$$x(2 - x)(2 + x) + 1 > 0$$

Multiplying out on the left side proves that

$$-x^3 + 4x + 1 > 0$$

as claimed. ■

There are a couple points here that apply to all proofs:

- You'll often need to do some scratchwork while you're trying to figure out the logical steps of a proof. Your scratchwork can be as disorganized as you like—full of dead-ends, strange diagrams, obscene words, whatever. But keep your scratchwork separate from your final proof, which should be clear and concise.

- Proofs typically begin with the word "Proof" and end with some sort of delimiter like □ or "QED." The only purpose for these conventions is to clarify where proofs begin and end.

1.5.2 Method #2 - Prove the Contrapositive

An implication ("P IMPLIES Q") is logically equivalent to its *contrapositive*

$$\text{NOT}(Q) \text{ IMPLIES } \text{NOT}(P).$$

Proving one is as good as proving the other, and proving the contrapositive is sometimes easier than proving the original statement. If so, then you can proceed as follows:

1. Write, "We prove the contrapositive:" and then state the contrapositive.
2. Proceed as in Method #1.

Example

Theorem 1.5.2. *If r is irrational, then $\sqrt{r}$ is also irrational.*

A number is *rational* when it equals a quotient of integers —that is, if it equals m/n for some integers m and n. If it's not rational, then it's called *irrational*. So we must show that if r is *not* a ratio of integers, then $\sqrt{r}$ is also *not* a ratio of integers. That's pretty convoluted! We can eliminate both *not*'s and simplify the proof by using the contrapositive instead.

Proof. We prove the contrapositive: if $\sqrt{r}$ is rational, then r is rational.

Assume that $\sqrt{r}$ is rational. Then there exist integers m and n such that:

$$\sqrt{r} = \frac{m}{n}$$

Squaring both sides gives:

$$r = \frac{m^2}{n^2}$$

Since m^2 and n^2 are integers, r is also rational. ■

1.6 Proving an "If and Only If"

Many mathematical theorems assert that two statements are logically equivalent; that is, one holds if and only if the other does. Here is an example that has been known for several thousand years:

> Two triangles have the same side lengths if and only if two side lengths and the angle between those sides are the same.

The phrase "if and only if" comes up so often that it is often abbreviated "iff."

1.6.1 Method #1: Prove Each Statement Implies the Other

The statement "P IFF Q" is equivalent to the two statements "P IMPLIES Q" and "Q IMPLIES P." So you can prove an "iff" by proving *two* implications:

1. Write, "We prove P implies Q and vice-versa."
2. Write, "First, we show P implies Q." Do this by one of the methods in Section 1.5.
3. Write, "Now, we show Q implies P." Again, do this by one of the methods in Section 1.5.

1.6.2 Method #2: Construct a Chain of Iffs

In order to prove that P is true iff Q is true:

1. Write, "We construct a chain of if-and-only-if implications."
2. Prove P is equivalent to a second statement which is equivalent to a third statement and so forth until you reach Q.

This method sometimes requires more ingenuity than the first, but the result can be a short, elegant proof.

Example

The *standard deviation* of a sequence of values $x_1, x_2, \ldots, x_n$ is defined to be:

$$\sqrt{\frac{(x_1-\mu)^2+(x_2-\mu)^2+\cdots+(x_n-\mu)^2}{n}} \tag{1.3}$$

where μ is the average or *mean* of the values:

$$\mu ::= \frac{x_1+x_2+\cdots+x_n}{n}$$

Theorem 1.6.1. *The standard deviation of a sequence of values $x_1, \ldots, x_n$ is zero iff all the values are equal to the mean.*

For example, the standard deviation of test scores is zero if and only if everyone scored exactly the class average.

Proof. We construct a chain of "iff" implications, starting with the statement that the standard deviation (1.3) is zero:

$$\sqrt{\frac{(x_1-\mu)^2+(x_2-\mu)^2+\cdots+(x_n-\mu)^2}{n}} = 0. \tag{1.4}$$

Now since zero is the only number whose square root is zero, equation (1.4) holds iff

$$(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_n - \mu)^2 = 0. \tag{1.5}$$

Squares of real numbers are always nonnegative, so every term on the left hand side of equation (1.5) is nonnegative. This means that (1.5) holds iff

Every term on the left hand side of (1.5) is zero. (1.6)

But a term $(x_i - \mu)^2$ is zero iff $x_i = \mu$, so (1.6) is true iff

Every x_i equals the mean.

■

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

1.7 Proof by Cases

Breaking a complicated proof into cases and proving each case separately is a common, useful proof strategy. Here's an amusing example.

Let's agree that given any two people, either they have met or not. If every pair of people in a group has met, we'll call the group a *club*. If every pair of people in a group has not met, we'll call it a group of *strangers*.

Theorem. *Every collection of 6 people includes a club of 3 people or a group of 3 strangers.*

Proof. The proof is by case analysis[5]. Let x denote one of the six people. There are two cases:

1. Among 5 other people besides x, at least 3 have met x.
2. Among the 5 other people, at least 3 have not met x.

Now, we have to be sure that at least one of these two cases must hold,[6] but that's easy: we've split the 5 people into two groups, those who have shaken hands with x and those who have not, so one of the groups must have at least half the people.

Case 1: Suppose that at least 3 people did meet x.

This case splits into two subcases:

[5] Describing your approach at the outset helps orient the reader.

[6] Part of a case analysis argument is showing that you've covered all the cases. This is often obvious, because the two cases are of the form "P" and "not P." However, the situation above is not stated quite so simply.

Case 1.1: No pair among those people met each other. Then these people are a group of at least 3 strangers. The theorem holds in this subcase.

Case 1.2: Some pair among those people have met each other. Then that pair, together with x, form a club of 3 people. So the theorem holds in this subcase.

This implies that the theorem holds in Case 1.

Case 2: Suppose that at least 3 people did not meet x.

This case also splits into two subcases:

Case 2.1: Every pair among those people met each other. Then these people are a club of at least 3 people. So the theorem holds in this subcase.

Case 2.2: Some pair among those people have not met each other. Then that pair, together with x, form a group of at least 3 strangers. So the theorem holds in this subcase.

This implies that the theorem also holds in Case 2, and therefore holds in all cases. ■

1.8 Proof by Contradiction

In a *proof by contradiction*, or *indirect proof*, you show that if a proposition were false, then some false fact would be true. Since a false fact by definition can't be true, the proposition must be true.

Proof by contradiction is *always* a viable approach. However, as the name suggests, indirect proofs can be a little convoluted, so direct proofs are generally preferable when they are available.

Method: In order to prove a proposition P by contradiction:

1. Write, "We use proof by contradiction."
2. Write, "Suppose P is false."
3. Deduce something known to be false (a logical contradiction).
4. Write, "This is a contradiction. Therefore, P must be true."

Example

We'll prove by contradiction that $\sqrt{2}$ is irrational. Remember that a number is *rational* if it is equal to a ratio of integers—for example, $3.5 = 7/2$ and $0.1111\cdots = 1/9$ are rational numbers.

Theorem 1.8.1. *$\sqrt{2}$ is irrational.*

Proof. We use proof by contradiction. Suppose the claim is false, and $\sqrt{2}$ is rational. Then we can write $\sqrt{2}$ as a fraction n/d in *lowest terms*.

Squaring both sides gives $2 = n^2/d^2$ and so $2d^2 = n^2$. This implies that n is a multiple of 2 (see Problems 1.10 and 1.11). Therefore n^2 must be a multiple of 4. But since $2d^2 = n^2$, we know $2d^2$ is a multiple of 4 and so d^2 is a multiple of 2. This implies that d is a multiple of 2.

So, the numerator and denominator have 2 as a common factor, which contradicts the fact that n/d is in lowest terms. Thus, $\sqrt{2}$ must be irrational. ■

1.9 *Good* Proofs in Practice

One purpose of a proof is to establish the truth of an assertion with absolute certainty, and mechanically checkable proofs of enormous length or complexity can accomplish this. But humanly intelligible proofs are the only ones that help someone understand the subject. Mathematicians generally agree that important mathematical results can't be fully understood until their proofs are understood. That is why proofs are an important part of the curriculum.

To be understandable and helpful, more is required of a proof than just logical correctness: a good proof must also be clear. Correctness and clarity usually go together; a well-written proof is more likely to be a correct proof, since mistakes are harder to hide.

In practice, the notion of proof is a moving target. Proofs in a professional research journal are generally unintelligible to all but a few experts who know all the terminology and prior results used in the proof. Conversely, proofs in the first weeks of a beginning course like 6.042 would be regarded as tediously long-winded by a professional mathematician. In fact, what we accept as a good proof later in the term will be different from what we consider good proofs in the first couple of weeks of 6.042. But even so, we can offer some general tips on writing good proofs:

State your game plan. A good proof begins by explaining the general line of reasoning, for example, "We use case analysis" or "We argue by contradiction."

Keep a linear flow. Sometimes proofs are written like mathematical mosaics, with juicy tidbits of independent reasoning sprinkled throughout. This is not good. The steps of an argument should follow one another in an intelligible order.

A proof is an essay, not a calculation. Many students initially write proofs the way they compute integrals. The result is a long sequence of expressions without explanation, making it very hard to follow. This is bad. A good proof usually looks like an essay with some equations thrown in. Use complete sentences.

Avoid excessive symbolism. Your reader is probably good at understanding words, but much less skilled at reading arcane mathematical symbols. Use words where you reasonably can.

Revise and simplify. Your readers will be grateful.

Introduce notation thoughtfully. Sometimes an argument can be greatly simplified by introducing a variable, devising a special notation, or defining a new term. But do this sparingly, since you're requiring the reader to remember all that new stuff. And remember to actually *define* the meanings of new variables, terms, or notations; don't just start using them!

Structure long proofs. Long programs are usually broken into a hierarchy of smaller procedures. Long proofs are much the same. When your proof needed facts that are easily stated, but not readily proved, those fact are best pulled out as preliminary lemmas. Also, if you are repeating essentially the same argument over and over, try to capture that argument in a general lemma, which you can cite repeatedly instead.

Be wary of the "obvious." When familiar or truly obvious facts are needed in a proof, it's OK to label them as such and to not prove them. But remember that what's obvious to you may not be—and typically is not—obvious to your reader.

Most especially, don't use phrases like "clearly" or "obviously" in an attempt to bully the reader into accepting something you're having trouble proving. Also, go on the alert whenever you see one of these phrases in someone else's proof.

Finish. At some point in a proof, you'll have established all the essential facts you need. Resist the temptation to quit and leave the reader to draw the "obvious" conclusion. Instead, tie everything together yourself and explain why the original claim follows.

Creating a good proof is a lot like creating a beautiful work of art. In fact, mathematicians often refer to really good proofs as being "elegant" or "beautiful." It takes a practice and experience to write proofs that merit such praises, but to get you started in the right direction, we will provide templates for the most useful proof techniques.

Throughout the text there are also examples of *bogus proofs*—arguments that look like proofs but aren't. Sometimes a bogus proof can reach false conclusions because of missteps or mistaken assumptions. More subtle bogus proofs reach correct conclusions, but do so in improper ways such as circular reasoning, leaping to unjustified conclusions, or saying that the hard part of the proof is "left to the reader." Learning to spot the flaws in improper proofs will hone your skills at seeing how each proof step follows logically from prior steps. It will also enable you to spot flaws in your own proofs.

The analogy between good proofs and good programs extends beyond structure. The same rigorous thinking needed for proofs is essential in the design of critical computer systems. When algorithms and protocols only "mostly work" due to reliance on hand-waving arguments, the results can range from problematic to catastrophic. An early example was the Therac 25, a machine that provided radiation therapy to cancer victims, but occasionally killed them with massive overdoses due to a software race condition. A more recent (August 2004) example involved a single faulty command to a computer system used by United and American Airlines that grounded the entire fleet of both companies—and all their passengers!

It is a certainty that we'll all one day be at the mercy of critical computer systems designed by you and your classmates. So we really hope that you'll develop the ability to formulate rock-solid logical arguments that a system actually does what you think it does!

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

2 The Well Ordering Principle

Every *nonempty* set of *nonnegative integers* has a *smallest* element.

This statement is known as The *Well Ordering Principle*. Do you believe it? Seems sort of obvious, right? But notice how tight it is: it requires a *nonempty* set—it's false for the empty set which has *no* smallest element because it has no elements at all. And it requires a set of *nonnegative* integers—it's false for the set of *negative* integers and also false for some sets of nonnegative *rationals*—for example, the set of positive rationals. So, the Well Ordering Principle captures something special about the nonnegative integers.

While the Well Ordering Principle may seem obvious, it's hard to see offhand why it is useful. But in fact, it provides one of the most important proof rules in discrete mathematics. In this chapter, we'll illustrate the power of this proof method with a few simple examples.

2.1 Well Ordering Proofs

We actually have already taken the Well Ordering Principle for granted in proving that $\sqrt{2}$ is irrational. That proof assumed that for any positive integers m and n, the fraction m/n can be written in *lowest terms*, that is, in the form m'/n' where m' and n' are positive integers with no common prime factors. How do we know this is always possible?

Suppose to the contrary that there are positive integers m and n such that the fraction m/n cannot be written in lowest terms. Now let C be the set of positive integers that are numerators of such fractions. Then $m \in C$, so C is nonempty. Therefore, by Well Ordering, there must be a smallest integer, $m_0 \in C$. So by definition of C, there is an integer $n_0 > 0$ such that

$$\text{the fraction } \frac{m_0}{n_0} \text{ cannot be written in lowest terms.}$$

This means that m_0 and n_0 must have a common prime factor, $p > 1$. But

$$\frac{m_0/p}{n_0/p} = \frac{m_0}{n_0},$$

so any way of expressing the left hand fraction in lowest terms would also work for m_0/n_0, which implies

$$\text{the fraction } \frac{m_0/p}{n_0/p} \text{ cannot be in written in lowest terms either.}$$

So by definition of C, the numerator, m_0/p, is in C. But $m_0/p < m_0$, which contradicts the fact that m_0 is the smallest element of C.

Since the assumption that C is nonempty leads to a contradiction, it follows that C must be empty. That is, that there are no numerators of fractions that can't be written in lowest terms, and hence there are no such fractions at all.

We've been using the Well Ordering Principle on the sly from early on!

2.2 Template for Well Ordering Proofs

More generally, there is a standard way to use Well Ordering to prove that some property, $P(n)$ holds for every nonnegative integer, n. Here is a standard way to organize such a well ordering proof:

To prove that "$P(n)$ is true for all $n \in \mathbb{N}$" using the Well Ordering Principle:

- Define the set, C, of *counterexamples* to P being true. Specifically, define

$$C ::= \{n \in \mathbb{N} \mid \text{NOT}(P(n)) \text{ is true}\}.$$

  (The notation $\{n \mid Q(n)\}$ means "the set of all elements n for which $Q(n)$ is true." See Section 4.1.4.)

- Assume for proof by contradiction that C is nonempty.
- By the Well Ordering Principle, there will be a smallest element, n, in C.
- Reach a contradiction somehow—often by showing that $P(n)$ is actually true or by showing that there is another member of C that is smaller than n. This is the open-ended part of the proof task.
- Conclude that C must be empty, that is, no counterexamples exist. ■

2.2.1 Summing the Integers

Let's use this template to prove

Theorem 2.2.1.

$$1 + 2 + 3 + \cdots + n = n(n+1)/2 \tag{2.1}$$

for all nonnegative integers, n.

First, we'd better address a couple of ambiguous special cases before they trip us up:

- If $n = 1$, then there is only one term in the summation, and so $1 + 2 + 3 + \cdots + n$ is just the term 1. Don't be misled by the appearance of 2 and 3 or by the suggestion that 1 and n are distinct terms!
- If $n = 0$, then there are no terms at all in the summation. By convention, the sum in this case is 0.

So, while the three dots notation, which is called an *ellipsis*, is convenient, you have to watch out for these special cases where the notation is misleading. In fact, whenever you see an ellipsis, you should be on the lookout to be sure you understand the pattern, watching out for the beginning and the end.

We could have eliminated the need for guessing by rewriting the left side of (2.1) with *summation notation*:

$$\sum_{i=1}^{n} i \qquad \text{or} \qquad \sum_{1 \le i \le n} i.$$

Both of these expressions denote the sum of all values taken by the expression to the right of the sigma as the variable, i, ranges from 1 to n. Both expressions make it clear what (2.1) means when $n = 1$. The second expression makes it clear that when $n = 0$, there are no terms in the sum, though you still have to know the convention that a sum of no numbers equals 0 (the *product* of no numbers is 1, by the way).

OK, back to the proof:

Proof. By contradiction. Assume that Theorem 2.2.1 is *false*. Then, some nonnegative integers serve as *counterexamples* to it. Let's collect them in a set:

$$C ::= \{n \in \mathbb{N} \mid 1 + 2 + 3 + \cdots + n \neq \frac{n(n+1)}{2}\}.$$

Assuming there are counterexamples, C is a nonempty set of nonnegative integers. So, by the Well Ordering Principle, C has a minimum element, which we'll call c. That is, among the nonnegative integers, c is the *smallest counterexample* to equation (2.1).

Since c is the smallest counterexample, we know that (2.1) is false for $n = c$ but true for all nonnegative integers $n < c$. But (2.1) is true for $n = 0$, so $c > 0$. This means $c - 1$ is a nonnegative integer, and since it is less than c, equation (2.1) is true for $c - 1$. That is,

$$1 + 2 + 3 + \cdots + (c-1) = \frac{(c-1)c}{2}.$$

But then, adding c to both sides, we get

$$1 + 2 + 3 + \cdots + (c-1) + c = \frac{(c-1)c}{2} + c = \frac{c^2 - c + 2c}{2} = \frac{c(c+1)}{2},$$

which means that (2.1) does hold for c, after all! This is a contradiction, and we are done. ■

2.3 Factoring into Primes

We've previously taken for granted the *Prime Factorization Theorem*, also known as the *Unique Factorization Theorem* and the *Fundamental Theorem of Arithmetic*, which states that every integer greater than one has a unique[1] expression as a product of prime numbers. This is another of those familiar mathematical facts which are taken for granted but are not really obvious on closer inspection. We'll prove the uniqueness of prime factorization in a later chapter, but well ordering gives an easy proof that every integer greater than one can be expressed as *some* product of primes.

Theorem 2.3.1. *Every positive integer greater than one can be factored as a product of primes.*

Proof. The proof is by well ordering.

Let C be the set of all integers greater than one that cannot be factored as a product of primes. We assume C is not empty and derive a contradiction.

If C is not empty, there is a least element, $n \in C$, by well ordering. The n can't be prime, because a prime by itself is considered a (length one) product of primes and no such products are in C.

So n must be a product of two integers a and b where $1 < a, b < n$. Since a and b are smaller than the smallest element in C, we know that $a, b \notin C$. In other words, a can be written as a product of primes $p_1 p_2 \cdots p_k$ and b as a product of

[1] . . . unique up to the order in which the prime factors appear

primes $q_1 \cdots q_l$. Therefore, $n = p_1 \cdots p_k q_1 \cdots q_l$ can be written as a product of primes, contradicting the claim that $n \in C$. Our assumption that C is not empty must therefore be false. ■

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

3.1 Propositions from Propositions

In English, we can modify, combine, and relate propositions with words such as "not," "and," "or," "implies," and "if-then." For example, we can combine three propositions into one like this:

If all humans are mortal **and** all Greeks are human, **then** all Greeks are mortal.

For the next while, we won't be much concerned with the internals of propositions—whether they involve mathematics or Greek mortality—but rather with how propositions are combined and related. So, we'll frequently use variables such as P and Q in place of specific propositions such as "All humans are mortal" and "$2 + 3 = 5$." The understanding is that these *propositional variables*, like propositions, can take on only the values **T** (true) and **F** (false). Propositional variables are also called *Boolean variables* after their inventor, the nineteenth century mathematician George—you guessed it—Boole.

3.1.1 NOT, AND, and OR

Mathematicians use the words NOT, AND, and OR for operations that change or combine propositions. The precise mathematical meaning of these special words can be specified by *truth tables*. For example, if P is a proposition, then so is "NOT(P)," and the truth value of the proposition "NOT(P)" is determined by the truth value of P according to the following truth table:

P	NOT(P)
T	**F**
F	**T**

The first row of the table indicates that when proposition P is true, the proposition "NOT(P)" is false. The second line indicates that when P is false, "NOT(P)" is true. This is probably what you would expect.

In general, a truth table indicates the true/false value of a proposition for each possible set of truth values for the variables. For example, the truth table for the proposition "P AND Q" has four lines, since there are four settings of truth values for the two variables:

P	Q	P AND Q
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

According to this table, the proposition "P AND Q" is true only when P and Q are both true. This is probably the way you ordinarily think about the word "and."

There is a subtlety in the truth table for "P OR Q":

P	Q	P OR Q
T	**T**	**T**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

The first row of this table says that "P OR Q" is true even if *both* P and Q are true. This isn't always the intended meaning of "or" in everyday speech, but this is the standard definition in mathematical writing. So if a mathematician says, "You may have cake, or you may have ice cream," he means that you *could* have both.

If you want to exclude the possibility of having both cake *and* ice cream, you should combine them with the *exclusive-or* operation, XOR:

P	Q	P XOR Q
T	**T**	**F**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

3.1.2 IMPLIES

The combining operation with the least intuitive technical meaning is "implies." Here is its truth table, with the lines labeled so we can refer to them later.

P	Q	P IMPLIES Q	
T	**T**	**T**	(tt)
T	**F**	**F**	(tf)
F	**T**	**T**	(ft)
F	**F**	**T**	(ff)

The truth table for implications can be summarized in words as follows:

> An implication is true exactly when the if-part is false or the then-part is true.

This sentence is worth remembering; a large fraction of all mathematical statements are of the if-then form!

Let's experiment with this definition. For example, is the following proposition true or false?

"If Goldbach's Conjecture is true, then $x^2 \geq 0$ for every real number x."

Now, we already mentioned that no one knows whether Goldbach's Conjecture, Proposition 1.1.8, is true or false. But that doesn't prevent you from answering the question! This proposition has the form P IMPLIES Q where the *hypothesis*, P, is "Goldbach's Conjecture is true" and the *conclusion*, Q, is "$x^2 \geq 0$ for every real number x." Since the conclusion is definitely true, we're on either line (tt) or line (ft) of the truth table. Either way, the proposition as a whole is *true*!

One of our original examples demonstrates an even stranger side of implications.

"If pigs fly, then you can understand the Chebyshev bound."

Don't take this as an insult; we just need to figure out whether this proposition is true or false. Curiously, the answer has *nothing* to do with whether or not you can understand the Chebyshev bound. Pigs do not fly, so we're on either line (ft) or line (ff) of the truth table. In both cases, the proposition is *true*!

In contrast, here's an example of a false implication:

"If the moon shines white, then the moon is made of white cheddar."

Yes, the moon shines white. But, no, the moon is not made of white cheddar cheese. So we're on line (tf) of the truth table, and the proposition is false.

False Hypotheses

It often bothers people when they first learn that implications which have false hypotheses are considered to be true. But implications with false hypotheses hardly ever come up in ordinary settings, so there's not much reason to be bothered by whatever truth assignment logicians and mathematicians choose to give them.

There are, of course, good reasons for the mathematical convention that implications are true when their hypotheses are false. An illustrative example is a system specification (see Problem 3.12) which consisted of a series of, say, a dozen rules,

> if C_i: the system sensors are in condition i, then A_i: the system takes action i,

or more concisely,

$$C_i \text{ IMPLIES } A_i$$

for $1 \leq i \leq 12$. Then the fact that the system obeys the specification would be expressed by saying that the AND

$$[C_1 \text{ IMPLIES } A_1] \text{ AND } [C_2 \text{ IMPLIES } A_2] \text{ AND } \cdots \text{ AND } [C_{12} \text{ IMPLIES } A_{12}] \quad (3.1)$$

of these rules was always true.

For example, suppose only conditions C_2 and C_5 are true, and the system indeed takes the specified actions A_2 and A_5. This means that in this case the system is behaving according to specification, and accordingly we want the formula (3.1) to come out true. Now the implications C_2 IMPLIES A_2 and C_5 IMPLIES A_5 are both true because both their hypotheses and their conclusions are true. But in order for (3.1) to be true, we need all the other implications with the false hypotheses C_i for $i \neq 2, 5$ to be true. This is exactly what the rule for implications with false hypotheses accomplishes.

3.1.3 If and Only If

Mathematicians commonly join propositions in one additional way that doesn't arise in ordinary speech. The proposition "P if and only if Q" asserts that P and Q have the same truth value. Either both are true or both are false.

P	Q	P IFF Q
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**T**

For example, the following if-and-only-if statement is true for every real number x:

$$x^2 - 4 \geq 0 \text{ IFF } |x| \geq 2.$$

For some values of x, *both* inequalities are true. For other values of x, *neither* inequality is true. In every case, however, the IFF proposition as a whole is true.

3.2 Propositional Logic in Computer Programs

Propositions and logical connectives arise all the time in computer programs. For example, consider the following snippet, which could be either C, C++, or Java:

```
if ( x > 0 || (x <= 0 && y > 100) )
    ⋮
(further instructions)
```

Java uses the symbol `||` for "OR," and the symbol `&&` for "AND." The *further instructions* are carried out only if the proposition following the word `if` is true. On closer inspection, this big expression is built from two simpler propositions.

Let A be the proposition that `x > 0`, and let B be the proposition that `y > 100`. Then we can rewrite the condition as

$$A \text{ OR } (\text{NOT}(A) \text{ AND } B). \tag{3.2}$$

3.2.1 Truth Table Calculation

A truth table calculation reveals that the more complicated expression 3.2 always has the same truth value as

$$A \text{ OR } B. \tag{3.3}$$

We begin with a table with just the truth values of A and B:

A	B	A OR ($\text{NOT}(A)$ AND B)	A OR B
T	**T**		
T	**F**		
F	**T**		
F	**F**		

These values are enough to fill in two more columns:

A	B	A OR ($\text{NOT}(A)$ AND B)	A OR B
T	**T**	**F**	**T**
T	**F**	**F**	**T**
F	**T**	**T**	**T**
F	**F**	**T**	**F**

Now we have the values needed to fill in the AND column:

A	B	A OR ($\text{NOT}(A)$ AND B)	A OR B
T	**T**	**F** **F**	**T**
T	**F**	**F** **F**	**T**
F	**T**	**T** **T**	**T**
F	**F**	**T** **F**	**F**

and this provides the values needed to fill in the remaining column for the first OR:

A	B	A OR ($\text{NOT}(A)$ AND B)	A OR B
T	**T**	**T** **F** **F**	**T**
T	**F**	**T** **F** **F**	**T**
F	**T**	**T** **T** **T**	**T**
F	**F**	**F** **T** **F**	**F**

Expressions whose truth values always match are called *equivalent*. Since the two emphasized columns of truth values of the two expressions are the same, they are

equivalent. So we can simplify the code snippet without changing the program's behavior by replacing the complicated expression with an equivalent simpler one:

```
if ( x > 0 || y > 100 )
    ⋮
(further instructions)
```

The equivalence of (3.2) and (3.3) can also be confirmed reasoning by cases:

A is **T**. An expression of the form (**T** OR anything) is equivalent to **T**. Since *A* is **T** both (3.2) and (3.3) in this case are of this form, so they have the same truth value, namely, **T**.

A is **F**. An expression of the form (**F** OR *anything*) will have same truth value as *anything*. Since *A* is **F**, (3.3) has the same truth value as *B*.

An expression of the form (**T** AND *anything*) is equivalent to *anything*, as is any expression of the form **F** OR *anything*. So in this case *A* OR (NOT(*A*) AND *B*) is equivalent to (NOT(*A*) AND *B*), which in turn is equivalent to *B*.

Therefore both (3.2) and (3.3) will have the same truth value in this case, namely, the value of *B*.

Simplifying logical expressions has real practical importance in computer science. Expression simplification in programs like the one above can make a program easier to read and understand. Simplified programs may also run faster, since they require fewer operations. In hardware, simplifying expressions can decrease the number of logic gates on a chip because digital circuits can be described by logical formulas (see Problems 3.5 and 3.6). Minimizing the logical formulas corresponds to reducing the number of gates in the circuit. The payoff of gate minimization is potentially enormous: a chip with fewer gates is smaller, consumes less power, has a lower defect rate, and is cheaper to manufacture.

3.2.2 Cryptic Notation

Java uses symbols like "&&" and "||" in place of AND and OR. Circuit designers use "·" and "+," and actually refer to AND as a product and OR as a sum. Mathematicians use still other symbols, given in the table below.

English	Symbolic Notation
NOT(P)	$\neg P$ (alternatively, $\overline{P}$)
P AND Q	$P \wedge Q$
P OR Q	$P \vee Q$
P IMPLIES Q	$P \longrightarrow Q$
if P then Q	$P \longrightarrow Q$
P IFF Q	$P \longleftrightarrow Q$
P XOR Q	$P \oplus Q$

For example, using this notation, "If P AND NOT(Q), then R" would be written:

$$(P \wedge \overline{Q}) \longrightarrow R.$$

The mathematical notation is concise but cryptic. Words such as "AND" and "OR" are easier to remember and won't get confused with operations on numbers. We will often use $\overline{P}$ as an abbreviation for NOT(P), but aside from that, we mostly stick to the words—except when formulas would otherwise run off the page.

3.3 Equivalence and Validity

3.3.1 Implications and Contrapositives

Do these two sentences say the same thing?

If I am hungry, then I am grumpy.
If I am not grumpy, then I am not hungry.

We can settle the issue by recasting both sentences in terms of propositional logic. Let P be the proposition "I am hungry" and Q be "I am grumpy." The first sentence says "P IMPLIES Q" and the second says "NOT(Q) IMPLIES NOT(P)." Once more, we can compare these two statements in a truth table:

P	Q	(P IMPLIES Q)	(NOT(Q)	IMPLIES	NOT(P))
T	**T**	**T**	**F**	**T**	**F**
T	**F**	**F**	**T**	**F**	**F**
F	**T**	**T**	**F**	**T**	**T**
F	**F**	**T**	**T**	**T**	**T**

Sure enough, the highlighted columns showing the truth values of these two statements are the same. A statement of the form "NOT(Q) IMPLIES NOT(P)" is called

the *contrapositive* of the implication "P IMPLIES Q." The truth table shows that an implication and its contrapositive are equivalent—they are just different ways of saying the same thing.

In contrast, the *converse* of "P IMPLIES Q" is the statement "Q IMPLIES P." The converse to our example is:

If I am grumpy, then I am hungry.

This sounds like a rather different contention, and a truth table confirms this suspicion:

P	Q	P IMPLIES Q	Q IMPLIES P
T	**T**	**T**	**T**
T	**F**	**F**	**T**
F	**T**	**T**	**F**
F	**F**	**T**	**T**

Now the highlighted columns differ in the second and third row, confirming that an implication is generally *not* equivalent to its converse.

One final relationship: an implication and its converse together are equivalent to an iff statement, specifically, to these two statements together. For example,

If I am grumpy then I am hungry, and if I am hungry then I am grumpy.

are equivalent to the single statement:

I am grumpy iff I am hungry.

Once again, we can verify this with a truth table.

P	Q	(P IMPLIES Q)	AND	(Q IMPLIES P)	P IFF Q
T	**T**	**T**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**	**F**
F	**T**	**T**	**F**	**F**	**F**
F	**F**	**T**	**T**	**T**	**T**

The fourth column giving the truth values of

(P IMPLIES Q) AND (Q IMPLIES P)

is the same as the sixth column giving the truth values of P IFF Q, which confirms that the AND of the implications is equivalent to the IFF statement.

3.3.2 Validity and Satisfiability

A *valid* formula is one which is *always* true, no matter what truth values its variables may have. The simplest example is

$$P \text{ OR NOT}(P).$$

You can think about valid formulas as capturing fundamental logical truths. For example, a property of implication that we take for granted is that if one statement implies a second one, and the second one implies a third, then the first implies the third. The following valid formula confirms the truth of this property of implication.

$$[(P \text{ IMPLIES } Q) \text{ AND } (Q \text{ IMPLIES } R)] \text{ IMPLIES } (P \text{ IMPLIES } R).$$

Equivalence of formulas is really a special case of validity. Namely, statements F and G are equivalent precisely when the statement $(F \text{ IFF } G)$ is valid. For example, the equivalence of the expressions (3.3) and (3.2) means that

$$(A \text{ OR } B) \text{ IFF } (A \text{ OR } (\text{NOT}(A) \text{ AND } B))$$

is valid. Of course, validity can also be viewed as an aspect of equivalence. Namely, a formula is valid iff it is equivalent to **T**.

A *satisfiable* formula is one which can *sometimes* be true—that is, there is some assignment of truth values to its variables that makes it true. One way satisfiability comes up is when there are a collection of system specifications. The job of the system designer is to come up with a system that follows all the specs. This means that the AND of all the specs must be satisfiable or the designer's job will be impossible (see Problem 3.12).

There is also a close relationship between validity and satisfiability: a statement P is satisfiable iff its negation $\text{NOT}(P)$ is *not* valid.

3.4 The Algebra of Propositions

3.4.1 Propositions in Normal Form

Every propositional formula is equivalent to a "sum-of-products" or *disjunctive form*. More precisely, a disjunctive form is simply an OR of AND-terms, where each AND-term is an AND of variables or negations of variables, for example,

$$(A \text{ AND } B) \text{ OR } (A \text{ AND } C). \tag{3.4}$$

You can read a disjunctive form for any propositional formula directly from its truth table. For example, the formula

$$A \text{ AND } (B \text{ OR } C) \tag{3.5}$$

has truth table:

A	B	C	A AND (B OR C)
T	**T**	**T**	**T**
T	**T**	**F**	**T**
T	**F**	**T**	**T**
T	**F**	**F**	**F**
F	**T**	**T**	**F**
F	**T**	**F**	**F**
F	**F**	**T**	**F**
F	**F**	**F**	**F**

The formula (3.5) is true in the first row when A, B, and C are all true, that is, where A AND B AND C is true. It is also true in the second row where A AND B AND $\overline{C}$ is true, and in the third row when A AND $\overline{B}$ AND C is true, and that's all. So (3.5) is true exactly when

$$(A \text{ AND } B \text{ AND } C) \text{ OR } (A \text{ AND } B \text{ AND } \overline{C}) \text{ OR } (A \text{ AND } \overline{B} \text{ AND } C) \tag{3.6}$$

is true.

Theorem 3.4.1. *[Distributive Law of* AND *over* OR*]*

$$A \text{ AND } (B \text{ OR } C) \textit{ is equivalent to } (A \text{ AND } B) \text{ OR } (A \text{ AND } C).$$

Theorem 3.4.1 is called a *distributive law* because of its resemblance to the distributivity of products over sums in arithmetic.

Similarly, we have (Problem 3.10):

Theorem 3.4.2. *[Distributive Law of* OR *over* AND*]*

$$A \text{ OR } (B \text{ AND } C) \textit{ is equivalent to } (A \text{ OR } B) \text{ AND } (A \text{ OR } C).$$

Note the contrast between Theorem 3.4.2 and arithmetic, where sums do not distribute over products.

The expression (3.6) is a disjunctive form where each AND-term is an AND of *every one* of the variables or their negations in turn. An expression of this form is called a *disjunctive normal form* (*DNF*). A DNF formula can often be simplified into a smaller disjunctive form. For example, the DNF (3.6) further simplifies to the equivalent disjunctive form (3.4) above.

Applying the same reasoning to the **F** entries of a truth table yields a *conjunctive form* for any formula—an AND of OR-terms in which the OR-terms are OR's only of variables or their negations. For example, formula (3.5) is false in the fourth row of its truth table (3.4.1) where A is **T**, B is **F** and C is **F**. But this is exactly the one row where $(\overline{A}$ OR B OR $C)$ is **F**! Likewise, the (3.5) is false in the fifth row which is exactly where $(A$ OR $\overline{B}$ OR $\overline{C})$ is **F**. This means that (3.5) will be **F** whenever the AND of these two OR-terms is false. Continuing in this way with the OR-terms corresponding to the remaining three rows where (3.5) is false, we get a *conjunctive normal form* (*CNF*) that is equivalent to (3.5), namely,

$$(\overline{A} \text{ OR } B \text{ OR } C) \text{ AND } (A \text{ OR } \overline{B} \text{ OR } \overline{C}) \text{ AND } (A \text{ OR } \overline{B} \text{ OR } C) \text{AND}$$
$$(A \text{ OR } B \text{ OR } \overline{C}) \text{ AND } (A \text{ OR } B \text{ OR } C)$$

The methods above can be applied to any truth table, which implies

Theorem 3.4.3. *Every propositional formula is equivalent to both a disjunctive normal form and a conjunctive normal form.*

3.4.2 Proving Equivalences

A check of equivalence or validity by truth table runs out of steam pretty quickly: a proposition with n variables has a truth table with 2^n lines, so the effort required to check a proposition grows exponentially with the number of variables. For a proposition with just 30 variables, that's already over a billion lines to check!

An alternative approach that *sometimes* helps is to use algebra to prove equivalence. A lot of different operators may appear in a propositional formula, so a useful first step is to get rid of all but three: AND, OR, and NOT. This is easy because each of the operators is equivalent to a simple formula using only these three. For example, A IMPLIES B is equivalent to NOT(A) OR B. Formulas using only AND, OR, and NOT for the remaining operators are left to Problem 3.13.

We list below a bunch of equivalence axioms with the symbol " $\longleftrightarrow$ " between equivalent formulas. These axioms are important because they are all that's needed to prove every possible equivalence. We'll start with some equivalences for AND's that look like the familiar ones for multiplication of numbers:

$$A \text{ AND } B \longleftrightarrow B \text{ AND } A \qquad \text{(commutativity of AND)} \tag{3.7}$$
$$(A \text{ AND } B) \text{ AND } C \longleftrightarrow A \text{ AND } (B \text{ AND } C) \qquad \text{(associativity of AND)} \tag{3.8}$$
$$\mathbf{T} \text{ AND } A \longleftrightarrow A \qquad \text{(identity for AND)}$$
$$\mathbf{F} \text{ AND } A \longleftrightarrow \mathbf{F} \qquad \text{(zero for AND)}$$

Three axioms that don't directly correspond to number properties are

$$A \text{ AND } A \longleftrightarrow A \qquad \text{(idempotence for AND)}$$

$$A \text{ AND } \overline{A} \longleftrightarrow \mathbf{F} \qquad \text{(contradiction for AND)} \tag{3.9}$$

$$\text{NOT}(\overline{A}) \longleftrightarrow A \qquad \text{(double negation)} \tag{3.10}$$

It is associativity (3.8) that justifies writing A AND B AND C without specifying whether it is parenthesized as A AND (B AND C) or (A AND B) AND C. Both ways of inserting parentheses yield equivalent formulas.

There are a corresponding set of equivalences for OR which we won't bother to list, except for the OR rule corresponding to contradiction for AND (3.9):

$$A \text{ OR } \overline{A} \longleftrightarrow \mathbf{T} \qquad \text{(validity for OR)}$$

Finally, there are *DeMorgan's Laws* which explain how to distribute NOT's over AND's and OR's:

$$\text{NOT}(A \text{ AND } B) \longleftrightarrow \overline{A} \text{ OR } \overline{B} \qquad \text{(DeMorgan for AND)} \tag{3.11}$$

$$\text{NOT}(A \text{ OR } B) \longleftrightarrow \overline{A} \text{ AND } \overline{B} \qquad \text{(DeMorgan for OR)} \tag{3.12}$$

All of these axioms can be verified easily with truth tables.

These axioms are all that's needed to convert any formula to a disjunctive normal form. We can illustrate how they work by applying them to turn the negation of formula (3.5),

$$\text{NOT}((A \text{ AND } B) \text{ OR } (A \text{ AND } C)). \tag{3.13}$$

into disjunctive normal form.

We start by applying DeMorgan's Law for OR (3.12) to (3.13) in order to move the NOT deeper into the formula. This gives

$$\text{NOT}(A \text{ AND } B) \text{ AND NOT}(A \text{ AND } C).$$

Now applying Demorgan's Law for AND (3.11) to the two innermost AND-terms, gives

$$(\overline{A} \text{ OR } \overline{B}) \text{ AND } (\overline{A} \text{ OR } \overline{C}). \tag{3.14}$$

At this point NOT only applies to variables, and we won't need Demorgan's Laws any further.

Now we will repeatedly apply The Distributivity of AND over OR (Theorem 3.4.1) to turn (3.14) into a disjunctive form. To start, we'll distribute ($\overline{A}$ OR $\overline{B}$) over AND to get

$$((\overline{A} \text{ OR } \overline{B}) \text{ AND } \overline{A}) \text{ OR } ((\overline{A} \text{ OR } \overline{B}) \text{ AND } \overline{C}).$$

Using distributivity over both AND's we get

$$((\overline{A} \text{ AND } \overline{A}) \text{ OR } (\overline{B} \text{ AND } \overline{A})) \text{ OR } ((\overline{A} \text{ AND } \overline{C}) \text{ OR } (\overline{B} \text{ AND } \overline{C})).$$

By the way, we've implicitly used commutativity (3.7) here to justify distributing over an AND from the right. Now applying idempotence to remove the duplicate occurrence of $\overline{A}$ we get

$$(\overline{A} \text{ OR } (\overline{B} \text{ AND } \overline{A})) \text{ OR } ((\overline{A} \text{ AND } \overline{C}) \text{ OR } (\overline{B} \text{ AND } \overline{C})).$$

Associativity now allows dropping the parentheses around the terms being OR'd to yield the following disjunctive form for (3.13):

$$\overline{A} \text{ OR } (\overline{B} \text{ AND } \overline{A}) \text{ OR } (\overline{A} \text{ AND } \overline{C}) \text{ OR } (\overline{B} \text{ AND } \overline{C}). \tag{3.15}$$

The last step is to turn each of these AND-terms into a disjunctive normal form with all three variables A, B, and C. We'll illustrate how to do this for the second AND-term $(\overline{B} \text{ AND } \overline{A})$. This term needs to mention C to be in normal form. To introduce C, we use validity for OR and identity for AND to conclude that

$$(\overline{B} \text{ AND } \overline{A}) \longleftrightarrow (\overline{B} \text{ AND } \overline{A}) \text{ AND } (C \text{ OR } \overline{C}).$$

Now distributing $(\overline{B} \text{ AND } \overline{A})$ over the OR yields the disjunctive normal form

$$(\overline{B} \text{ AND } \overline{A} \text{ AND } C) \text{ OR } (\overline{B} \text{ AND } \overline{A} \text{ AND } \overline{C}).$$

Doing the same thing to the other AND-terms in (3.15) finally gives a disjunctive normal form for (3.5):

$$\begin{array}{l}
(\overline{A} \text{ AND } B \text{ AND } C) \text{ OR } (\overline{A} \text{ AND } B \text{ AND } \overline{C}) \text{ OR} \\
(\overline{A} \text{ AND } \overline{B} \text{ AND } C) \text{ OR } (\overline{A} \text{ AND } \overline{B} \text{ AND } \overline{C}) \text{ OR} \\
(\overline{B} \text{ AND } \overline{A} \text{ AND } C) \text{ OR } (\overline{B} \text{ AND } \overline{A} \text{ AND } \overline{C}) \text{ OR} \\
(\overline{A} \text{ AND } \overline{C} \text{ AND } B) \text{ OR } (\overline{A} \text{ AND } \overline{C} \text{ AND } \overline{B}) \text{ OR} \\
(\overline{B} \text{ AND } \overline{C} \text{ AND } A) \text{ OR } (\overline{B} \text{ AND } \overline{C} \text{ AND } \overline{A}).
\end{array}$$

Using commutativity to sort the term and OR-idempotence to remove duplicates, finally yields a unique sorted DNF:

$$\begin{array}{l}
(A \text{ AND } \overline{B} \text{ AND } \overline{C}) \text{ OR} \\
(\overline{A} \text{ AND } B \text{ AND } C) \text{ OR} \\
(\overline{A} \text{ AND } B \text{ AND } \overline{C}) \text{ OR} \\
(\overline{A} \text{ AND } \overline{B} \text{ AND } C) \text{ OR} \\
(\overline{A} \text{ AND } \overline{B} \text{ AND } \overline{C}).
\end{array}$$

This example illustrates a strategy for applying these equivalences to convert any formula into disjunctive normal form, and conversion to conjunctive normal form works similarly, which explains:

Theorem 3.4.4. *Any propositional formula can be transformed into disjunctive normal form or a conjunctive normal form using the equivalences listed above.*

What has this got to do with equivalence? That's easy: to prove that two formulas are equivalent, convert them both to disjunctive normal form over the set of variables that appear in the terms. Then use commutativity to sort the variables and AND-terms so they all appear in some standard order. We claim the formulas are equivalent iff they have the same sorted disjunctive normal form. This is obvious if they do have the same disjunctive normal form. But conversely, the way we read off a disjunctive normal form from a truth table shows that two different sorted DNF's over the same set of variables correspond to different truth tables and hence to inequivalent formulas. This proves

Theorem 3.4.5 (Completeness of the propositional equivalence axioms)**.** *Two propositional formula are equivalent iff they can be proved equivalent using the equivalence axioms listed above.*

The benefit of the axioms is that they leave room for ingeniously applying them to prove equivalences with less effort than the truth table method. Theorem 3.4.5 then adds the reassurance that the axioms are guaranteed to prove every equivalence, which is a great punchline for this section. But we don't want to mislead you: it's important to realize that using the strategy we gave for applying the axioms involves essentially the same effort it would take to construct truth tables, and there is no guarantee that applying the axioms will generally be any easier than using truth tables.

3.5 The SAT Problem

Determining whether or not a more complicated proposition is satisfiable is not so easy. How about this one?

$$(P \text{ OR } Q \text{ OR } R) \text{ AND } (\overline{P} \text{ OR } \overline{Q}) \text{ AND } (\overline{P} \text{ OR } \overline{R}) \text{ AND } (\overline{R} \text{ OR } \overline{Q})$$

The general problem of deciding whether a proposition is satisfiable is called *SAT*. One approach to SAT is to construct a truth table and check whether or not a **T** ever appears, but as with testing validity, this approach quickly bogs down for formulas with many variables because truth tables grow exponentially with the number of variables.

Is there a more efficient solution to SAT? In particular, is there some brilliant procedure that determines SAT in a number of steps that grows *polynomially*—like

n^2 or n^{14}—instead of *exponentially*—2^n—whether any given proposition of size n is satisfiable or not? No one knows. And an awful lot hangs on the answer.

The general definition of an "efficient" procedure is one that runs in *polynomial time*, that is, that runs in a number of basic steps bounded by a polynomial in s, where s is the size of an input. It turns out that an efficient solution to SAT would immediately imply efficient solutions to many other important problems involving scheduling, routing, resource allocation, and circuit verification across multiple disciplines including programming, algebra, finance, and political theory. This would be wonderful, but there would also be worldwide chaos. Decrypting coded messages would also become an easy task, so online financial transactions would be insecure and secret communications could be read by everyone. Why this would happen is explained in Section 8.12.

Of course, the situation is the same for validity checking, since you can check for validity by checking for satisfiability of a negated formula. This also explains why the simplification of formulas mentioned in Section 3.2 would be hard—validity testing is a special case of determining if a formula simplifies to **T**.

Recently there has been exciting progress on *SAT-solvers* for practical applications like digital circuit verification. These programs find satisfying assignments with amazing efficiency even for formulas with millions of variables. Unfortunately, it's hard to predict which kind of formulas are amenable to SAT-solver methods, and for formulas that are *un*satisfiable, SAT-solvers generally get nowhere.

So no one has a good idea how to solve SAT in polynomial time, or how to prove that it can't be done—researchers are completely stuck. The problem of determining whether or not SAT has a polynomial time solution is known as the "**P** vs. **NP**" problem.[1] It is the outstanding unanswered question in theoretical computer science. It is also one of the seven Millenium Problems: the Clay Institute will award you $1,000,000 if you solve the **P** vs. **NP** problem.

---

[1] **P** stands for problems whose instances can be solved in time that grows polynomially with the size of the instance. **NP** stands for ***n****ondeterministtic* ***p****olynomial time*, but we'll leave an explanation of what that is to texts on the theory of computational complexity.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

3.6 Predicate Formulas

3.6.1 Quantifiers

The "for all" notation, $\forall$, has already made an early appearance in Section 1.1. For example, the predicate

$$"x^2 \geq 0"$$

is always true when x is a real number. That is,

$$\forall x \in \mathbb{R}.\, x^2 \geq 0$$

is a true statement. On the other hand, the predicate

$$\text{“}5x^2 - 7 = 0\text{”}$$

is only sometimes true; specifically, when $x = \pm\sqrt{7/5}$. There is a "there exists" notation, $\exists$, to indicate that a predicate is true for at least one, but not necessarily all objects. So

$$\exists x \in \mathbb{R}.\, 5x^2 - 7 = 0$$

is true, while

$$\forall x \in \mathbb{R}.\, 5x^2 - 7 = 0$$

is not true.

There are several ways to express the notions of "always true" and "sometimes true" in English. The table below gives some general formats on the left and specific examples using those formats on the right. You can expect to see such phrases hundreds of times in mathematical writing!

Always True

For all $x \in D$, $P(x)$ is true.	For all $x \in \mathbb{R}$, $x^2 \geq 0$.
$P(x)$ is true for every x in the set, D.	$x^2 \geq 0$ for every $x \in \mathbb{R}$.

Sometimes True

There is an $x \in D$ such that $P(x)$ is true.	There is an $x \in \mathbb{R}$ such that $5x^2 - 7 = 0$.
$P(x)$ is true for some x in the set, D.	$5x^2 - 7 = 0$ for some $x \in \mathbb{R}$.
$P(x)$ is true for at least one $x \in D$.	$5x^2 - 7 = 0$ for at least one $x \in \mathbb{R}$.

All these sentences "quantify" how often the predicate is true. Specifically, an assertion that a predicate is always true is called a *universal quantification*, and an assertion that a predicate is sometimes true is an *existential quantification*. Sometimes the English sentences are unclear with respect to quantification:

$$\begin{aligned}&\text{If you can solve any problem we come up with,}\\ &\qquad\text{then you get an } A \text{ for the course.}\end{aligned} \tag{3.16}$$

The phrase "you can solve any problem we can come up with" could reasonably be interpreted as either a universal or existential quantification:

$$\text{you can solve } \textit{every} \text{ problem we come up with,} \tag{3.17}$$

or maybe

you can solve *at least one* problem we come up with. (3.18)

To be precise, let Probs be the set of problems we come up with, Solves(x) be the predicate "You can solve problem x," and G be the proposition, "You get an A for the course." Then the two different interpretations of (3.16) can be written as follows:

$$(\forall x \in \text{Probs. Solves}(x)) \text{ IMPLIES } G, \qquad \text{for (3.17)},$$
$$(\exists x \in \text{Probs. Solves}(x)) \text{ IMPLIES } G. \qquad \text{for (3.18)}.$$

3.6.2 Mixing Quantifiers

Many mathematical statements involve several quantifiers. For example, we already described

> Goldbach's Conjecture 1.1.8: Every even integer greater than 2 is the sum of two primes.

Let's write this out in more detail to be precise about the quantification:

> For every even integer n greater than 2, there exist primes p and q such that $n = p + q$.

Let Evens be the set of even integers greater than 2, and let Primes be the set of primes. Then we can write Goldbach's Conjecture in logic notation as follows:

$$\underbrace{\forall n \in \text{Evens}}_{\substack{\text{for every even} \\ \text{integer } n > 2}} \underbrace{\exists p \in \text{Primes}\, \exists q \in \text{Primes}.}_{\substack{\text{there exist primes} \\ p \text{ and } q \text{ such that}}} n = p + q.$$

3.6.3 Order of Quantifiers

Swapping the order of different kinds of quantifiers (existential or universal) usually changes the meaning of a proposition. For example, let's return to one of our initial, confusing statements:

"Every American has a dream."

This sentence is ambiguous because the order of quantifiers is unclear. Let A be the set of Americans, let D be the set of dreams, and define the predicate $H(a, d)$ to be "American a has dream d." Now the sentence could mean there is a single dream that every American shares—such as the dream of owning their own home:

$$\exists d \in D\, \forall a \in A.\, H(a, d)$$

Or it could mean that every American has a personal dream:

$$\forall a \in A\, \exists\, d \in D.\, H(a, d)$$

For example, some Americans may dream of a peaceful retirement, while others dream of continuing practicing their profession as long as they live, and still others may dream of being so rich they needn't think about work at all.

Swapping quantifiers in Goldbach's Conjecture creates a patently false statement that every even number ≥ 2 is the sum of *the same* two primes:

$$\underbrace{\exists\, p \in \text{Primes}\, \exists\, q \in \text{Primes}.}_{\substack{\text{there exist primes}\\ p \text{ and } q \text{ such that}}}\ \underbrace{\forall n \in \text{Evens}}_{\substack{\text{for every even}\\ \text{integer } n > 2}}\ n = p + q.$$

3.6.4 Variables Over One Domain

When all the variables in a formula are understood to take values from the same nonempty set, D, it's conventional to omit mention of D. For example, instead of $\forall x \in D\, \exists y \in D.\, Q(x, y)$ we'd write $\forall x \exists y.\, Q(x, y)$. The unnamed nonempty set that x and y range over is called the *domain of discourse*, or just plain *domain*, of the formula.

It's easy to arrange for all the variables to range over one domain. For example, Goldbach's Conjecture could be expressed with all variables ranging over the domain $\mathbb{N}$ as

$$\forall n.\, n \in \text{Evens IMPLIES } (\exists\, p\, \exists q.\, p \in \text{Primes AND } q \in \text{Primes AND } n = p + q).$$

3.6.5 Negating Quantifiers

There is a simple relationship between the two kinds of quantifiers. The following two sentences mean the same thing:

> Not everyone likes ice cream.
>
> There is someone who does not like ice cream.

The equivalence of these sentences is a instance of a general equivalence that holds between predicate formulas:

$$\text{NOT}(\forall x.\, P(x)) \quad \text{is equivalent to} \quad \exists x.\, \text{NOT}(P(x)). \tag{3.19}$$

Similarly, these sentences mean the same thing:

> There is no one who likes being mocked.
>
> Everyone dislikes being mocked.

The corresponding predicate formula equivalence is

$$\text{NOT}(\exists x.\, P(x)) \quad \text{is equivalent to} \quad \forall x.\, \text{NOT}(P(x)). \tag{3.20}$$

The general principle is that *moving a* NOT *across a quantifier changes the kind of quantifier.* Note that (3.20) follows from negating both sides of (3.19).

3.6.6 Validity for Predicate Formulas

The idea of validity extends to predicate formulas, but to be valid, a formula now must evaluate to true no matter what the domain of discourse may be, no matter what values its variables may take over the domain, and no matter what interpretations its predicate variables may be given. For example, the equivalence (3.19) that gives the rule for negating a universal quantifier means that the following formula is valid:

$$\text{NOT}(\forall x.\, P(x)) \text{ IFF } \exists x.\, \text{NOT}(P(x)). \tag{3.21}$$

Another useful example of a valid assertion is

$$\exists x \forall y.\, P(x, y) \text{ IMPLIES } \forall y \exists x.\, P(x, y). \tag{3.22}$$

Here's an explanation why this is valid:

> Let D be the domain for the variables and P_0 be some binary predicate[2] on D. We need to show that if
>
> $$\exists x \in D.\, \forall y \in D.\, P_0(x, y) \tag{3.23}$$
>
> holds under this interpretation, then so does
>
> $$\forall y \in D\, \exists x \in D.\, P_0(x, y). \tag{3.24}$$
>
> So suppose (3.23) is true. Then by definition of $\exists$, this means that some element $d_0 \in D$ has the property that
>
> $$\forall y \in D.\, P_0(d_0, y).$$
>
> By definition of $\forall$, this means that
>
> $$P_0(d_0, d)$$
>
> is true for all $d \in D$. So given any $d \in D$, there is an element in D, namely, d_0, such that $P_0(d_0, d)$ is true. But that's exactly what (3.24) means, so we've proved that (3.24) holds under this interpretation, as required.

[2] That is, a predicate that depends on two variables.

We hope this is helpful as an explanation, but we don't really want to call it a "proof." The problem is that with something as basic as (3.22), it's hard to see what more elementary axioms are ok to use in proving it. What the explanation above did was translate the logical formula (3.22) into English and then appeal to the meaning, in English, of "for all" and "there exists" as justification.

In contrast to (3.22), the formula

$$\forall y \exists x.\ P(x, y) \quad \text{IMPLIES} \quad \exists x \forall y.\ P(x, y). \tag{3.25}$$

is *not* valid. We can prove this just by describing an interpretation where the hypothesis, $\forall y \exists x.\ P(x, y)$, is true but the conclusion, $\exists x \forall y.\ P(x, y)$, is not true. For example, let the domain be the integers and $P(x, y)$ mean $x > y$. Then the hypothesis would be true because, given a value, n, for y we could choose the value of x to be $n + 1$, for example. But under this interpretation the conclusion asserts that there is an integer that is bigger than all integers, which is certainly false. An interpretation like this that falsifies an assertion is called a *counter model* to that assertion.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

4 Mathematical Data Types

We have assumed that you've already been introduced to the concepts of sets, sequences, and functions, and we've used them informally several times in previous sections. In this chapter, we'll now take a more careful look at these mathematical data types. We'll quickly review the basic definitions, add a few more such as "images" and "inverse images" that may not be familiar, and end the chapter with some methods for comparing the sizes of sets.

4.1 Sets

Informally, a *set* is a bunch of objects, which are called the *elements* of the set. The elements of a set can be just about anything: numbers, points in space, or even other sets. The conventional way to write down a set is to list the elements inside curly-braces. For example, here are some sets:

$$A = \{\text{Alex}, \text{Tippy}, \text{Shells}, \text{Shadow}\} \qquad \text{dead pets}$$
$$B = \{\text{red}, \text{blue}, \text{yellow}\} \qquad \text{primary colors}$$
$$C = \{\{a, b\}, \{a, c\}, \{b, c\}\} \qquad \text{a set of sets}$$

This works fine for small finite sets. Other sets might be defined by indicating how to generate a list of them:

$$D ::= \{1, 2, 4, 8, 16, \ldots\} \qquad \text{the powers of 2}$$

The order of elements is not significant, so $\{x, y\}$ and $\{y, x\}$ are the same set written two different ways. Also, any object is, or is not, an element of a given set—there is no notion of an element appearing more than once in a set.[1] So, writing $\{x, x\}$ is just indicating the same thing twice: that x is in the set. In particular, $\{x, x\} = \{x\}$.

The expression $e \in S$ asserts that e is an element of set S. For example, $32 \in D$ and blue $\in B$, but Tailspin $\notin A$—yet.

Sets are simple, flexible, and everywhere. You'll find some set mentioned in nearly every section of this text.

[1] It's not hard to develop a notion of *multisets* in which elements can occur more than once, but multisets are not ordinary sets and are not covered in this text.

4.1.1 Some Popular Sets

Mathematicians have devised special symbols to represent some common sets.

symbol	set	elements
$\emptyset$	the empty set	none
$\mathbb{N}$	nonnegative integers	$\{0, 1, 2, 3, \ldots\}$
$\mathbb{Z}$	integers	$\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
$\mathbb{Q}$	rational numbers	$\frac{1}{2},\ -\frac{5}{3},\ 16,$ etc.
$\mathbb{R}$	real numbers	$\pi,\ e,\ -9,\ \sqrt{2},$ etc.
$\mathbb{C}$	complex numbers	$i,\ \frac{19}{2},\ \sqrt{2} - 2i,$ etc.

A superscript "$+$" restricts a set to its positive elements; for example, $\mathbb{R}^+$ denotes the set of positive real numbers. Similarly, $\mathbb{Z}^-$ denotes the set of negative integers.

4.1.2 Comparing and Combining Sets

The expression $S \subseteq T$ indicates that set S is a *subset* of set T, which means that every element of S is also an element of T. For example, $\mathbb{N} \subseteq \mathbb{Z}$ because every nonnegative integer is an integer; $\mathbb{Q} \subseteq \mathbb{R}$ because every rational number is a real number, but $\mathbb{C} \not\subseteq \mathbb{R}$ because not every complex number is a real number.

As a memory trick, think of the "$\subseteq$" symbol as like the "$\leq$" sign with the smaller set or number on the left hand side. Notice that just as $n \leq n$ for any number n, also $S \subseteq S$ for any set S.

There is also a relation, $\subset$, on sets like the "less than" relation $<$ on numbers. $S \subset T$ means that S is a subset of T, but the two are *not* equal. So just as $n \not< n$ for every number n, also $A \not\subset A$, for every set A. "$S \subset T$" is read as "S is a *strict subset* of T."

There are several basic ways to combine sets. For example, suppose

$$X ::= \{1, 2, 3\},$$
$$Y ::= \{2, 3, 4\}.$$

Definition 4.1.1.

- The *union* of sets A and B, denoted $A \cup B$, includes exactly the elements appearing in A or B or both. That is,

$$x \in A \cup B \quad \text{IFF} \quad x \in A \text{ OR } x \in B.$$

So $X \cup Y = \{1, 2, 3, 4\}$.

- The *intersection* of A and B, denoted $A \cap B$, consists of all elements that appear in *both* A and B. That is,

$$x \in A \cap B \quad \text{IFF} \quad x \in A \text{ AND } x \in B.$$

  So, $X \cap Y = \{2, 3\}$.

- The *set difference* of A and B, denoted $A - B$, consists of all elements that are in A, but not in B. That is,

$$x \in A - B \quad \text{IFF} \quad x \in A \text{ AND } x \notin B.$$

  So, $X - Y = \{1\}$ and $Y - X = \{4\}$.

Often all the sets being considered are subsets of a known domain of discourse, D. Then for any subset, A, of D, we define $\overline{A}$ to be the set of all elements of D *not* in A. That is,

$$\overline{A} ::= D - A.$$

The set $\overline{A}$ is called the *complement* of A. So

$$\overline{A} = \emptyset \text{ IFF } A = D.$$

For example, if the domain we're working with is the integers, the complement of the nonnegative integers is the set of negative integers:

$$\overline{\mathbb{N}} = \mathbb{Z}^{-}.$$

We can use complement to rephrase subset in terms of equality

$$A \subseteq B \text{ is equivalent to } A \cap \overline{B} = \emptyset.$$

4.1.3 Power Set

The set of all the subsets of a set, A, is called the *power set*, $\text{pow}(A)$, of A. So

$$B \in \text{pow}(A) \quad \text{IFF} \quad B \subseteq A.$$

For example, the elements of $\text{pow}(\{1, 2\})$ are $\emptyset$, $\{1\}$, $\{2\}$ and $\{1, 2\}$.

More generally, if A has n elements, then there are 2^n sets in $\text{pow}(A)$—see Theorem 4.5.5. For this reason, some authors use the notation 2^A instead of $\text{pow}(A)$.

4.1.4 Set Builder Notation

An important use of predicates is in *set builder notation*. We'll often want to talk about sets that cannot be described very well by listing the elements explicitly or by taking unions, intersections, etc., of easily described sets. Set builder notation often comes to the rescue. The idea is to define a *set* using a *predicate*; in particular, the set consists of all values that make the predicate true. Here are some examples of set builder notation:

$$A ::= \{n \in \mathbb{N} \mid n \text{ is a prime and } n = 4k + 1 \text{ for some integer } k\}$$
$$B ::= \{x \in \mathbb{R} \mid x^3 - 3x + 1 > 0\}$$
$$C ::= \{a + bi \in \mathbb{C} \mid a^2 + 2b^2 \leq 1\}$$

The set A consists of all nonnegative integers n for which the predicate

"n is a prime and $n = 4k + 1$ for some integer k"

is true. Thus, the smallest elements of A are:

$$5, 13, 17, 29, 37, 41, 53, 61, 73, \ldots.$$

Trying to indicate the set A by listing these first few elements wouldn't work very well; even after ten terms, the pattern is not obvious! Similarly, the set B consists of all real numbers x for which the predicate

$$x^3 - 3x + 1 > 0$$

is true. In this case, an explicit description of the set B in terms of intervals would require solving a cubic equation. Finally, set C consists of all complex numbers $a + bi$ such that:

$$a^2 + 2b^2 \leq 1$$

This is an oval-shaped region around the origin in the complex plane.

4.1.5 Proving Set Equalities

Two sets are defined to be equal if they have exactly the same elements. That is, $X = Y$ means that $z \in X$ if and only if $z \in Y$, for all elements, z.[2] So, set equalities can be formulated and proved as "iff" theorems. For example:

[2]This is actually the first of the ZFC axioms for set theory mentioned at the end of Section 1.3 and discussed further in Section 7.3.2.

Theorem 4.1.2. *[*Distributive Law *for Sets] Let A, B, and C be sets. Then:*

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \tag{4.1}$$

Proof. The equality (4.1) is equivalent to the assertion that

$$z \in A \cap (B \cup C) \quad \text{iff} \quad z \in (A \cap B) \cup (A \cap C) \tag{4.2}$$

for all z. Now we'll prove (4.2) by a chain of iff's.

Now we have

$z \in A \cap (B \cup C)$

iff $(z \in A)$ AND $(z \in B \cup C)$ (def of $\cap$)

iff $(z \in A)$ AND $(z \in B$ OR $z \in C)$ (def of $\cup$)

iff $(z \in A$ AND $z \in B)$ OR $(z \in A$ AND $z \in C)$ (AND distributivity Thm 3.4.1)

iff $(z \in A \cap B)$ OR $(z \in A \cap C)$ (def of $\cap$)

iff $z \in (A \cap B) \cup (A \cap C)$ (def of $\cup$)

■

Although the basic set operations and propositional connectives are similar, it's important not to confuse one with the other. For example, $\cup$ resembles OR, and in fact was defined directly in terms of OR:

$$x \in A \cup B \text{ is equivalent to } (x \in A \text{ OR } x \in B).$$

Similarly, $\cap$ resembles AND, and complement resembles NOT.

But if A and B are sets, writing A AND B is a type-error, since AND is an operation on truth-values, not sets. Similarly, if P and Q are propositional variables, writing $P \cup Q$ is another type-error.

The proof of Theorem 4.1.2 illustrates a general method for proving a set equality involving the basic set operations by checking that a corresponding propositional formula is valid. As a further example, from De Morgan's Law (3.11) for propositions

$$\text{NOT}(P \text{ AND } Q) \text{ is equivalent to } \overline{P} \text{ OR } \overline{Q}$$

we can derive (Problem 4.5) a corresponding De Morgan's Law for set equality:

$$\overline{A \cap B} = \overline{A} \cup \overline{B}. \tag{4.3}$$

Despite this correspondence between two kinds of operations, it's important not to confuse propositional operations with set operations. For example, if X and Y

are sets, then it is wrong to write "X AND Y" instead of "$X \cap Y$." Applying AND to sets will cause your compiler—or your grader—to throw a type error, because an operation that is only supposed to be applied to truth values has been applied to sets. Likewise, if P and Q are propositions, then it is a type error to write "$P \cup Q$" instead of "P OR Q."

4.2 Sequences

Sets provide one way to group a collection of objects. Another way is in a *sequence*, which is a list of objects called *terms* or *components*. Short sequences are commonly described by listing the elements between parentheses; for example, (a, b, c) is a sequence with three terms.

While both sets and sequences perform a gathering role, there are several differences.

- The elements of a set are required to be distinct, but terms in a sequence can be the same. Thus, (a, b, a) is a valid sequence of length three, but $\{a, b, a\}$ is a set with two elements, not three.
- The terms in a sequence have a specified order, but the elements of a set do not. For example, (a, b, c) and (a, c, b) are different sequences, but $\{a, b, c\}$ and $\{a, c, b\}$ are the same set.
- Texts differ on notation for the *empty sequence*; we use λ for the empty sequence.

The product operation is one link between sets and sequences. A *Cartesian product* of sets, $S_1 \times S_2 \times \cdots \times S_n$, is a new set consisting of all sequences where the first component is drawn from S_1, the second from S_2, and so forth. Length two sequences are called *pairs*.[3] For example, $\mathbb{N} \times \{a, b\}$ is the set of all pairs whose first element is a nonnegative integer and whose second element is an a or a b:

$$\mathbb{N} \times \{a, b\} = \{(0, a), (0, b), (1, a), (1, b), (2, a), (2, b), \ldots\}$$

A product of n copies of a set S is denoted S^n. For example, $\{0, 1\}^3$ is the set of all 3-bit sequences:

$$\{0, 1\}^3 = \{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$$

[3] Some texts call them *ordered pairs*.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

4.3 Functions

4.3.1 Domains and Images

A *function* assigns an element of one set, called the *domain*, to an element of another set, called the *codomain*. The notation

$$f : A \to B$$

indicates that f is a function with domain, A, and codomain, B. The familiar notation "$f(a) = b$" indicates that f assigns the element $b \in B$ to a. Here b would be called the *value* of f at *argument* a.

Functions are often defined by formulas, as in:

$$f_1(x) ::= \frac{1}{x^2}$$

where x is a real-valued variable, or

$$f_2(y, z) ::= y10yz$$

where y and z range over binary strings, or

$$f_3(x, n) ::= \text{ the length } n \text{ sequence } \underbrace{(x, \ldots, x)}_{n\ x\text{'s}}$$

where n ranges over the nonnegative integers.

A function with a finite domain could be specified by a table that shows the value of the function at each element of the domain. For example, a function $f_4(P, Q)$ where P and Q are propositional variables is specified by:

P	Q	$f_4(P, Q)$
T	**T**	**T**
T	**F**	**F**
F	**T**	**T**
F	**F**	**T**

Notice that f_4 could also have been described by a formula:

$$f_4(P, Q) ::= [P \text{ IMPLIES } Q].$$

A function might also be defined by a procedure for computing its value at any element of its domain, or by some other kind of specification. For example, define

$f_5(y)$ to be the length of a left to right search of the bits in the binary string y until a 1 appears, so

$$\begin{aligned} f_5(0010) &= 3, \\ f_5(100) &= 1, \\ f_5(0000) &\text{ is undefined.} \end{aligned}$$

Notice that f_5 does not assign a value to any string of just 0's. This illustrates an important fact about functions: they need not assign a value to every element in the domain. In fact this came up in our first example $f_1(x) = 1/x^2$, which does not assign a value to 0. So in general, functions may be *partial functions*, meaning that there may be domain elements for which the function is not defined. If a function is defined on every element of its domain, it is called a *total function*.

It's often useful to find the set of values a function takes when applied to the elements in *a set* of arguments. So if $f : A \to B$, and S is a subset of A, we define $f(S)$ to be the set of all the values that f takes when it is applied to elements of S. That is,

$$f(S) ::= \{b \in B \mid f(s) = b \text{ for some } s \in S\}.$$

For example, if we let $[r, s]$ denote set of numbers in the interval from r to s on the real line, then $f_1([1, 2]) = [1/4, 1]$.

For another example, let's take the "search for a 1" function, f_5. If we let X be the set of binary words which start with an even number of 0's followed by a 1, then $f_5(X)$ would be the odd nonnegative integers.

Applying f to a set, S, of arguments is referred to as "applying f *pointwise* to S", and the set $f(S)$ is referred to as the *image* of S under f.[4] The set of values that arise from applying f to all possible arguments is called the *range* of f. That is,

$$\text{range}(f) ::= f(\text{domain}(f)).$$

Some authors refer to the codomain as the range of a function, but they shouldn't. The distinction between the range and codomain will be important later in Sections 4.5 when we relate sizes of sets to properties of functions between them.

4.3.2 Function Composition

Doing things step by step is a universal idea. Taking a walk is a literal example, but so is cooking from a recipe, executing a computer program, evaluating a formula, and recovering from substance abuse.

[4] There is a picky distinction between the function f which applies to elements of A and the function which applies f pointwise to subsets of A, because the domain of f is A, while the domain of pointwise-f is $\text{pow}(A)$. It is usually clear from context whether f or pointwise-f is meant, so there is no harm in overloading the symbol f in this way.

Abstractly, taking a step amounts to applying a function, and going step by step corresponds to applying functions one after the other. This is captured by the operation of *composing* functions. Composing the functions f and g means that first f is applied to some argument, x, to produce $f(x)$, and then g is applied to that result to produce $g(f(x))$.

Definition 4.3.1. For functions $f : A \to B$ and $g : B \to C$, the *composition*, $g \circ f$, of g with f is defined to be the function from A to C defined by the rule:

$$(g \circ f)(x) ::= g(f(x)),$$

for all $x \in A$.

Function composition is familiar as a basic concept from elementary calculus, and it plays an equally basic role in discrete mathematics.

4.4 Binary Relations

Binary relations define relations between two objects. For example, "less-than" on the real numbers relates every real number, a, to a real number, b, precisely when $a < b$. Similarly, the subset relation relates a set, A, to another set, B, precisely when $A \subseteq B$. A function $f : A \to B$ is a special case of binary relation in which an element $a \in A$ is related to an element $b \in B$ precisely when $b = f(a)$.

In this section we'll define some basic vocabulary and properties of binary relations.

Definition 4.4.1. A *binary relation*, R, consists of a set, A, called the *domain* of R, a set, B, called the *codomain* of R, and a subset of $A \times B$ called the *graph of* R.

A relation whose domain is A and codomain is B is said to be "between A and B", or "from A to B." As with functions, we write $R : A \to B$ to indicate that R is a relation from A to B. When the domain and codomain are the same set, A, we simply say the relation is "on A." It's common to use "$a\ R\ b$" to mean that the pair (a, b) is in the graph of R.[5]

Notice that Definition 4.4.1 is exactly the same as the definition in Section 4.3 of a *function*, except that it doesn't require the functional condition that, for each

[5]Writing the relation or operator symbol between its arguments is called *infix notation*. Infix expressions like "$m < n$" or "$m + n$" are the usual notation used for things like the less-then relation or the addition operation rather than prefix notation like "$< (m, n)$" or "$+(m, n)$."

domain element, a, there is *at most* one pair in the graph whose first coordinate is a. As we said, a function is a special case of a binary relation.

The "in-charge of" relation, *Chrg*, for MIT in Spring '10 subjects and instructors is a handy example of a binary relation. Its domain, Fac, is the names of all the MIT faculty and instructional staff, and its codomain is the set, SubNums, of subject numbers in the Fall '09–Spring '10 MIT subject listing. The graph of *Chrg* contains precisely the pairs of the form

$$(\langle\text{instructor-name}\rangle\,,\langle\text{subject-num}\rangle)$$

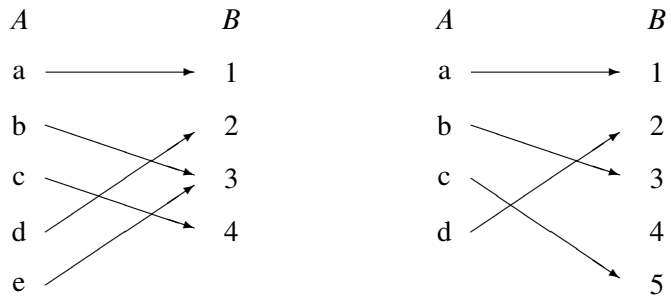
such that the faculty member named ⟨instructor-name⟩ is in charge of the subject with number ⟨subject-num⟩ that was offered in Spring '10. So graph(*Chrg*) contains pairs like

```
(T. Eng,       6.UAT)
(G. Freeman,   6.011)
(G. Freeman,   6.UAT)
(G. Freeman,   6.881)
(G. Freeman,   6.882)
(J. Guttag,    6.00)
(A. R. Meyer,  6.042)
(A. R. Meyer,  18.062)
(A. R. Meyer,  6.844)
(T. Leighton,  6.042)
(T. Leighton,  18.062)
    ⋮
```
(4.4)

Some subjects in the codomain, SubNums, do not appear among this list of pairs—that is, they are not in range(*Chrg*). These are the Fall term-only subjects. Similarly, there are instructors in the domain, Fac, who do not appear in the list because they are not in charge of any Spring term subjects.

4.4.1 Relation Diagrams

Some standard properties of a relation can be visualized in terms of a diagram. The diagram for a binary relation, R, has points corresponding to the elements of the domain appearing in one column (a very long column if domain(R) is infinite). All the elements of the codomain appear in another column which we'll usually picture as being to the right of the domain column. There is an arrow going from a point, a, in the lefthand, domain column to a point, b, in the righthand, codomain column, precisely when the corresponding elements are related by R. For example, here are diagrams for two functions:



Being a function is certainly an important property of a binary relation. What it means is that every point in the domain column has *at most one arrow coming out of it*. So we can describe being a function as the "≤ 1 arrow out" property. There are four more standard properties of relations that come up all the time. Here are all five properties defined in terms of arrows:

Definition 4.4.2. A binary relation, R, is:

- a *function* when it has the [≤ 1 arrow **out**] property.
- *surjective* when it has the [≥ 1 arrows **in**] property. That is, every point in the righthand, codomain column has at least one arrow pointing to it.
- *total* when it has the [≥ 1 arrows **out**] property.
- *injective* when it has the [≤ 1 arrow **in**] property.
- *bijective* when it has both the [$= 1$ arrow **out**] and the [$= 1$ arrow **in**] property.

From here on, we'll stop mentioning the arrows in these properties and for example, just write [≤ 1 in] instead of [≤ 1 arrows in].

So in the diagrams above, the relation on the left has the [$= 1$ out] and [≥ 1 in] properties, which means it is a total, surjective function. But it does not have the [≤ 1 in] property because element 3 has two arrows going into it; it is not injective.

The relation on the right has the [$= 1$ out] and [≤ 1 in] properties, which means it is a total, injective function. But it does not have the [≥ 1 in] property because element 4 has no arrow going into it; it is not surjective.

The arrows in a diagram for R correspond, of course, exactly to the pairs in the graph of R. Notice that the arrows alone are not enough to determine, for example, if R has the [≥ 1 out], total, property. If all we knew were the arrows, we wouldn't know about any points in the domain column that had no arrows out. In other words, graph(R) alone does not determine whether R is total: we also need to know what domain(R) is.

Example 4.4.3. The function defined by the formula $1/x^2$ has the [$\geq$ 1 out] property if its domain is $\mathbb{R}^+$, but not if its domain is some set of real numbers including 0. It has the [$=$ 1 in] and [$=$ 1 out] property if its domain and codomain are both $\mathbb{R}^+$, but it has neither the [$\leq$ 1 in] nor the [$\geq$ 1 out] property if its domain and codomain are both $\mathbb{R}$.

4.4.2 Relational Images

The idea of the image of a set under a function extends directly to relations.

Definition 4.4.4. The *image* of a set, Y, under a relation, R, written $R(Y)$, is the set of elements of the codomain, B, of R that are related to some element in Y. In terms of the relation diagram, $R(Y)$ is the set of points with an arrow coming in that starts from some point in Y.

For example, the set of subject numbers that Meyer is in charge of in Spring '10 is exactly *Chrg*(A. Meyer). To figure out what this is, we look for all the arrows in the *Chrg* diagram that start at "A. Meyer," and see which subject-numbers are at the other end of these arrows. Looking at the list (4.4) of pairs in graph(*Chrg*), we see that these subject-numbers are {6.042, 18.062, 6.844}. Similarly, to find the subject numbers that either Freeman or Eng are in charge of, we can collect all the arrows that start at either "G. Freeman," or "T. Eng" and, again, see which subject-numbers are at the other end of these arrows. This is *Chrg*({G. Freeman, T. Eng}). Looking again at the list (4.4), we see that

$$Chrg(\{\text{G. Freeman, T. Eng}\}) = \{6.011, 6.881, 6.882, 6.\text{UAT}\}$$

Finally, Fac is the set of all in-charge instructors, so *Chrg*(Fac) is the set of all the subjects listed for Spring '10.

Inverse Relations and Images

Definition 4.4.5. The *inverse*, R^{-1} of a relation $R : A \to B$ is the relation from B to A defined by the rule

$$b\ R^{-1}\ a \quad \text{IFF} \quad a\ R\ b.$$

In other words, R^{-1} is the relation you get by reversing the direction of the arrows in the diagram of R.

Definition 4.4.6. The image of a set under the relation, R^{-1}, is called the *inverse image* of the set. That is, the inverse image of a set, X, under the relation, R, is defined to be $R^{-1}(X)$.

Continuing with the in-charge example above, the set of instructors in charge of 6.UAT in Spring '10 is exactly the inverse image of {6.UAT} under the *Chrg* relation. From the list (4.4), we see that Eng and Freeman are both in charge of 6.UAT, that is,

$$\{\text{T. Eng, D. Freeman}\} \subseteq Chrg^{-1}(\{\text{6.UAT}\}).$$

We can't assert equality here because there may be additional pairs further down the list showing that additional instructors are co-incharge of 6.UAT.

Now let Intro be the set of introductory course 6 subject numbers. These are the subject numbers that start with "6.0." So the set of names of the instructors who were in-charge of introductory course 6 subjects in Spring '10, is $Chrg^{-1}(\text{Intro})$. From the part of the *Chrg* list shown in (4.4), we see that Meyer, Leighton, Freeman, and Guttag were among the instructors in charge of introductory subjects in Spring '10. That is,

$$\{\text{Meyer, Leighton, Freeman, Guttag}\} \subseteq Chrg^{-1}(\text{Intro}).$$

Finally, $Chrg^{-1}(\text{SubNums})$, is the set of all instructors who were in charge of a subject listed for Spring '10.

4.5 Finite Cardinality

A finite set is one that has only a finite number of elements. This number of elements is the "size" or *cardinality* of the set:

Definition 4.5.1. If A is a finite set, the *cardinality* of A, written $|A|$, is the number of elements in A.

A finite set may have no elements (the empty set), or one element, or two elements,..., so the cardinality of finite sets is always a nonnegative integer.

Now suppose $R : A \to B$ is a function. This means that every element of A contributes at most one arrow to the diagram for R, so the number of arrows is at most the number of elements in A. That is, if R is a function, then

$$|A| \geq \#\text{arrows}.$$

If R is also surjective, then every element of B has an arrow into it, so there must be at least as many arrows in the diagram as the size of B. That is,

$$\#\text{arrows} \geq |B|.$$

Combining these inequalities implies that if R is a surjective function, then $|A| \geq |B|$.

In short, if we write A surj B to mean that there is a surjective function from A to B, then we've just proved a lemma: if A surj B for finite sets A, B, then $|A| \geq |B|$. The following definition and lemma lists this statement and three similar rules relating domain and codomain size to relational properties.

Definition 4.5.2. Let A, B be (not necessarily finite) sets. Then

1. A surj B iff there is a surjective *function* from A to B.
2. A inj B iff there is an injective *total* relation from A to B.
3. A bij B iff there is a bijection from A to B.

Lemma 4.5.3. *For finite sets A, B:*

1. *If A surj B, then $|A| \geq |B|$.*
2. *If A inj B, then $|A| \leq |B|$.*
3. *If A bij B, then $|A| = |B|$.*

Proof. We've already given an "arrow" proof of implication 1. Implication 2. follows immediately from the fact that if R has the [≤ 1 out], function property, and the [≥ 1 in], surjective property, then R^{-1} is total and injective, so A surj B iff B inj A. Finally, since a bijection is both a surjective function and a total injective relation, implication 3. is an immediate consequence of the first two. ■

Lemma 4.5.3.1. has a converse: if the size of a finite set, A, is greater than or equal to the size of another finite set, B, then it's always possible to define a surjective function from A to B. In fact, the surjection can be a total function. To see how this works, suppose for example that

$$A = \{a_0, a_1, a_2, a_3, a_4, a_5\}$$
$$B = \{b_0, b_1, b_2, b_3\}.$$

Then define a total function $f : A \to B$ by the rules

$$f(a_0) ::= b_0,\ f(a_1) ::= b_1,\ f(a_2) ::= b_2,\ f(a_3) = f(a_4) = f(a_5) ::= b_3.$$

More concisely,

$$f(a_i) ::= b_{\min(i,3)},$$

for $0 \leq i \leq 5$. Since $5 \geq 3$, this f is a surjection.

So we have figured out that if A and B are finite sets, then $|A| \geq |B|$ *if and only if* A surj B. All told, this argument wraps up the proof of a theorem that summarizes the whole finite cardinality story:

Theorem 4.5.4. *[Mapping Rules] For* finite *sets, A, B,*

$$|A| \geq |B| \quad \textit{iff} \quad A \text{ surj } B, \tag{4.5}$$
$$|A| \leq |B| \quad \textit{iff} \quad A \text{ inj } B, \tag{4.6}$$
$$|A| = |B| \quad \textit{iff} \quad A \text{ bij } B, \tag{4.7}$$

4.5.1 How Many Subsets of a Finite Set?

As an application of the bijection mapping rule (4.7), we can give an easy proof of:

Theorem 4.5.5. *There are 2^n subsets of an n-element set. That is,*

$$|A| = n \quad \textit{implies} \quad |\operatorname{pow}(A)| = 2^n.$$

For example, the three-element set $\{a_1, a_2, a_3\}$ has eight different subsets:

$$\begin{array}{cccc} \emptyset & \{a_1\} & \{a_2\} & \{a_1, a_2\} \\ \{a_3\} & \{a_1, a_3\} & \{a_2, a_3\} & \{a_1, a_2, a_3\} \end{array}$$

Theorem 4.5.5 follows from the fact that there is a simple bijection from subsets of A to $\{0,1\}^n$, the n-bit sequences. Namely, let $a_1, a_2, \ldots, a_n$ be the elements of A. The bijection maps each subset of $S \subseteq A$ to the bit sequence $(b_1, \ldots, b_n)$ defined by the rule that

$$b_i = 1 \quad \text{iff} \quad a_i \in S.$$

For example, if $n = 10$, then the subset $\{a_2, a_3, a_5, a_7, a_{10}\}$ maps to a 10-bit sequence as follows:

$$\begin{array}{rcccccccccc} \text{subset:} & \{ & a_2, & a_3, & & a_5, & & a_7, & & & a_{10} & \} \\ \text{sequence:} & (& 0, & 1, & 1, & 0, & 1, & 0, & 1, & 0, & 0, & 1 &) \end{array}$$

Now by bijection case of the Mapping Rules 4.5.4.(4.7),

$$|\operatorname{pow}(A)| = |\{0,1\}^n|.$$

But every computer scientist knows[6] that there are 2^n n-bit sequences! So we've proved Theorem 4.5.5!

[6] In case you're someone who doesn't know how many n-bit sequences there are, you'll find the 2^n explained in Section 14.2.2.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

5 Induction

Induction is a powerful method for showing a property is true for all nonnegative integers. Induction plays a central role in discrete mathematics and computer science. In fact, its use is a defining characteristic of *discrete*—as opposed to *continuous*—mathematics. This chapter introduces two versions of induction, Ordinary and Strong, and explains why they work and how to use them in proofs. It also introduces the Invariant Principle, which is a version of induction specially adapted for reasoning about step-by-step processes.

5.1 Ordinary Induction

To understand how induction works, suppose there is a professor who brings a bottomless bag of assorted miniature candy bars to her large class. She offers to share the candy in the following way. First, she lines the students up in order. Next she states two rules:

1. The student at the beginning of the line gets a candy bar.
2. If a student gets a candy bar, then the following student in line also gets a candy bar.

Let's number the students by their order in line, starting the count with 0, as usual in computer science. Now we can understand the second rule as a short description of a whole sequence of statements:

- If student 0 gets a candy bar, then student 1 also gets one.
- If student 1 gets a candy bar, then student 2 also gets one.
- If student 2 gets a candy bar, then student 3 also gets one.

$$\vdots$$

Of course, this sequence has a more concise mathematical description:

> If student n gets a candy bar, then student $n + 1$ gets a candy bar, for all nonnegative integers n.

So suppose you are student 17. By these rules, are you entitled to a miniature candy bar? Well, student 0 gets a candy bar by the first rule. Therefore, by the second rule, student 1 also gets one, which means student 2 gets one, which means student 3 gets one as well, and so on. By 17 applications of the professor's second rule, you get your candy bar! Of course the rules really guarantee a candy bar to *every* student, no matter how far back in line they may be.

5.1.1 A Rule for Ordinary Induction

The reasoning that led us to conclude that every student gets a candy bar is essentially all there is to induction.

The Induction Principle.

Let P be a predicate on nonnegative integers. If

- $P(0)$ is true, and
- $P(n)$ IMPLIES $P(n+1)$ for all nonnegative integers, n,

then

- $P(m)$ is true for all nonnegative integers, m.

Since we're going to consider several useful variants of induction in later sections, we'll refer to the induction method described above as *ordinary induction* when we need to distinguish it. Formulated as a proof rule as in Section 1.4.1, this would be

Rule. ***Induction Rule***

$$\frac{P(0), \quad \forall n \in \mathbb{N}.\, P(n) \text{ IMPLIES } P(n+1)}{\forall m \in \mathbb{N}.\, P(m)}$$

This Induction Rule works for the same intuitive reason that all the students get candy bars, and we hope the explanation using candy bars makes it clear why the soundness of ordinary induction can be taken for granted. In fact, the rule is so obvious that it's hard to see what more basic principle could be used to justify it.[1] What's not so obvious is how much mileage we get by using it.

[1] But see Section 5.3.

5.1.2 A Familiar Example

Below is the formula (5.1) for the sum of the nonnegative integers up to n. The formula holds for all nonnegative integers, so it is the kind of statement to which induction applies directly. We've already proved this formula using the Well Ordering Principle (Theorem 2.2.1), but now we'll prove it *by induction*, that is, using the Induction Principle.

Theorem 5.1.1. *For all $n \in \mathbb{N}$,*

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \tag{5.1}$$

To prove the theorem by induction, define predicate $P(n)$ to be the equation (5.1). Now the theorem can be restated as the claim that $P(n)$ is true for all $n \in \mathbb{N}$. This is great, because the Induction Principle lets us reach precisely that conclusion, provided we establish two simpler facts:

- $P(0)$ is true.
- For all $n \in \mathbb{N}$, $P(n)$ IMPLIES $P(n+1)$.

So now our job is reduced to proving these two statements.

The first statement follows because of the convention that a sum of zero terms is equal to 0. So $P(0)$ is the true assertion that a sum of zero terms is equal to $0(0+1)/2 = 0$.

The second statement is more complicated. But remember the basic plan from Section 1.5 for proving the validity of any implication: *assume* the statement on the left and then *prove* the statement on the right. In this case, we assume $P(n)$—namely, equation (5.1)—in order to prove $P(n+1)$, which is the equation

$$1 + 2 + 3 + \cdots + n + (n+1) = \frac{(n+1)(n+2)}{2}. \tag{5.2}$$

These two equations are quite similar; in fact, adding $(n+1)$ to both sides of equation (5.1) and simplifying the right side gives the equation (5.2):

$$\begin{aligned} 1 + 2 + 3 + \cdots + n + (n+1) &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{(n+2)(n+1)}{2} \end{aligned}$$

Thus, if $P(n)$ is true, then so is $P(n+1)$. This argument is valid for every nonnegative integer n, so this establishes the second fact required by the induction proof. Therefore, the Induction Principle says that the predicate $P(m)$ is true for all nonnegative integers, m. The theorem is proved.

5.1.3 A Template for Induction Proofs

The proof of equation (5.1) was relatively simple, but even the most complicated induction proof follows exactly the same template. There are five components:

1. **State that the proof uses induction.** This immediately conveys the overall structure of the proof, which helps your reader follow your argument.

2. **Define an appropriate predicate $P(n)$.** The predicate $P(n)$ is called the *induction hypothesis*. The eventual conclusion of the induction argument will be that $P(n)$ is true for all nonnegative n. A clearly stated induction hypothesis is often the most important part of an induction proof, and its omission is the largest source of confused proofs by students.

   In the simplest cases, the induction hypothesis can be lifted straight from the proposition you are trying to prove, as we did with equation (5.1). Sometimes the induction hypothesis will involve several variables, in which case you should indicate which variable serves as n.

3. **Prove that $P(0)$ is true.** This is usually easy, as in the example above. This part of the proof is called the *base case* or *basis step*.

4. **Prove that $P(n)$ implies $P(n+1)$ for every nonnegative integer n.** This is called the *inductive step*. The basic plan is always the same: assume that $P(n)$ is true and then use this assumption to prove that $P(n+1)$ is true. These two statements should be fairly similar, but bridging the gap may require some ingenuity. Whatever argument you give must be valid for every nonnegative integer n, since the goal is to prove that *all* the following implications are true:

   $$P(0) \to P(1),\ P(1) \to P(2),\ P(2) \to P(3), \ldots.$$

5. **Invoke induction.** Given these facts, the induction principle allows you to conclude that $P(n)$ is true for all nonnegative n. This is the logical capstone to the whole argument, but it is so standard that it's usual not to mention it explicitly.

Always be sure to explicitly label the *base case* and the *inductive step*. Doing so will make your proofs clearer and will decrease the chance that you forget a key step—like checking the base case.

5.1.4 A Clean Writeup

The proof of Theorem 5.1.1 given above is perfectly valid; however, it contains a lot of extraneous explanation that you won't usually see in induction proofs. The writeup below is closer to what you might see in print and should be prepared to produce yourself.

Revised proof of Theorem 5.1.1. We use induction. The induction hypothesis, $P(n)$, will be equation (5.1).

Base case: $P(0)$ is true, because both sides of equation (5.1) equal zero when $n = 0$.

Inductive step: Assume that $P(n)$ is true, that is equation (5.1) holds for some nonnegative integer n. Then adding $n + 1$ to both sides of the equation implies that

$$\begin{aligned} 1 + 2 + 3 + \cdots + n + (n+1) &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{(n+1)(n+2)}{2} \qquad \text{(by simple algebra)} \end{aligned}$$

which proves $P(n+1)$.

So it follows by induction that $P(n)$ is true for all nonnegative n. ■

It probably bothers you that induction led to a proof of this summation formula but did not provide an intuitive way to understand it nor did it explain where the formula came from in the first place.[2] This is both a weakness and a strength. It is a weakness when a proof does not provide insight. But it is a strength that a proof can provide a reader with a reliable guarantee of correctness without *requiring* insight.

5.1.5 A More Challenging Example

During the development of MIT's famous Stata Center, as costs rose further and further beyond budget, some radical fundraising ideas were proposed. One rumored plan was to install a big square courtyard divided into unit squares. The big square would be 2^n units on a side for some undetermined nonnegative integer n, and one of the unit squares in the center[3] occupied by a statue of a wealthy potential donor—whom the fund raisers privately referred to as "Bill." The $n = 3$ case is shown in Figure 5.1.

A complication was that the building's unconventional architect, Frank Gehry, was alleged to require that only special L-shaped tiles (shown in Figure 5.2) be

[2] Methods for finding such formulas are covered in Part III of the text.

[3] In the special case $n = 0$, the whole courtyard consists of a single central square; otherwise, there are four central squares.

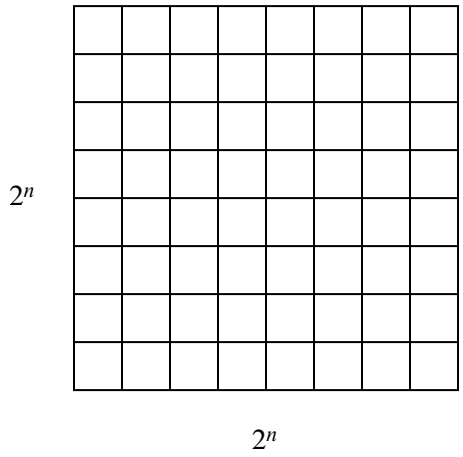


Figure 5.1 A $2^n \times 2^n$ courtyard for $n = 3$.

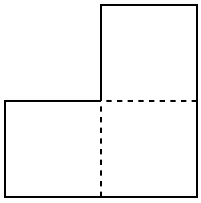


Figure 5.2 The special L-shaped tile.

used for the courtyard. For $n = 2$, a courtyard meeting these constraints is shown in Figure 5.3. But what about for larger values of n? Is there a way to tile a $2^n \times 2^n$ courtyard with L-shaped tiles around a statue in the center? Let's try to prove that this is so.

Theorem 5.1.2. *For all $n \geq 0$ there exists a tiling of a $2^n \times 2^n$ courtyard with Bill in a central square.*

Proof. (doomed attempt) The proof is by induction. Let $P(n)$ be the proposition that there exists a tiling of a $2^n \times 2^n$ courtyard with Bill in the center.

Base case: $P(0)$ is true because Bill fills the whole courtyard.

Inductive step: Assume that there is a tiling of a $2^n \times 2^n$ courtyard with Bill in the center for some $n \geq 0$. We must prove that there is a way to tile a $2^{n+1} \times 2^{n+1}$ courtyard with Bill in the center .... ■

Now we're in trouble! The ability to tile a smaller courtyard with Bill in the

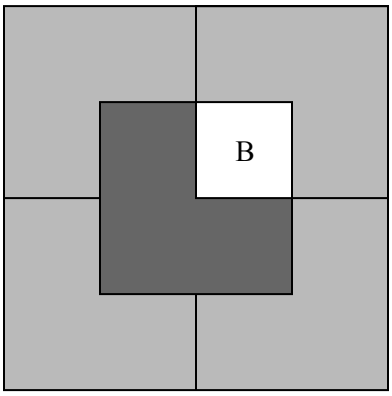


Figure 5.3 A tiling using L-shaped tiles for $n = 2$ with Bill in a center square.

center isn't much help in tiling a larger courtyard with Bill in the center. We haven't figured out how to bridge the gap between $P(n)$ and $P(n+1)$.

So if we're going to prove Theorem 5.1.2 by induction, we're going to need some *other* induction hypothesis than simply the statement about n that we're trying to prove.

When this happens, your first fallback should be to look for a *stronger* induction hypothesis; that is, one which implies your previous hypothesis. For example, we could make $P(n)$ the proposition that for *every* location of Bill in a $2^n \times 2^n$ courtyard, there exists a tiling of the remainder.

This advice may sound bizarre: "If you can't prove something, try to prove something grander!" But for induction arguments, this makes sense. In the inductive step, where you have to prove $P(n)$ IMPLIES $P(n+1)$, you're in better shape because you can *assume* $P(n)$, which is now a more powerful statement. Let's see how this plays out in the case of courtyard tiling.

Proof (successful attempt). The proof is by induction. Let $P(n)$ be the proposition that for every location of Bill in a $2^n \times 2^n$ courtyard, there exists a tiling of the remainder.

Base case: $P(0)$ is true because Bill fills the whole courtyard.

Inductive step: Assume that $P(n)$ is true for some $n \geq 0$; that is, for every location of Bill in a $2^n \times 2^n$ courtyard, there exists a tiling of the remainder. Divide the $2^{n+1} \times 2^{n+1}$ courtyard into four quadrants, each $2^n \times 2^n$. One quadrant contains Bill (**B** in the diagram below). Place a temporary Bill (**X** in the diagram) in each of the three central squares lying outside this quadrant as shown in Figure 5.4.

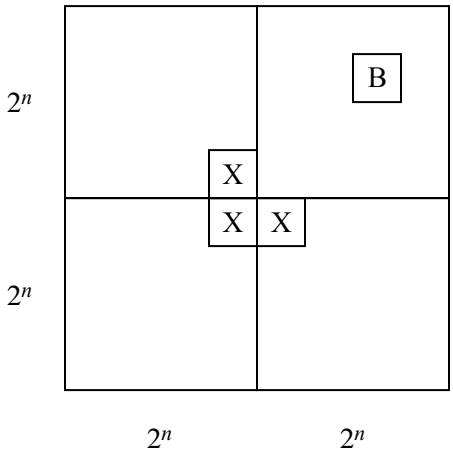


Figure 5.4 Using a stronger inductive hypothesis to prove Theorem 5.1.2.

Now we can tile each of the four quadrants by the induction assumption. Replacing the three temporary Bills with a single L-shaped tile completes the job. This proves that $P(n)$ implies $P(n+1)$ for all $n \geq 0$. Thus $P(m)$ is true for all $m \in \mathbb{N}$, and the theorem follows as a special case where we put Bill in a central square. ■

This proof has two nice properties. First, not only does the argument guarantee that a tiling exists, but also it gives an algorithm for finding such a tiling. Second, we have a stronger result: if Bill wanted a statue on the edge of the courtyard, away from the pigeons, we could accommodate him!

Strengthening the induction hypothesis is often a good move when an induction proof won't go through. But keep in mind that the stronger assertion must actually be *true*; otherwise, there isn't much hope of constructing a valid proof. Sometimes finding just the right induction hypothesis requires trial, error, and insight. For example, mathematicians spent almost twenty years trying to prove or disprove the conjecture that every planar graph is 5-choosable.[4] Then, in 1994, Carsten Thomassen gave an induction proof simple enough to explain on a napkin. The key turned out to be finding an extremely clever induction hypothesis; with that in hand, completing the argument was easy!

[4] 5-choosability is a slight generalization of 5-colorability. Although every planar graph is 4-colorable and therefore 5-colorable, not every planar graph is 4-choosable. If this all sounds like nonsense, don't panic. We'll discuss graphs, planarity, and coloring in Part II of the text.

5.1.6 A Faulty Induction Proof

If we have done a good job in writing this text, right about now you should be thinking, "Hey, this induction stuff isn't so hard after all—just show $P(0)$ is true and that $P(n)$ implies $P(n+1)$ for any number n." And, you would be right, although sometimes when you start doing induction proofs on your own, you can run into trouble. For example, we will now use induction to "prove" that all horses are the same color—just when you thought it was safe to skip class and work on your robot program instead. Sorry!

False Theorem. *All horses are the same color.*

Notice that no n is mentioned in this assertion, so we're going to have to reformulate it in a way that makes an n explicit. In particular, we'll (falsely) prove that

False Theorem 5.1.3. *In every set of $n \geq 1$ horses, all the horses are the same color.*

This is a statement about all integers $n \geq 1$ rather ≥ 0, so it's natural to use a slight variation on induction: prove $P(1)$ in the base case and then prove that $P(n)$ implies $P(n+1)$ for all $n \geq 1$ in the inductive step. This is a perfectly valid variant of induction and is *not* the problem with the proof below.

Bogus proof. The proof is by induction on n. The induction hypothesis, $P(n)$, will be

$$\text{In every set of } n \text{ horses, all are the same color.} \tag{5.3}$$

Base case: $(n = 1)$. $P(1)$ is true, because in a size-1 set of horses, there's only one horse, and this horse is definitely the same color as itself.

Inductive step: Assume that $P(n)$ is true for some $n \geq 1$. That is, assume that in every set of n horses, all are the same color. Now suppose we have a set of $n+1$ horses:

$$h_1,\ h_2,\ \ldots,\ h_n,\ h_{n+1}.$$

We need to prove these $n+1$ horses are all the same color.

By our assumption, the first n horses are the same color:

$$\underbrace{h_1,\ h_2,\ \ldots,\ h_n}_{\text{same color}}, h_{n+1}$$

Also by our assumption, the last n horses are the same color:

$$h_1,\ \underbrace{h_2,\ \ldots,\ h_n,\ h_{n+1}}_{\text{same color}}$$

So h_1 is the same color as the remaining horses besides h_{n+1} —that is, $h_2, \ldots, h_n$. Likewise, h_{n+1} is the same color as the remaining horses besides h_1—that is, $h_2, \ldots, h_n$, again. Since h_1 and h_{n+1} are the same color as $h_2, \ldots, h_n$, all $n+1$ horses must be the same color, and so $P(n+1)$ is true. Thus, $P(n)$ implies $P(n+1)$.

By the principle of induction, $P(n)$ is true for all $n \geq 1$. ■

We've proved something false! Does this mean that math broken and we should all take up poetry instead? Of course not! It just means that this proof has a mistake.

The mistake in this argument is in the sentence that begins "So h_1 is the same color as the remaining horses besides h_{n+1}—that is $h_2, \ldots, h_n, \ldots$." The ellipis notation ("...") in the expression "$h_1, h_2, \ldots, h_n, h_{n+1}$" creates the impression that there are some remaining horses—namely $h_2, \ldots, h_n$ —besides h_1 and h_{n+1}. However, this is not true when $n = 1$. In that case, $h_1, h_2, \ldots, h_n, h_{n+1}$ is just h_1, h_2 and *there are no "remaining" horses* for h_1 to share a color with. And of course, in this case h_1 and h_2 really don't need to be the same color.

This mistake knocks a critical link out of our induction argument. We proved $P(1)$ and we *correctly* proved $P(2) \longrightarrow P(3)$, $P(3) \longrightarrow P(4)$, etc. But we failed to prove $P(1) \longrightarrow P(2)$, and so everything falls apart: we cannot conclude that $P(2)$, $P(3)$, etc., are true. And naturally, these propositions are all false; there are sets of n horses of different colors for all $n \geq 2$.

Students sometimes explain that the mistake in the proof is because $P(n)$ is false for $n \geq 2$, and the proof assumes something false, $P(n)$, in order to prove $P(n+1)$. You should think about how to help such a student understand why this explanation would get no credit on a Math for Computer Science exam.

5.2 Strong Induction

A useful variant of induction is called *strong induction*. Strong induction and ordinary induction are used for exactly the same thing: proving that a predicate is true for all nonnegative integers. Strong induction is useful when a simple proof that the predicate holds for $n+1$ does not follow just from the fact that it holds at n, but from the fact that it holds for other values $\leq n$.

5.2.1 A Rule for Strong Induction

Principle of Strong Induction.

Let P be a predicate on nonnegative integers. If

- $P(0)$ is true, and
- for all $n \in \mathbb{N}$, $P(0)$, $P(1)$, ..., $P(n)$ *together* imply $P(n+1)$,

then $P(m)$ is true for all $m \in \mathbb{N}$.

The only change from the ordinary induction principle is that strong induction allows you make more assumptions in the inductive step of your proof! In an ordinary induction argument, you assume that $P(n)$ is true and try to prove that $P(n+1)$ is also true. In a strong induction argument, you may assume that $P(0)$, $P(1)$, ..., and $P(n)$ are *all* true when you go to prove $P(n+1)$. So you can assume a *stronger* set of hypotheses which can make your job easier.

Formulated as a proof rule, strong induction is

Rule. ***Strong Induction Rule***

$$\frac{P(0), \quad \forall n \in \mathbb{N}.\ \big(P(0) \text{ AND } P(1) \text{ AND } \ldots \text{ AND } P(n)\big) \text{ IMPLIES } P(n+1)}{\forall m \in \mathbb{N}.\ P(m)}$$

Stated more succintly, the rule is

Rule.

$$\frac{P(0), \quad [\forall k \leq n \in \mathbb{N}.\ P(k)] \text{ IMPLIES } P(n+1)}{\forall m \in \mathbb{N}.\ P(m)}$$

The template for strong induction proofs is identical to the template given in Section 5.1.3 for ordinary induction except for two things:

- you should state that your proof is by strong induction, and
- you can assume that $P(0)$, $P(1)$, ..., $P(n)$ are all true instead of only $P(n)$ during the inductive step.

5.2.2 Products of Primes

As a first example, we'll use strong induction to re-prove Theorem 2.3.1 which we previously proved using Well Ordering.

Theorem. *Every integer greater than 1 is a product of primes.*

Proof. We will prove the Theorem by strong induction, letting the induction hypothesis, $P(n)$, be

$$n \text{ is a product of primes.}$$

So the Theorem will follow if we prove that $P(n)$ holds for all $n \geq 2$.

Base Case: ($n = 2$): $P(2)$ is true because 2 is prime, so it is a length one product of primes by convention.

Inductive step: Suppose that $n \geq 2$ and that every number from 2 to n is a product of primes. We must show that $P(n+1)$ holds, namely, that $n+1$ is also a product of primes. We argue by cases:

If $n+1$ is itself prime, then it is a length one product of primes by convention, and so $P(n+1)$ holds in this case.

Otherwise, $n+1$ is not prime, which by definition means $n+1 = k \cdot m$ for some integers k, m between 2 and n. Now by the strong induction hypothesis, we know that both k and m are products of primes. By multiplying these products, it follows immediately that $k \cdot m = n+1$ is also a product of primes. Therefore, $P(n+1)$ holds in this case as well.

So $P(n+1)$ holds in any case, which completes the proof by strong induction that $P(n)$ holds for all $n \geq 2$.

■

5.2.3 Making Change

The country Inductia, whose unit of currency is the Strong, has coins worth 3Sg (3 Strongs) and 5Sg. Although the Inductians have some trouble making small change like 4Sg or 7Sg, it turns out that they can collect coins to make change for any number that is at least 8 Strongs.

Strong induction makes this easy to prove for $n+1 \geq 11$, because then $(n+1)-3 \geq 8$, so by strong induction the Inductians can make change for exactly $(n+1)-3$ Strongs, and then they can add a 3Sg coin to get $(n+1)$Sg. So the only thing to do is check that they can make change for all the amounts from 8 to 10Sg, which is not too hard to do.

Here's a detailed writeup using the official format:

Proof. We prove by strong induction that the Inductians can make change for any amount of at least 8Sg. The induction hypothesis, $P(n)$ will be:

There is a collection of coins whose value is $n+8$ Strongs.

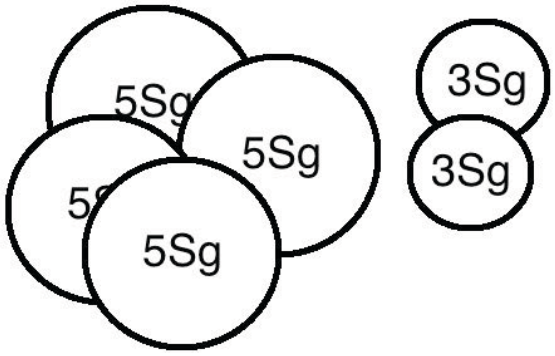


Figure 5.5 One way to make 26 Sg using Strongian currency

We now proceed with the induction proof:

Base case: $P(0)$ is true because a 3Sg coin together with a 5Sg coin makes 8Sg.

Inductive step: We assume $P(k)$ holds for all $k \leq n$, and prove that $P(n+1)$ holds. We argue by cases:

Case ($n + 1 = 1$): We have to make $(n + 1) + 8 = 9$Sg. We can do this using three 3Sg coins.

Case ($n + 1 = 2$): We have to make $(n + 1) + 8 = 10$Sg. Use two 5Sg coins.

Case ($n + 1 \geq 3$): Then $0 \leq n - 2 \leq n$, so by the strong induction hypothesis, the Inductians can make change for $(n-2)+8$Sg. Now by adding a 3Sg coin, they can make change for $(n + 1) + 8$Sg, so $P(n + 1)$ holds in this case.

Since $n \geq 0$, we know that $n + 1 \geq 1$ and thus that the three cases cover every possibility. Since $P(n + 1)$ is true in every case, we can conclude by strong induction that for all $n \geq 0$, the Inductians can make change for $n + 8$ Strong. That is, they can make change for any number of eight or more Strong. ■

5.2.4 The Stacking Game

Here is another exciting game that's surely about to sweep the nation!

You begin with a stack of n boxes. Then you make a sequence of moves. In each move, you divide one stack of boxes into two nonempty stacks. The game ends when you have n stacks, each containing a single box. You earn points for each move; in particular, if you divide one stack of height $a + b$ into two stacks with heights a and b, then you score ab points for that move. Your overall score is the sum of the points that you earn for each move. What strategy should you use to maximize your total score?

Stack Heights										Score
$\underline{10}$										
5	$\underline{5}$									25 points
$\underline{5}$	3	2								6
$\underline{4}$	3	2	1							4
2	$\underline{3}$	2	1	2						4
$\underline{2}$	2	2	1	2	1					2
1	$\underline{2}$	2	1	2	1	1				1
1	1	$\underline{2}$	1	2	1	1	1			1
1	1	1	1	$\underline{2}$	1	1	1	1		1
1	1	1	1	1	1	1	1	1	1	1
									Total Score	= 45 points

Figure 5.6 An example of the stacking game with $n = 10$ boxes. On each line, the underlined stack is divided in the next step.

As an example, suppose that we begin with a stack of $n = 10$ boxes. Then the game might proceed as shown in Figure 5.6. Can you find a better strategy?

Analyzing the Game

Let's use strong induction to analyze the unstacking game. We'll prove that your score is determined entirely by the number of boxes—your strategy is irrelevant!

Theorem 5.2.1. *Every way of unstacking n blocks gives a score of $n(n-1)/2$ points.*

There are a couple technical points to notice in the proof:

- The template for a strong induction proof mirrors the one for ordinary induction.
- As with ordinary induction, we have some freedom to adjust indices. In this case, we prove $P(1)$ in the base case and prove that $P(1), \ldots, P(n)$ imply $P(n+1)$ for all $n \geq 1$ in the inductive step.

Proof. The proof is by strong induction. Let $P(n)$ be the proposition that every way of unstacking n blocks gives a score of $n(n-1)/2$.

Base case: If $n = 1$, then there is only one block. No moves are possible, and so the total score for the game is $1(1-1)/2 = 0$. Therefore, $P(1)$ is true.

Inductive step: Now we must show that $P(1), \ldots, P(n)$ imply $P(n+1)$ for all $n \geq 1$. So assume that $P(1), \ldots, P(n)$ are all true and that we have a stack of $n+1$ blocks. The first move must split this stack into substacks with positive sizes a and b where $a + b = n + 1$ and $0 < a, b \leq n$. Now the total score for the game is the sum of points for this first move plus points obtained by unstacking the two resulting substacks:

$$\begin{aligned}
\text{total score} &= (\text{score for 1st move}) \\
&\quad + (\text{score for unstacking } a \text{ blocks}) \\
&\quad + (\text{score for unstacking } b \text{ blocks}) \\
&= ab + \frac{a(a-1)}{2} + \frac{b(b-1)}{2} && \text{by } P(a) \text{ and } P(b) \\
&= \frac{(a+b)^2 - (a+b)}{2} = \frac{(a+b)((a+b)-1)}{2} \\
&= \frac{(n+1)n}{2}
\end{aligned}$$

This shows that $P(1), P(2), \ldots, P(n)$ imply $P(n+1)$.

Therefore, the claim is true by strong induction. ■

5.3 Strong Induction vs. Induction vs. Well Ordering

Strong induction looks genuinely "stronger" than ordinary induction —after all, you can assume a lot more when proving the induction step. Since ordinary induction is a special case of strong induction, you might wonder why anyone would bother with the ordinary induction.

But strong induction really isn't any stronger, because a simple text manipulation program can automatically reformat any proof using strong induction into a proof using ordinary induction—just by decorating the induction hypothesis with a universal quantifier in a standard way. Still, it's worth distinguishing these two kinds of induction, since which you use will signal whether the inductive step for $n+1$ follows directly from the case for n or requires cases smaller than n, and that is generally good for your reader to know.

The template for the two kinds of induction rules looks nothing like the one for the Well Ordering Principle, but this chapter included a couple of examples where induction was used to prove something already proved using well ordering. In fact, this can always be done. As the examples may suggest, any well ordering proof can automatically be reformatted into an induction proof. So theoretically, no one

need bother with the Well Ordering Principle either.

But it's equally easy to go the other way, and automatically reformat any strong induction proof into a Well Ordering proof. The three proof methods—well ordering, induction, and strong induction—are simply different formats for presenting the same mathematical reasoning!

So why three methods? Well, sometimes induction proofs are clearer because they don't require proof by contradiction. Also, induction proofs often provide recursive procedures that reduce large inputs to smaller ones. On the other hand, well ordering can come out slightly shorter and sometimes seem more natural and less worrisome to beginners.

So which method should you use? There is no simple recipe. Sometimes the only way to decide is to write up a proof using more than one method and compare how they come out. But whichever method you choose, be sure to state the method up front to help a reader follow your proof.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

5.4 State Machines

State machines are a simple, abstract model of step-by-step processes. Since computer programs can be understood as defining step-by-step computational processes, it's not surprising that state machines come up regularly in computer science. They also come up in many other settings such as designing digital circuits and modeling probabilistic processes. This section introduces *Floyd's Invariant Principle* which is a version of induction tailored specifically for proving properties of state machines.

One of the most important uses of induction in computer science involves proving one or more desirable properties continues to hold at every step in a process. A property that is preserved through a series of operations or steps is known as a *preserved invariant* . Examples of desirable invariants include properties such as a variable never exceeding a certain value, the altitude of a plane never dropping below 1,000 feet without the wingflaps being deployed, and the temperature of a nuclear reactor never exceeding the threshold for a meltdown.

5.4.1 States and Transitions

Formally, a *state machine* is nothing more than a binary relation on a set, except that the elements of the set are called "states," the relation is called the transition relation, and an arrow in the graph of the transition relation is called a *transition*. A transition from state q to state r will be written $q \longrightarrow r$. The transition relation

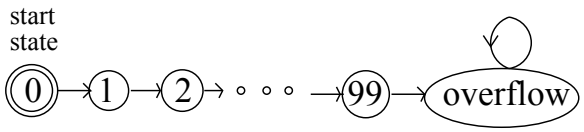


Figure 5.7 *State transitions for the 99-bounded counter.*

is also called the *state graph* of the machine. A state machine also comes equipped with a designated *start state*.

A simple example is a bounded counter, which counts from 0 to 99 and overflows at 100. This state machine is pictured in Figure 5.7, with states pictured as circles, transitions by arrows, and with start state 0 indicated by the double circle. To be precise, what the picture tells us is that this bounded counter machine has

$$\begin{aligned} \text{states} &::= \{0, 1, \ldots, 99, \text{overflow}\}, \\ \text{start state} &::= 0, \\ \text{transitions} &::= \{n \longrightarrow n+1 \mid 0 \le n < 99\} \\ &\quad \cup \{99 \longrightarrow \text{overflow}, \text{overflow} \longrightarrow \text{overflow}\}. \end{aligned}$$

This machine isn't much use once it overflows, since it has no way to get out of its overflow state.

State machines for digital circuits and string pattern matching algorithms, for instance, usually have only a finite number of states. Machines that model continuing computations typically have an infinite number of states. For example, instead of the 99-bounded counter, we could easily define an "unbounded" counter that just keeps counting up without overflowing. The unbounded counter has an infinite state set, the nonnegative integers, which makes its state diagram harder to draw.

State machines are often defined with labels on states and/or transitions to indicate such things as input or output values, costs, capacities, or probabilities. Our state machines don't include any such labels because they aren't needed for our purposes. We do name states, as in Figure 5.7, so we can talk about them, but the names aren't part of the state machine.

5.4.2 Invariant for a Diagonally-Moving Robot

Suppose we have a robot that starts at the origin and moves on an infinite 2-dimensional integer grid. The *state* of the robot at any time can be specified by the integer coordinates (x, y) of the robot's current position. So the *start state* is $(0, 0)$. At each step, the robot may move to a diagonally adjacent grid point, as illustrated in Figure 5.8.

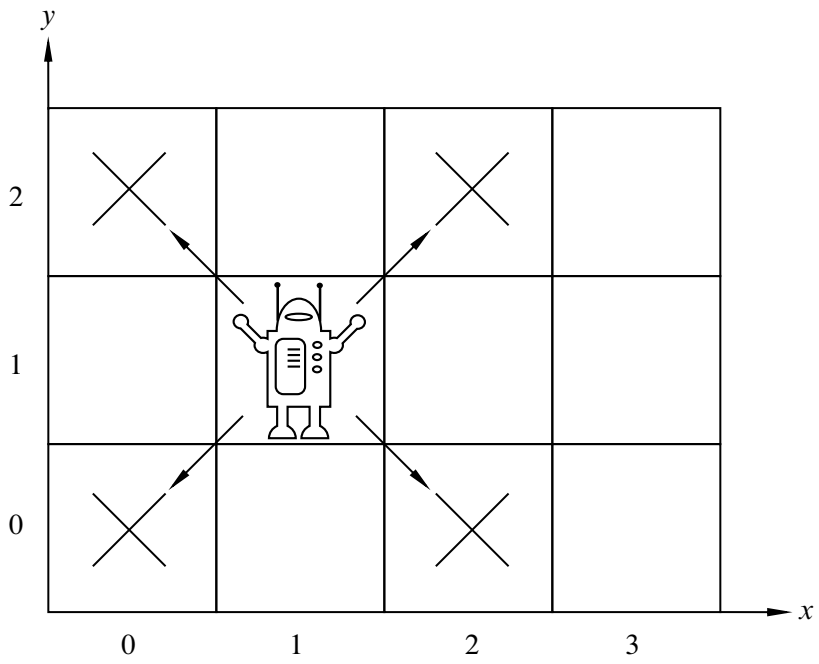


Figure 5.8 *The Diagonally Moving Robot.*

To be precise, the robot's transitions are:

$$\{(m,n) \longrightarrow (m \pm 1, n \pm 1) \mid m,n \in \mathbb{Z}\}.$$

For example, after the first step, the robot could be in states $(1,1)$, $(1,-1)$, $(-1,1)$, or $(-1,-1)$. After two steps, there are 9 possible states for the robot, including $(0,0)$. The question is, can the robot ever reach position $(1,0)$?

If you play around with the robot a bit, you'll probably notice that the robot can only reach positions (m,n) for which $m+n$ is even, which of course means that it can't reach $(1,0)$. This follows because the evenness of the sum of the coordinates is preserved by transitions.

This once, let's go through this preserved-property argument, carefully highlighting where induction comes in. Specifically, define the even-sum property of states to be:

$$\text{Even-sum}((m,n)) ::= [m+n \text{ is even}].$$

Lemma 5.4.1. *For any transition, $q \longrightarrow r$, of the diagonally-moving robot, if Even-sum(q), then Even-sum(r).*

This lemma follows immediately from the definition of the robot's transitions: $(m,n) \longrightarrow (m \pm 1, n \pm 1)$. After a transition, the sum of coordinates changes by

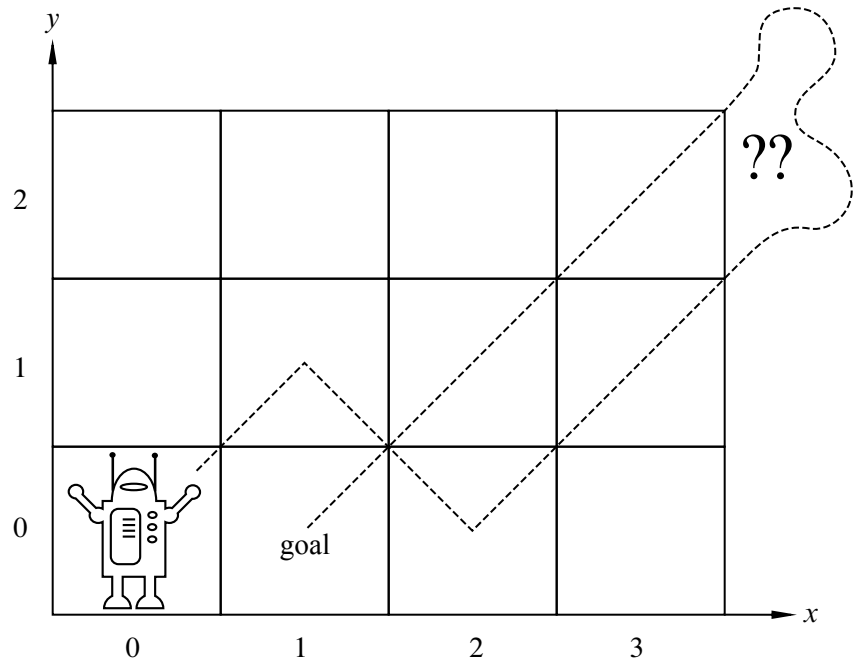


Figure 5.9 *Can the Robot get to* $(1, 0)$*?*

$(\pm 1) + (\pm 1)$, that is, by 0, 2, or -2. Of course, adding 0, 2 or -2 to an even number gives an even number. So by a trivial induction on the number of transitions, we can prove:

Theorem 5.4.2. *The sum of the coordinates of any state reachable by the diagonally-moving robot is even.*

Proof. The proof is induction on the number of transitions the robot has made. The induction hypothesis is

$$P(n) ::= \text{if } q \text{ is a state reachable in } n \text{ transitions, then Even-sum}(q).$$

Base case: $P(0)$ is true since the only state reachable in 0 transitions is the start state $(0, 0)$, and $0 + 0$ is even.

Inductive step: Assume that $P(n)$ is true, and let r be any state reachable in $n + 1$ transitions. We need to prove that Even-sum(r) holds.

Since r is reachable in $n + 1$ transitions, there must be a state, q, reachable in n transitions such that $q \longrightarrow r$. Since $P(n)$ is assumed to be true, Even-sum(q) holds, and so by Lemma 5.4.1, Even-sum(r) also holds. This proves that $P(n)$ IMPLIES $P(n + 1)$ as required, completing the proof of the inductive step.

We conclude by induction that for all $n \geq 0$, if q is reachable in n transitions, then Even-sum(q). This implies that every reachable state has the Even-sum property. ■

Corollary 5.4.3. *The robot can never reach position* $(1, 0)$.

Proof. By Theorem 5.4.2, we know the robot can only reach positions with coordinates that sum to an even number, and thus it cannot reach position $(1, 0)$. ■

5.4.3 The Invariant Principle

Using the Even-sum invariant to understand the diagonally-moving robot is a simple example of a basic proof method called The Invariant Principle. The Principle summarizes how induction on the number of steps to reach a state applies to invariants.

A state machine *execution* describes a possible sequence of steps a machine might take.

Definition 5.4.4. An *execution* of the state machine is a (possibly infinite) sequence of states with the property that

- it begins with the start state, and

- if q and r are consecutive states in the sequence, then $q \longrightarrow r$.

A state is called *reachable* if it appears in some execution.

Definition 5.4.5. A *preserved invariant* of a state machine is a predicate, P, on states, such that whenever $P(q)$ is true of a state, q, and $q \longrightarrow r$ for some state, r, then $P(r)$ holds.

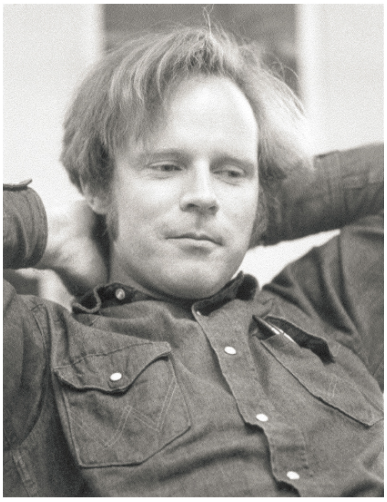
The Invariant Principle

If a preserved invariant of a state machine is true for the start state, then it is true for all reachable states.

The Invariant Principle is nothing more than the Induction Principle reformulated in a convenient form for state machines. Showing that a predicate is true in the start state is the base case of the induction, and showing that a predicate is a preserved invariant corresponds to the inductive step.[5]

[5]Preserved invariants are commonly just called "invariants" in the literature on program correctness, but we decided to throw in the extra adjective to avoid confusion with other definitions. For example, other texts (as well as another subject at MIT) use "invariant" to mean "predicate true of all reachable states." Let's call this definition "invariant-2." Now invariant-2 seems like a reasonable definition, since unreachable states by definition don't matter, and all we want to show is that a desired property is invariant-2. But this confuses the *objective* of demonstrating that a property is invariant-2 with the *method* of finding a *preserved* invariant to *show* that it is invariant-2.

Robert W. Floyd



The Invariant Principle was formulated by Robert W. Floyd at Carnegie Tech in 1967. (Carnegie Tech was renamed Carnegie-Mellon University the following year.) Floyd was already famous for work on the formal grammars that transformed the field of programming language parsing; that was how he got to be a professor even though he never got a Ph.D. (He had beenadmitted to a PhD program as a teenage prodigy, but flunked out and never went back.)

In that same year, Albert R. Meyer was appointed Assistant Professor in the Carnegie Tech Computer Science Department, where he first met Floyd. Floyd and Meyer were the only theoreticians in the department, and they were both delighted to talk about their shared interests. After just a few conversations, Floyd's new junior colleague decided that Floyd was the smartest person he had ever met.

Naturally, one of the first things Floyd wanted to tell Meyer about was his new, as yet unpublished, Invariant Principle. Floyd explained the result to Meyer, and Meyer wondered (privately) how someone as brilliant as Floyd could be excited by such a trivial observation. Floyd had to show Meyer a bunch of examples before Meyer understood Floyd's excitement —not at the truth of the utterly obvious Invariant Principle, but rather at the insight that such a simple method could be so widely and easily applied in verifying programs.

Floyd left for Stanford the following year. He won the Turing award—the "Nobel prize" of computer science—in the late 1970's, in recognition of his work on grammars and on the foundations of program verification. He remained at Stanford from 1968 until his death in September, 2001. You can learn more about Floyd's life and work by reading the eulogy at

http://oldwww.acm.org/pubs/membernet/stories/floyd.pdf

written by his closest colleague, Don Knuth.

5.4.4 The Die Hard Example

The movie *Die Hard 3: With a Vengeance* includes an amusing example of a state machine. The lead characters played by Samuel L. Jackson and Bruce Willis have to disarm a bomb planted by the diabolical Simon Gruber:

> **Simon:** On the fountain, there should be 2 jugs, do you see them? A 5-gallon and a 3-gallon. Fill one of the jugs with exactly 4 gallons of water and place it on the scale and the timer will stop. You must be precise; one ounce more or less will result in detonation. If you're still alive in 5 minutes, we'll speak.
>
> **Bruce:** Wait, wait a second. I don't get it. Do you get it?
>
> **Samuel:** No.
>
> **Bruce:** Get the jugs. Obviously, we can't fill the 3-gallon jug with 4 gallons of water.
>
> **Samuel:** Obviously.
>
> **Bruce:** All right. I know, here we go. We fill the 3-gallon jug exactly to the top, right?
>
> **Samuel:** Uh-huh.
>
> **Bruce:** Okay, now we pour this 3 gallons into the 5-gallon jug, giving us exactly 3 gallons in the 5-gallon jug, right?
>
> **Samuel:** Right, then what?
>
> **Bruce:** All right. We take the 3-gallon jug and fill it a third of the way...
>
> **Samuel:** No! He said, "Be precise." Exactly 4 gallons.
>
> **Bruce:** Sh - -. Every cop within 50 miles is running his a - - off and I'm out here playing kids games in the park.
>
> **Samuel:** Hey, you want to focus on the problem at hand?

Fortunately, they find a solution in the nick of time. You can work out how.

The Die Hard 3 State Machine

The jug-filling scenario can be modeled with a state machine that keeps track of the amount, b, of water in the big jug, and the amount, l, in the little jug. With the 3 and 5 gallon water jugs, the states formally will be pairs, (b, l), of real numbers

such that $0 \le b \le 5, 0 \le l \le 3$. (We can prove that the reachable values of b and l will be nonnegative integers, but we won't assume this.) The start state is $(0,0)$, since both jugs start empty.

Since the amount of water in the jug must be known exactly, we will only consider moves in which a jug gets completely filled or completely emptied. There are several kinds of transitions:

1. Fill the little jug: $(b,l) \longrightarrow (b,3)$ for $l < 3$.
2. Fill the big jug: $(b,l) \longrightarrow (5,l)$ for $b < 5$.
3. Empty the little jug: $(b,l) \longrightarrow (b,0)$ for $l > 0$.
4. Empty the big jug: $(b,l) \longrightarrow (0,l)$ for $b > 0$.
5. Pour from the little jug into the big jug: for $l > 0$,
$$(b,l) \longrightarrow \begin{cases} (b+l,0) & \text{if } b+l \le 5, \\ (5, l-(5-b)) & \text{otherwise.} \end{cases}$$
6. Pour from big jug into little jug: for $b > 0$,
$$(b,l) \longrightarrow \begin{cases} (0, b+l) & \text{if } b+l \le 3, \\ (b-(3-l), 3) & \text{otherwise.} \end{cases}$$

Note that in contrast to the 99-counter state machine, there is more than one possible transition out of states in the Die Hard machine. Machines like the 99-counter with at most one transition out of each state are called *deterministic*. The Die Hard machine is *nondeterministic* because some states have transitions to several different states.

The Die Hard 3 bomb gets disarmed successfully because the state (4,3) is reachable.

Die Hard Once and For All

The *Die Hard* series is getting tired, so we propose a final *Die Hard Once and For All*. Here, Simon's brother returns to avenge him, posing the same challenge, but with the 5 gallon jug replaced by a 9 gallon one. The state machine has the same specification as the Die Hard 3 version, except all occurrences of "5" are replaced by "9."

Now, reaching any state of the form $(4,l)$ is impossible. We prove this using the Invariant Principle. Specifically, we define the preserved invariant predicate, $P((b,l))$, to be that b and l are nonnegative integer multiples of 3.

To prove that P is a preserved invariant of Die-Hard-Once-and-For-All machine, we assume $P(q)$ holds for some state $q ::= (b, l)$ and that $q \longrightarrow r$. We have to show that $P(r)$ holds. The proof divides into cases, according to which transition rule is used.

One case is a "fill the little jug" transition. This means $r = (b, 3)$. But $P(q)$ implies that b is an integer multiple of 3, and of course 3 is an integer multiple of 3, so $P(r)$ still holds.

Another case is a "pour from big jug into little jug" transition. For the subcase when there isn't enough room in the little jug to hold all the water, that is, when $b + l > 3$, we have $r = (b - (3 - l), 3)$. But $P(q)$ implies that b and l are integer multiples of 3, which means $b - (3 - l)$ is too, so in this case too, $P(r)$ holds.

We won't bother to crank out the remaining cases, which can all be checked just as easily. Now by the Invariant Principle, we conclude that every reachable state satisifies P. But since no state of the form $(4, l)$ satisifies P, we have proved rigorously that Bruce dies once and for all!

By the way, notice that the state (1,0), which satisfies NOT(P), has a transition to (0,0), which satisfies P. So the negation of a preserved invariant may not be a preserved invariant.

5.4.5 Fast Exponentiation

Partial Correctness & Termination

Floyd distinguished two required properties to verify a program. The first property is called *partial correctness*; this is the property that the final results, if any, of the process must satisfy system requirements.

You might suppose that if a result was only partially correct, then it might also be partially incorrect, but that's not what Floyd meant. The word "partial" comes from viewing a process that might not terminate as computing a *partial relation*. Partial correctness means that *when there is a result*, it is correct, but the process might not always produce a result, perhaps because it gets stuck in a loop.

The second correctness property, called *termination*, is that the process does always produce some final value.

Partial correctness can commonly be proved using the Invariant Principle. Termination can commonly be proved using the Well Ordering Principle. We'll illustrate this by verifying a Fast Exponentiation procedure.

Exponentiating

The most straightforward way to compute the bth power of a number, a, is to multiply a by itself $b - 1$ times. But the solution can be found in considerably

fewer multiplications by using a technique called *Fast Exponentiation*. The register machine program below defines the fast exponentiation algorithm. The letters x, y, z, r denote registers that hold numbers. An *assignment statement* has the form "$z := a$" and has the effect of setting the number in register z to be the number a.

A Fast Exponentiation Program

Given inputs $a \in \mathbb{R}, b \in \mathbb{N}$, initialize registers x, y, z to $a, 1, b$ respectively, and repeat the following sequence of steps until termination:

- if $z = 0$ **return** y and terminate
- $r :=$ remainder$(z, 2)$
- $z :=$ quotient$(z, 2)$
- if $r = 1$, then $y := xy$
- $x := x^2$

We claim this program always terminates and leaves $y = a^b$.
To begin, we'll model the behavior of the program with a state machine:

1. states $::= \mathbb{R} \times \mathbb{R} \times \mathbb{N}$,
2. start state $::= (a, 1, b)$,
3. transitions are defined by the rule

$$(x, y, z) \longrightarrow \begin{cases} (x^2, y, \text{quotient}(z, 2)) & \text{if } z \text{ is nonzero and even,} \\ (x^2, xy, \text{quotient}(z, 2)) & \text{if } z \text{ is nonzero and odd.} \end{cases}$$

The preserved invariant, $P((x, y, z))$, will be

$$z \in \mathbb{N} \text{ AND } yx^z = a^b. \tag{5.4}$$

To prove that P is preserved, assume $P((x, y, z))$ holds and that $(x, y, z) \longrightarrow (x_t, y_t, z_t)$. We must prove that $P((x_t, y_t, z_t))$ holds, that is,

$$z_t \in \mathbb{N} \text{ AND } y_t x_t^{z_t} = a^b. \tag{5.5}$$

Since there is a transition from (x, y, z), we have $z \neq 0$, and since $z \in \mathbb{N}$ by (5.4), we can consider just two cases:

If z is even, then we have that $x_t = x^2, y_t = y, z_t = z/2$. Therefore, $z_t \in \mathbb{N}$ and

$$\begin{aligned} y_t x_t^{z_t} &= y(x^2)^{z/2} \\ &= yx^{2\cdot z/2} \\ &= yx^z \\ &= a^b && \text{(by (5.4))} \end{aligned}$$

If z is odd, then we have that $x_t = x^2, y_t = xy, z_t = (z-1)/2$. Therefore, $z_t \in \mathbb{N}$ and

$$\begin{aligned} y_t x_t^{z_t} &= xy(x^2)^{(z-1)/2} \\ &= yx^{1+2\cdot(z-1)/2} \\ &= yx^{1+(z-1)} \\ &= yx^z \\ &= a^b && \text{(by (5.4))} \end{aligned}$$

So in both cases, (5.5) holds, proving that P is a preserved invariant.

Now it's easy to prove partial correctness: if the Fast Exponentiation program terminates, it does so with a^b in register y. This works because $1 \cdot a^b = a^b$, which means that the start state, $(a, 1, b)$, satisifies P. By the Invariant Principle, P holds for all reachable states. But the program only stops when $z = 0$. If a terminated state $(x, y, 0)$ is reachable, then $y = yx^0 = a^b$ as required.

Ok, it's partially correct, but what's fast about it? The answer is that the number of multiplications it performs to compute a^b is roughly the length of the binary representation of b. That is, the Fast Exponentiation program uses roughly $\log b$[6] multiplications, compared to the naive approach of multiplying by a a total of $b-1$ times.

More precisely, it requires at most $2(\lceil \log b \rceil + 1)$ multiplications for the Fast Exponentiation algorithm to compute a^b for $b > 1$. The reason is that the number in register z is initially b, and gets at least halved with each transition. So it can't be halved more than $\lceil \log b \rceil + 1$ times before hitting zero and causing the program to terminate. Since each of the transitions involves at most two multiplications, the total number of multiplications until $z = 0$ is at most $2(\lceil \log b \rceil + 1)$ for $b > 0$ (see Problem 5.36).

[6] As usual in computer science, $\log b$ means the base two logarithm, $\log_2 b$. We use, $\ln b$ for the natural logarithm $\log_e b$, and otherwise write the logarithm base explicitly, as in $\log_{10} b$.

5.4.6 Derived Variables

The preceding termination proof involved finding a nonnegative integer-valued measure to assign to states. We might call this measure the "size" of the state. We then showed that the size of a state decreased with every state transition. By the Well Ordering Principle, the size can't decrease indefinitely, so when a minimum size state is reached, there can't be any transitions possible: the process has terminated.

More generally, the technique of assigning values to states—not necessarily nonnegative integers and not necessarily decreasing under transitions—is often useful in the analysis of algorithms. *Potential functions* play a similar role in physics. In the context of computational processes, such value assignments for states are called *derived variables*.

For example, for the Die Hard machines we could have introduced a derived variable, f : states $\to \mathbb{R}$, for the amount of water in both buckets, by setting $f((a,b)) ::= a+b$. Similarly, in the robot problem, the position of the robot along the x-axis would be given by the derived variable x-coord, where $x\text{-coord}((i,j)) ::= i$.

There are a few standard properties of derived variables that are handy in analyzing state machines.

Definition 5.4.6. A derived variable f : states $\to \mathbb{R}$ is *strictly decreasing* iff

$$q \longrightarrow q' \text{ IMPLIES } f(q') < f(q).$$

It is *weakly decreasing* iff

$$q \longrightarrow q' \text{ IMPLIES } f(q') \leq f(q).$$

Strictly increasing and *weakly increasing* derived variables are defined similarly.[7]

We confirmed termination of the Fast Exponentiation procedure by noticing that the derived variable z was nonnegative-integer-valued and strictly decreasing. We can summarize this approach to proving termination as follows:

Theorem 5.4.7. *If f is a strictly decreasing $\mathbb{N}$-valued derived variable of a state machine, then the length of any execution starting at state q is at most $f(q)$.*

Of course, we could prove Theorem 5.4.7 by induction on the value of $f(q)$, but think about what it says: "If you start counting down at some nonnegative integer $f(q)$, then you can't count down more than $f(q)$ times." Put this way, it's obvious.

[7] Weakly increasing variables are often also called *nondecreasing*. We will avoid this terminology to prevent confusion between nondecreasing variables and variables with the much weaker property of *not* being a decreasing variable.

Theorem 5.4.7 generalizes straightforwardly to derived variables taking values in a well ordered set (Section 2.4.

Theorem 5.4.8. *If there exists a strictly decreasing derived variable whose range is a well ordered set, then every execution terminates.*

Theorem 5.4.8 follows immediately from the observation that a set of numbers is well ordered iff it has no infinite decreasing sequences (Problem 2.17).

Note that the existence of a *weakly* decreasing derived variable does not guarantee that every execution terminates. An infinite execution could proceed through states in which a weakly decreasing variable remained constant.

A Southeast Jumping Robot (Optional)

Here's a contrived, simple example of proving termination based on a variable that is strictly decreasing over a well ordered set. Let's think about a robot positioned at an integer lattice-point in the Northeast quadrant of the plane, that is, at $(x, y) \in \mathbb{N}^2$.

At every second when it is away from the origin, $(0, 0)$, the robot must make a move, which may be

- a unit distance West when it is not at the boundary of the Northeast quadrant (that is, $(x, y) \longrightarrow (x - 1, y)$ for $x > 0$), or
- a unit distance South combined with an arbitrary jump East (that is, $(x, y) \longrightarrow (z, y - 1)$ for $z \geq x$).

Claim 5.4.9. *The robot will always get stuck at the origin.*

If we think of the robot as a nondeterministic state machine, then Claim 5.4.9 is a termination assertion. The Claim may seem obvious, but it really has a different character than termination based on nonnegative integer-valued variables. That's because, even knowing that the robot is at position $(0, 1)$, for example, there is no way to bound the time it takes for the robot to get stuck. It can delay getting stuck for as many seconds as it wants by making its next move to a distant point in the Far East. This rules out proving termination using Theorem 5.4.7.

So does Claim 5.4.9 still seem obvious?

Well it is if you see the trick. Define a derived variable, v, mapping robot states to the numbers in the well ordered set $\mathbb{N} + \mathbb{F}$ of Lemma 2.4.5. In particular, define $v : \mathbb{N}^2 \to \mathbb{N} + \mathbb{F}$ as follows

$$v(x, y) ::= y + \frac{x}{x + 1}.$$

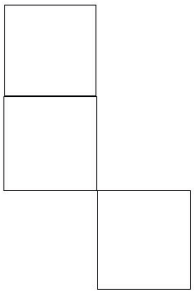


Figure 5.10 Gehry's new tile.

Now it's easy to check that if $(x, y) \longrightarrow (x', y')$ is a legitimate robot move, then $v((x', y')) < v((x, y))$. In particular, v is a strictly decreasing derived variable, so Theorem 5.4.8 implies that the robot always get stuck—even though we can't say how many moves it will take until it does.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

6 Recursive Data Types

Recursive data types play a central role in programming, and induction is really all about them.

Recursive data types are specified by *recursive definitions*, which say how to construct new data elements from previous ones. Along with each recursive data type there are recursive definitions of properties or functions on the data type. Most importantly, based on a recursive definition, there is a *structural induction* method for proving that all data of the given type have some property.

This chapter examines a few examples of recursive data types and recursively defined functions on them:

- strings of characters,
- "balanced" strings of brackets,
- the nonnegative integers, and
- arithmetic expressions.

6.1 Recursive Definitions and Structural Induction

We'll start off illustrating recursive definitions and proofs using the example of character strings. Normally we'd take strings of characters for granted, but it's informative to treat them as a recursive data type. In particular, strings are a nice first example because you will see recursive definitions of things that are easy to understand or that you already know, so you can focus on how the definitions work without having to figure out what they are for.

Definitions of recursive data types have two parts:

- **Base case(s)** specifying that some known mathematical elements are in the data type, and
- **Constructor case(s)** that specify how to construct new data elements from previously constructed elements or from base elements.

The definition of strings over a given character set, A, follows this pattern:

Definition 6.1.1. Let A be a nonempty set called an *alphabet*, whose elements are referred to as *characters*, *letters*, or *symbols*. The recursive data type, A^*, of strings over alphabet, A, are defined as follows:

- **Base case**: the empty string, λ, is in A^*.
- **Constructor case**: If $a \in A$ and $s \in A^*$, then the pair $\langle a, s\rangle \in A^*$.

So $\{0,1\}^*$ are the binary strings.

The usual way to treat binary strings is as sequences of 0's and 1's. For example, we have identified the length-4 binary string 1011 as a sequence of bits, the 4-tuple $(1,0,1,1)$. But according to the recursive Definition 6.1.1, this string would be represented by nested pairs, namely

$$\langle 1, \langle 0, \langle 1, \langle 1, \lambda\rangle\rangle\rangle\rangle\,.$$

These nested pairs are definitely cumbersome and may also seem bizarre, but they actually reflect the way that such lists of characters would be represented in programming languages like Scheme or Python, where $\langle a, s\rangle$ would correspond to $\text{cons}(a, s)$.

Notice that we haven't said exactly how the empty string is represented. It really doesn't matter, as long as we can recognize the empty string and not confuse it with any nonempty string.

Continuing the recursive approach, let's define the length of a string.

Definition 6.1.2. The length, $|s|$, of a string, s, is defined recursively based on the definition of $s \in A^*$:

Base case: $|\lambda| ::= 0$.

Constructor case: $|\langle a, s\rangle| ::= 1 + |s|$.

This definition of length follows a standard pattern: functions on recursive data types can be defined recursively using the same cases as the data type definition. Specifically, to define a function, f, on a recursive data type, define the value of f for the base cases of the data type definition, then define the value of f in each constructor case in terms of the values of f on the component data items.

Let's do another example: the *concatenation* $s \cdot t$ of the strings s and t is the string consisting of the letters of s followed by the letters of t. This is a perfectly clear mathematical definition of concatenation (except maybe for what to do with the empty string), and in terms of Scheme/Python lists, $s \cdot t$ would be the list $\text{append}(s, t)$. Here's a recursive definition of concatenation.

Definition 6.1.3. The *concatenation* $s \cdot t$ of the strings $s, t \in A^*$ is defined recursively based on the definition of $s \in A^*$:

Base case:

$$\lambda \cdot t ::= t.$$

Constructor case:

$$\langle a, s\rangle \cdot t ::= \langle a, s \cdot t\rangle .$$

6.1.1 Structural Induction

Structural induction is a method for proving that all the elements of a recursively defined data type have some property. A structural induction proof has two parts corresponding to the recursive definition:

- Prove that each base case element has the property.
- Prove that each constructor case element has the property, when the constructor is applied to elements that have the property.

For example, we can verify the familiar fact that the length of the concatenation of two strings is the sum of their lengths using structural induction:

Theorem 6.1.4. *For all* $s, t \in A^*$,

$$|s \cdot t| = |s| + |t|.$$

Proof. By structural induction on the definition of $s \in A^*$. The induction hypothesis is

$$P(s) ::= \forall t \in A^*. |s \cdot t| = |s| + |t|.$$

Base case ($s = \lambda$):

$$\begin{aligned} |s \cdot t| &= |\lambda \cdot t| \\ &= |t| && \text{(def } \cdot\text{, base case)} \\ &= 0 + |t| \\ &= |s| + |t| && \text{(def length, base case)} \end{aligned}$$

Constructor case: Suppose $s ::= \langle a, r\rangle$ and assume the induction hypothesis, $P(r)$. We must show that $P(s)$ holds:

$$\begin{aligned}
|s \cdot t| &= |\langle a, r\rangle \cdot t| \\
&= |\langle a, r \cdot t\rangle| && \text{(concat def, constructor case)} \\
&= 1 + |r \cdot t| && \text{(length def, constructor case)} \\
&= 1 + (|r| + |t|) && \text{since } P(r) \text{ holds} \\
&= (1 + |r|) + |t| \\
&= |\langle a, r\rangle| + |t| && \text{(length def, constructor case)} \\
&= |s| + |t|.
\end{aligned}$$

This proves that $P(s)$ holds as required, completing the constructor case. By structural induction we conclude that $P(s)$ holds for all strings $s \in A^*$. ■

This proof illustrates the general principle:

The Principle of Structural Induction.

Let P be a predicate on a recursively defined data type R. If

- $P(b)$ is true for each base case element, $b \in R$, and
- for all two-argument constructors, **c**,

$$[P(r) \text{ AND } P(s)] \text{ IMPLIES } P(\mathbf{c}(r, s))$$

for all $r, s \in R$,
and likewise for all constructors taking other numbers of arguments,

then

$P(r)$ is true for all $r \in R$.

6.1.2 One More Thing

The number, $\#_c(s)$, of occurrences of the character $c \in A$ in the string s has a simple recursive definition based on the definition of $s \in A^*$:

Definition 6.1.5.

Base case: $\#_c(\lambda) ::= 0$.

Constructor case:

$$\#_c(\langle a, s\rangle) ::= \begin{cases} \#_c(s) & \text{if } a \neq c, \\ 1 + \#_c(s) & \text{if } a = c. \end{cases}$$

We'll need the following lemma in the next section:

Lemma 6.1.6.

$$\#_c(s \cdot t) = \#_c(s) + \#_c(t).$$

The easy proof by structural induction is an exercise (Problem 6.7).

6.2 Strings of Matched Brackets

Let $\{\texttt{]}, \texttt{[}\}^*$ be the set of all strings of square brackets. For example, the following two strings are in $\{\texttt{]}, \texttt{[}\}^*$:

$$\texttt{[]][[[[[]]]}\quad \text{and} \quad \texttt{[[[]]][]][]} \tag{6.1}$$

A string, $s \in \{\texttt{]}, \texttt{[}\}^*$, is called a *matched string* if its brackets "match up" in the usual way. For example, the left hand string above is not matched because its second right bracket does not have a matching left bracket. The string on the right is matched.

We're going to examine several different ways to define and prove properties of matched strings using recursively defined sets and functions. These properties are pretty straightforward, and you might wonder whether they have any particular relevance in computer science. The honest answer is "not much relevance *any more*." The reason for this is one of the great successes of computer science, as explained in the text box below.

Expression Parsing

During the early development of computer science in the 1950's and 60's, creation of effective programming language compilers was a central concern. A key aspect in processing a program for compilation was expression parsing. One significant problem was to take an expression like

$$x + y * z^2 \div y + 7$$

and *put in* the brackets that determined how it should be evaluated—should it be

$$[[x + y] * z^2 \div y] + 7, \text{ or,}$$
$$x + [y * z^2 \div [y + 7]], \text{ or,}$$
$$[x + [y * z^2]] \div [y + 7], \text{ or} \ldots?$$

The Turing award (the "Nobel Prize" of computer science) was ultimately bestowed on Robert W. Floyd, for, among other things, discovering simple procedures that would insert the brackets properly.

In the 70's and 80's, this parsing technology was packaged into high-level compiler-compilers that automatically generated parsers from expression grammars. This automation of parsing was so effective that the subject no longer demanded attention. It had largely disappeared from the computer science curriculum by the 1990's.

The matched strings can be nicely characterized as a recursive data type:

Definition 6.2.1. Recursively define the set, RecMatch, of strings as follows:

- **Base case**: $\lambda \in \text{RecMatch}$.
- **Constructor case**: If $s, t \in \text{RecMatch}$, then

$$[\, s \,]\, t \in \text{RecMatch}.$$

Here $[\, s \,]\, t$ refers to the concatenation of strings which would be written in full as

$$[\ \cdot (s \cdot (]\ \cdot t)).$$

From now on, we'll usually omit the "·'s."

Using this definition, $\lambda \in \text{RecMatch}$ by the base case, so letting $s = t = \lambda$ in the constructor case implies

$$[\, \lambda]\, \lambda = [\ ] \in \text{RecMatch}.$$

Now,

$$\begin{aligned} [\,\lambda\,][\,] &= [\,][\,] \in \text{RecMatch} && \text{(letting } s = \lambda, t = [\,]\text{)} \\ [\,[\,]\,]\lambda &= [\,[\,]\,] \in \text{RecMatch} && \text{(letting } s = [\,]\,, t = \lambda\text{)} \\ [\,[\,]\,][\,] &\in \text{RecMatch} && \text{(letting } s = [\,]\,, t = [\,]\text{)} \end{aligned}$$

are also strings in RecMatch by repeated applications of the constructor case; and so on.

It's pretty obvious that in order for brackets to match, there had better be an equal number of left and right ones. For further practice, let's carefully prove this from the recursive definitions.

Lemma. *Every string in RecMatch has an equal number of left and right brackets.*

Proof. The proof is by structural induction with induction hypothesis

$$P(s) ::= \#_{[}(s) = \#_{]}(s).$$

Base case: $P(\lambda)$ holds because

$$\#_{[}(\lambda) = 0 = \#_{]}(\lambda)$$

by the base case of Definition 6.1.5 of $\#_c()$.

Constructor case: By structural induction hypothesis, we assume $P(s)$ and $P(t)$ and must show $P([\,s\,]\,t)$:

$$\begin{aligned} \#_{[}([\,s\,]\,t) &= \#_{[}([\,) + \#_{[}(s) + \#_{[}(\,]) + \#_{[}(t) && \text{(Lemma 6.1.6)} \\ &= 1 + \#_{[}(s) + 0 + \#_{[}(t) && \text{(def } \#_{[}()\text{)} \\ &= 1 + \#_{]}(s) + 0 + \#_{]}(t) && \text{(by } P(s) \text{ and } P(t)\text{)} \\ &= 0 + \#_{]}(s) + 1 + \#_{]}(t) \\ &= \#_{]}([\,) + \#_{]}(s) + \#_{]}(\,]) + \#_{]}(t) && \text{(def } \#_{]}()\text{)} \\ &= \#_{]}([\,s\,]\,t) && \text{(Lemma 6.1.6)} \end{aligned}$$

This completes the proof of the constructor case. We conclude by structural induction that $P(s)$ holds for all $s \in$ RecMatch. ■

Warning: When a recursive definition of a data type allows the same element to be constructed in more than one way, the definition is said to be *ambiguous*. We were careful to choose an *un*ambiguous definition of RecMatch to ensure that functions defined recursively on its definition would always be well-defined. Recursively defining a function on an ambiguous data type definition usually will not work. To illustrate the problem, here's another definition of the matched strings.

Definition 6.2.2. Define the set, AmbRecMatch $\subseteq$ {]\, [}* recursively as follows:

- **Base case**: $\lambda \in$ AmbRecMatch,
- **Constructor cases**: if $s, t \in$ AmbRecMatch, then the strings [s] and st are also in AmbRecMatch.

It's pretty easy to see that the definition of AmbRecMatch is just another way to define RecMatch, that is AmbRecMatch = RecMatch (see Problem 6.15). The definition of AmbRecMatch is arguably easier to understand, but we didn't use it because it's ambiguous, while the trickier definition of RecMatch is unambiguous. Here's why this matters. Let's define the number of operations, $f(s)$, to construct a matched string s recursively on the definition of $s \in$ AmbRecMatch:

$$\begin{aligned} f(\lambda) &::= 0, && (f \text{ base case}) \\ f([\,s\,]) &::= 1 + f(s), \\ f(st) &::= 1 + f(s) + f(t). && (f \text{ concat case}) \end{aligned}$$

This definition may seem ok, but it isn't: $f(\lambda)$ winds up with two values, and consequently:

$$\begin{aligned} 0 &= f(\lambda) && (f \text{ base case})) \\ &= f(\lambda \cdot \lambda) && (\text{concat def, base case}) \\ &= 1 + f(\lambda) + f(\lambda) && (f \text{ concat case}), \\ &= 1 + 0 + 0 = 1 && (f \text{ base case}). \end{aligned}$$

This is definitely not a situation we want to be in!

6.3 Recursive Functions on Nonnegative Integers

The nonnegative integers can be understood as a recursive data type.

Definition 6.3.1. The set, $\mathbb{N}$, is a data type defined recursively as:

- $0 \in \mathbb{N}$.
- If $n \in \mathbb{N}$, then the *successor*, $n + 1$, of n is in $\mathbb{N}$.

The point here is to make it clear that ordinary induction is simply the special case of structural induction on the recursive Definition 6.3.1. This also justifies the familiar recursive definitions of functions on the nonnegative integers.

6.3.1 Some Standard Recursive Functions on $\mathbb{N}$

Example 6.3.2. *The factorial function.* This function is often written "$n!$." You will see a lot of it in later chapters. Here, we'll use the notation $\text{fac}(n)$:

- $\text{fac}(0) ::= 1$.
- $\text{fac}(n+1) ::= (n+1) \cdot \text{fac}(n)$ for $n \geq 0$.

Example 6.3.3. *The Fibonacci numbers.* Fibonacci numbers arose out of an effort 800 years ago to model population growth. They have a continuing fan club of people captivated by their extraordinary properties (see Problems 5.8, 5.21, 5.26). The nth Fibonacci number, fib, can be defined recursively by:

$$\begin{aligned} F(0) &::= 0, \\ F(1) &::= 1, \\ F(n) &::= F(n-1) + F(n-2) \qquad \text{for } n \geq 2. \end{aligned}$$

Here the recursive step starts at $n = 2$ with base cases for 0 and 1. This is needed since the recursion relies on two previous values.

What is $F(4)$? Well, $F(2) = F(1) + F(0) = 1$, $F(3) = F(2) + F(1) = 2$, so $F(4) = 3$. The sequence starts out $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

Example 6.3.4. *Summation notation.* Let "$S(n)$" abbreviate the expression "$\sum_{i=1}^{n} f(i)$." We can recursively define $S(n)$ with the rules

- $S(0) ::= 0$.
- $S(n+1) ::= f(n+1) + S(n)$ for $n \geq 0$.

6.3.2 Ill-formed Function Definitions

There are some other blunders to watch out for when defining functions recursively. The main problems come when recursive definitions don't follow the recursive definition of the underlying data type. Below are some function specifications that resemble good definitions of functions on the nonnegative integers, but really aren't.

$$f_1(n) ::= 2 + f_1(n-1). \tag{6.2}$$

This "definition" has no base case. If some function, f_1, satisfied (6.2), so would a function obtained by adding a constant to the value of f_1. So equation (6.2) does not uniquely define an f_1.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_2(n+1) & \text{otherwise.} \end{cases} \tag{6.3}$$

This "definition" has a base case, but still doesn't uniquely determine f_2. Any function that is 0 at 0 and constant everywhere else would satisfy the specification, so (6.3) also does not uniquely define anything.

In a typical programming language, evaluation of $f_2(1)$ would begin with a recursive call of $f_2(2)$, which would lead to a recursive call of $f_2(3)$, ... with recursive calls continuing without end. This "operational" approach interprets (6.3) as defining a *partial* function, f_2, that is undefined everywhere but 0.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by 2,} \\ 1, & \text{if } n \text{ is divisible by 3,} \\ 2, & \text{otherwise.} \end{cases} \tag{6.4}$$

This "definition" is inconsistent: it requires $f_3(6) = 0$ and $f_3(6) = 1$, so (6.4) doesn't define anything.

Mathematicians have been wondering about this function specification, known as the Collatz conjecture for a while:

$$f_4(n) ::= \begin{cases} 1, & \text{if } n \leq 1, \\ f_4(n/2) & \text{if } n > 1 \text{ is even,} \\ f_4(3n+1) & \text{if } n > 1 \text{ is odd.} \end{cases} \tag{6.5}$$

For example, $f_4(3) = 1$ because

$$f_4(3) ::= f_4(10) ::= f_4(5) ::= f_4(16) ::= f_4(8) ::= f_4(4) ::= f_4(2) ::= f_4(1) ::= 1.$$

The constant function equal to 1 will satisfy (6.5), but it's not known if another function does as well. The problem is that the third case specifies $f_4(n)$ in terms of f_4 at arguments larger than n, and so cannot be justified by induction on $\mathbb{N}$. It's known that any f_4 satisfying (6.5) equals 1 for all n up to over 10^{18}.

A final example is the Ackermann function, which is an extremely fast-growing function of two nonnegative arguments. Its inverse is correspondingly slow-growing—it grows slower than $\log n$, $\log\log n$, $\log\log\log n$, ..., but it does grow unboundly. This inverse actually comes up analyzing a useful, highly efficient procedure known as the *Union-Find algorithm*. This algorithm was conjectured to run in a number of steps that grew linearly in the size of its input, but turned out to be "linear"

but with a slow growing coefficient nearly equal to the inverse Ackermann function. This means that pragmatically, *Union-Find* is linear, since the theoretically growing coefficient is less than 5 for any input that could conceivably come up.

The Ackermann function can be defined recursively as the function, A, given by the following rules:

$$A(m,n) = 2n, \qquad \text{if } m = 0 \text{ or } n \leq 1, \tag{6.6}$$

$$A(m,n) = A(m-1, A(m,n-1)), \qquad \text{otherwise.} \tag{6.7}$$

Now these rules are unusual because the definition of $A(m,n)$ involves an evaluation of A at arguments that may be a lot bigger than m and n. The definitions of f_2 above showed how definitions of function values at small argument values in terms of larger one can easily lead to nonterminating evaluations. The definition of the Ackermann function is actually ok, but proving this takes some ingenuity (see Problem 6.17).

6.4 Arithmetic Expressions

Expression evaluation is a key feature of programming languages, and recognition of expressions as a recursive data type is a key to understanding how they can be processed.

To illustrate this approach we'll work with a toy example: arithmetic expressions like $3x^2 + 2x + 1$ involving only one variable, "x." We'll refer to the data type of such expressions as Aexp. Here is its definition:

Definition 6.4.1.

- **Base cases**:
  - The variable, x, is in Aexp.
  - The arabic numeral, `k`, for any nonnegative integer, k, is in Aexp.
- **Constructor cases**: If e, $f \in$ Aexp, then
  - $[\, e + f \,] \in$ Aexp. The expression $[\, e + f \,]$ is called a *sum*. The Aexp's e and f are called the *components* of the sum; they're also called the *summands*.

- $[e * f] \in$ Aexp. The expression $[e * f]$ is called a *product*. The Aexp's e and f are called the *components* of the product; they're also called the *multiplier* and *multiplicand*.
- $\texttt{-}[e] \in$ Aexp. The expression $\texttt{-}[e]$ is called a *negative*.

Notice that Aexp's are fully bracketed, and exponents aren't allowed. So the Aexp version of the polynomial expression $3x^2+2x+1$ would officially be written as

$$[[3 * [x * x]] + [[2 * x] + 1]]. \tag{6.8}$$

These brackets and $*$'s clutter up examples, so we'll often use simpler expressions like "$3x^2+2x+1$" instead of (6.8). But it's important to recognize that $3x^2+2x+1$ is not an Aexp; it's an *abbreviation* for an Aexp.

6.4.1 Evaluation and Substitution with Aexp's

Evaluating Aexp's

Since the only variable in an Aexp is x, the value of an Aexp is determined by the value of x. For example, if the value of x is 3, then the value of $3x^2 + 2x + 1$ is 34. In general, given any Aexp, e, and an integer value, n, for the variable, x, we can evaluate e to finds its value, $\text{eval}(e, n)$. It's easy, and useful, to specify this evaluation process with a recursive definition.

Definition 6.4.2. The *evaluation function*, $\text{eval} : \text{Aexp} \times \mathbb{Z} \to \mathbb{Z}$, is defined recursively on expressions, $e \in$ Aexp, as follows. Let n be any integer.

- **Base cases**:

$$\text{eval}(x, n) ::= n, \qquad \text{(value of variable } x \text{ is } n.) \tag{6.9}$$

$$\text{eval}(\texttt{k}, n) ::= k, \qquad \text{(value of numeral } \texttt{k} \text{ is } k\text{, regardless of } x.) \tag{6.10}$$

- **Constructor cases**:

$$\text{eval}([e_1 \texttt{+} e_2], n) ::= \text{eval}(e_1, n) + \text{eval}(e_2, n), \tag{6.11}$$

$$\text{eval}([e_1 * e_2], n) ::= \text{eval}(e_1, n) \cdot \text{eval}(e_2, n), \tag{6.12}$$

$$\text{eval}(\texttt{-}[e_1], n) ::= -\text{eval}(e_1, n). \tag{6.13}$$

For example, here's how the recursive definition of eval would arrive at the value of $3 + x^2$ when x is 2:

$$\begin{aligned}
\text{eval}([\,3 + [\,x * x\,]\,], 2) &= \text{eval}(3, 2) + \text{eval}([\,x * x\,], 2) && \text{(by Def 6.4.2.6.11)}\\
&= 3 + \text{eval}([\,x * x\,], 2) && \text{(by Def 6.4.2.6.10)}\\
&= 3 + (\text{eval}(x, 2) \cdot \text{eval}(x, 2)) && \text{(by Def 6.4.2.6.12)}\\
&= 3 + (2 \cdot 2) && \text{(by Def 6.4.2.6.9)}\\
&= 3 + 4 = 7.
\end{aligned}$$

Substituting into Aexp's

Substituting expressions for variables is a standard operation used by compilers and algebra systems. For example, the result of substituting the expression $3x$ for x in the expression $x(x-1)$ would be $3x(3x-1)$. We'll use the general notation $\text{subst}(f, e)$ for the result of substituting an Aexp, f, for each of the x's in an Aexp, e. So as we just explained,

$$\text{subst}(3x, x(x-1)) = 3x(3x-1).$$

This substitution function has a simple recursive definition:

Definition 6.4.3. The *substitution function* from Aexp × Aexp to Aexp is defined recursively on expressions, $e \in$ Aexp, as follows. Let f be any Aexp.

- **Base cases**:

$$\text{subst}(f, x) ::= f, \quad \text{(subbing } f \text{ for variable, } x\text{, just gives } f\text{)} \tag{6.14}$$
$$\text{subst}(f, \mathtt{k}) ::= \mathtt{k} \quad \text{(subbing into a numeral does nothing.)} \tag{6.15}$$

- **Constructor cases**:

$$\text{subst}(f, [\,e_1 + e_2\,]) ::= [\,\text{subst}(f, e_1) + \text{subst}(f, e_2)\,] \tag{6.16}$$
$$\text{subst}(f, [\,e_1 * e_2\,]) ::= [\,\text{subst}(f, e_1) * \text{subst}(f, e_2)\,] \tag{6.17}$$
$$\text{subst}(f, -[\,e_1\,]) ::= -[\,\text{subst}(f, e_1)\,]. \tag{6.18}$$

Here's how the recursive definition of the substitution function would find the result of substituting $3x$ for x in the $x(x-1)$:

$$\begin{aligned}
&\text{subst}(3x, x(x-1)) \\
&= \text{subst}([\,3 * x\,], [\,x * [\,x + -[\,1]\,]\,]) && \text{(unabbreviating)} \\
&= [\ \text{subst}([\,3 * x\,], x) * \\
&\qquad \text{subst}([\,3 * x\,], [\,x + -[\,1]\,])] && \text{(by Def 6.4.3 6.17)} \\
&= [\,[\,3 * x\,] * \text{subst}([\,3 * x\,], [\,x + -[\,1]\,])] && \text{(by Def 6.4.3 6.14)} \\
&= [\,[\,3 * x\,] * [\ \text{subst}([\,3 * x\,], x) \\
&\qquad + \text{subst}([\,3 * x\,], -[\,1]\,)]\,] && \text{(by Def 6.4.3 6.16)} \\
&= [\,[\,3 * x\,] * [\,[\,3 * x\,] + -[\ \text{subst}([\,3 * x\,], 1)]\,]\,] && \text{(by Def 6.4.3 6.14 \& 6.18)} \\
&= [\,[\,3 * x\,] * [\,[\,3 * x\,] + -[\,1]\,]\,] && \text{(by Def 6.4.3 6.15)} \\
&= 3x(3x-1) && \text{(abbreviation)}
\end{aligned}$$

Now suppose we have to find the value of $\text{subst}(3x, x(x-1))$ when $x = 2$. There are two approaches.

First, we could actually do the substitution above to get $3x(3x-1)$, and then we could evaluate $3x(3x-1)$ when $x = 2$, that is, we could recursively calculate $\text{eval}(3x(3x-1), 2)$ to get the final value 30. This approach is described by the expression

$$\text{eval}(\text{subst}(3x, x(x-1)), 2) \tag{6.19}$$

In programming jargon, this would be called evaluation using the *Substitution Model*. With this approach, the formula $3x$ appears twice after substitution, so the multiplication $3 \cdot 2$ that computes its value gets performed twice.

The other approach is called evaluation using the *Environment Model*. Namely, to compute the value of (6.19), we evaluate $3x$ when $x = 2$ using just 1 multiplication to get the value 6. Then we evaluate $x(x-1)$ when x has this value 6 to arrive at the value $6 \cdot 5 = 30$. This approach is described by the expression

$$\text{eval}(x(x-1), \text{eval}(3x, 2)). \tag{6.20}$$

The Environment Model only computes the value of $3x$ once, and so it requires one fewer multiplication than the Substitution model to compute (6.20). This is a good place to stop and work this example out yourself (Problem 6.18).

But how do we know that these final values reached by these two approaches, that is, the final integer values of (6.19) and (6.20), agree? In fact, we can prove pretty easily that these two approaches *always* agree by structural induction on the definitions of the two approaches. More precisely, what we want to prove is

Theorem 6.4.4. *For all expressions e, $f \in Aexp$ and $n \in \mathbb{Z}$,*

$$\text{eval}(\text{subst}(f, e), n) = \text{eval}(e, \text{eval}(f, n)). \tag{6.21}$$

Proof. The proof is by structural induction on e.[1]

Base cases:

- Case[x]

  The left hand side of equation (6.21) equals eval(f, n) by this base case in Definition 6.4.3 of the substitution function, and the right hand side also equals eval(f, n) by this base case in Definition 6.4.2 of eval.

- Case[k].

  The left hand side of equation (6.21) equals k by this base case in Definitions 6.4.3 and 6.4.2 of the substitution and evaluation functions. Likewise, the right hand side equals k by two applications of this base case in the Definition 6.4.2 of eval.

Constructor cases:

- Case[$[\, e_1 + e_2 \,]$]

  By the structural induction hypothesis (6.21), we may assume that for all $f \in$ Aexp and $n \in \mathbb{Z}$,

  $$\text{eval}(\text{subst}(f, e_i), n) = \text{eval}(e_i, \text{eval}(f, n)) \tag{6.22}$$

  for $i = 1, 2$. We wish to prove that

  $$\text{eval}(\text{subst}(f, [\, e_1 + e_2 \,]), n) = \text{eval}([\, e_1 + e_2 \,], \text{eval}(f, n)) \tag{6.23}$$

  The left hand side of (6.23) equals

  $$\text{eval}([\ \text{subst}(f, e_1) + \text{subst}(f, e_2)\,], n)$$

  by Definition 6.4.3.6.16 of substitution into a sum expression. But this equals

  $$\text{eval}(\text{subst}(f, e_1), n) + \text{eval}(\text{subst}(f, e_2), n)$$

[1] This is an example of why it's useful to notify the reader what the induction variable is—in this case it isn't n.

by Definition 6.4.2.(6.11) of eval for a sum expression. By induction hypothesis (6.22), this in turn equals

$$\text{eval}(e_1, \text{eval}(f, n)) + \text{eval}(e_2, \text{eval}(f, n)).$$

Finally, this last expression equals the right hand side of (6.23) by Definition 6.4.2.(6.11) of eval for a sum expression. This proves (6.23) in this case.

- Case[$[\, e_1 * e_2 \,]$] Similar.
- Case[$-[\, e_1 \,]$] Even easier.

This covers all the constructor cases, and so completes the proof by structural induction.

■

6.5 Induction in Computer Science

Induction is a powerful and widely applicable proof technique, which is why we've devoted two entire chapters to it. Strong induction and its special case of ordinary induction are applicable to any kind of thing with nonnegative integer sizes—which is an awful lot of things, including all step-by-step computational processes.

Structural induction then goes beyond number counting, and offers a simple, natural approach to proving things about recursive data types and recursive computation.

In many cases, a nonnegative integer size can be defined for a recursively defined datum, such as the length of a string, or the number of operations in an Aexp. It is then possible to prove properties of data by ordinary induction on their size. But this approach often produces more cumbersome proofs than structural induction.

In fact, structural induction is theoretically more powerful than ordinary induction. However, it's only more powerful when it comes to reasoning about infinite data types—like infinite trees, for example—so this greater power doesn't matter in practice. What does matter is that for recursively defined data types, structural induction is a simple and natural approach. This makes it a technique every computer scientist should embrace.

Problems for Section 6.1

Class Problems

Problem 6.1.
Prove that for all strings $r, s, t \in A^*$

$$(r \cdot s) \cdot t = r \cdot (s \cdot t).$$

Problem 6.2.
The *reversal* of a string is the string written backwards, for example, $\text{rev}(abcde) = edcba$.

(a) Give a simple recursive definition of $\text{rev}(s)$ based on the recursive definition 6.1.1 of $s \in A^*$ and using the concatenation operation 6.1.3.

(b) Prove that

$$\text{rev}(s \cdot t) = \text{rev}(t) \cdot \text{rev}(s),$$

for all strings $s, t \in A^*$.

Problem 6.3.
The Elementary 18.01 Functions (F18's) are the set of functions of one real variable defined recursively as follows:

Base cases:

- The identity function, $\text{id}(x) ::= x$ is an F18,
- any constant function is an F18,
- the sine function is an F18,

Constructor cases:
If f, g are F18's, then so are

1. $f + g$, fg, 2^g,
2. the inverse function f^{-1},
3. the composition $f \circ g$.

(a) Prove that the function $1/x$ is an F18.

Warning: Don't confuse $1/x = x^{-1}$ with the inverse id^{-1} of the identity function $\text{id}(x)$. The inverse id^{-1} is equal to id.

(b) Prove by Structural Induction on this definition that the Elementary 18.01 Functions are *closed under taking derivatives*. That is, show that if $f(x)$ is an F18, then so is $f' ::= df/dx$. (Just work out 2 or 3 of the most interesting constructor cases; you may skip the less interesting ones.)

Problem 6.4.

Here is a simple recursive definition of the set, E, of even integers:

Definition. Base case: $0 \in E$.

Constructor cases: If $n \in E$, then so are $n + 2$ and $-n$.

Provide similar simple recursive definitions of the following sets:

(a) The set $S ::= \{2^k 3^m 5^n \in \mathbb{N} \mid k, m, n \in \mathbb{N}\}$.

(b) The set $T ::= \{2^k 3^{2k+m} 5^{m+n} \in \mathbb{N} \mid k, m, n \in \mathbb{N}\}$.

(c) The set $L ::= \{(a, b) \in \mathbb{Z}^2 \mid (a - b) \text{ is a multiple of } 3\}$.

Let L' be the set defined by the recursive definition you gave for L in the previous part. Now if you did it right, then $L' = L$, but maybe you made a mistake. So let's check that you got the definition right.

(d) Prove by structural induction on your definition of L' that

$$L' \subseteq L.$$

(e) Confirm that you got the definition right by proving that

$$L \subseteq L'.$$

(f) See if you can give an *unambiguous* recursive definition of L.

Problem 6.5.

Definition. The recursive data type, binary-2PTG, of *binary trees* with leaf labels, L, is defined recursively as follows:

- **Base case:** $\langle \texttt{leaf}, l \rangle \in$ binary-2PTG, for all labels $l \in L$.
- **Constructor case:** If $G_1, G_2 \in$ binary-2PTG, then

$$\langle \texttt{bintree}, G_1, G_2 \rangle \in \text{binary-2PTG}.$$

The *size*, $|G|$, of $G \in$ binary-2PTG is defined recursively on this definition by:

- **Base case:**
$$|\langle \texttt{leaf}, l\rangle| ::= 1, \quad \text{for all } l \in L.$$
- **Constructor case:**
$$|\langle \texttt{bintree}, G_1, G_2\rangle| ::= |G_1| + |G_2| + 1.$$

For example, the size of the binary-2PTG, G, pictured in Figure 6.1, is 7.

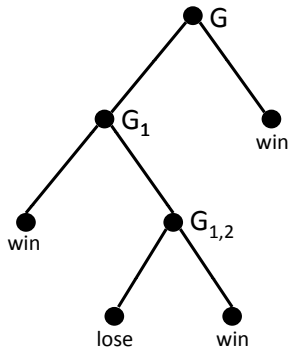


Figure 6.1 A picture of a binary tree G.

(a) Write out (using angle brackets and labels `bintree`, `leaf`, etc.) the binary-2PTG, G, pictured in Figure 6.1.

The value of flatten(G) for $G \in$ binary-2PTG is the sequence of labels in L of the leaves of G. For example, for the binary-2PTG, G, pictured in Figure 6.1,

$$\text{flatten}(G) = (\texttt{win}, \texttt{lose}, \texttt{win}, \texttt{win}).$$

(b) Give a recursive definition of flatten. (You may use the operation of *concatenation* (append) of two sequences.)

(c) Prove by structural induction on the definitions of flatten and size that

$$2 \cdot \text{length}(\text{flatten}(G)) = |G| + 1. \tag{6.24}$$

Homework Problems

Problem 6.6.
Let m, n be integers, not both zero. Define a set of integers, $L_{m,n}$, recursively as follows:

- **Base cases**: $m, n \in L_{m,n}$.
- **Constructor cases**: If $j, k \in L_{m,n}$, then
  1. $-j \in L_{m,n}$,
  2. $j + k \in L_{m,n}$.

Let L be an abbreviation for $L_{m,n}$ in the rest of this problem.

(a) Prove *by structural induction* that every common divisor of m and n also divides every member of L.

(b) Prove that any integer multiple of an element of L is also in L.

(c) Show that if $j, k \in L$ and $k \neq 0$, then $\text{rem}(j, k) \in L$.

(d) Show that there is a positive integer $g \in L$ which divides every member of L.
Hint: The least positive integer in L.

(e) Conclude that $g = \text{GCD}(m, n)$ for g from part (d).

Problem 6.7.

Definition. Define the number, $\#_c(s)$, of occurrences of the character $c \in A$ in the string s recursively on the definition of $s \in A^*$:
base case: $\#_c(\lambda) ::= 0$.
constructor case:

$$\#_c(\langle a, s\rangle) ::= \begin{cases} \#_c(s) & \text{if } a \neq c, \\ 1 + \#_c(s) & \text{if } a = c. \end{cases}$$

Prove by structural induction that for all $s, t \in A^*$ and $c \in A$

$$\#_c(s \cdot t) = \#_c(s) + \#_c(t).$$

Figure 6.2 Constructing the Koch Snowflake.

Problem 6.8.
Fractals are an example of mathematical objects that can be defined recursively. In this problem, we consider the Koch snowflake. Any Koch snowflake can be constructed by the following recursive definition.

- **Base case**: An equilateral triangle with a positive integer side length is a Koch snowflake.
- **Constructor case**: Let K be a Koch snowflake, and let l be a line segment on the snowflake. Remove the middle third of l, and replace it with two line segments of the same length as is done in Figure 6.2

  The resulting figure is also a Koch snowflake.

Prove by structural induction that the area inside any Koch snowflake is of the form $q\sqrt{3}$, where q is a rational number.

Problem 6.9.
Let L be some convenient set whose elements will be called *labels*. The labeled binary trees, LBT's, are defined recursively as follows:

Definition. Base case: if l is a label, then $\langle l, \texttt{leaf}\rangle$ is an LBT, and

Constructor case: if B and C are LBT's, then $\langle l, B, C\rangle$ is an LBT.

The *leaf-labels* and *internal-labels* of an LBT are defined recursively in the obvious way:

Definition. Base case: The set of leaf-labels of the LBT $\langle l, \texttt{leaf}\rangle$ is $\{l\}$, and its set of internal-labels is the empty set.

Constructor case: The set of leaf labels of the LBT $\langle l, B, C\rangle$ is the union of the leaf-labels of B and of C; the set of internal-labels is the union of $\{l\}$ and the sets of internal-labels of B and of C.

The set of *labels* of an LBT is the union of its leaf- and internal-labels.
The LBT's with *unique* labels are also defined recursively:

Definition. Base case: The LBT $\langle l, \texttt{leaf} \rangle$ has *unique labels*.

Constructor case: If B and C are LBT's with unique labels, no label of B is a label C and vice-versa, and l is not a label of B or C, then $\langle l, B, C \rangle$ has *unique labels*.

If B is an LBT, let n_B be the number of distinct i**n**ternal-labels appearing in B and f_B be the number of distinct lea**f** labels of B. Prove by structural induction that

$$f_B = n_B + 1 \tag{6.25}$$

for all LBT's B with unique labels. This equation can obviously fail if labels are not unique, so your proof had better use uniqueness of labels at some point; be sure to indicate where.

Exam Problems

Problem 6.10.

The Arithmetic Trig Functions (*Atrig*'s) are the set of functions of one real variable defined recursively as follows:

Base cases:

- The identity function, $\mathrm{id}(x) ::= x$ is an *Atrig*,
- any constant function is an *Atrig*,
- the sine function is an *Atrig*,

Constructor cases:

If f, g are *Atrig*'s, then so are

1. $f + g$
2. $f \cdot g$
3. the composition $f \circ g$.

Prove by structural induction on this definition that if $f(x)$ is an *Atrig*, then so is $f' ::= df/dx$.

Problem 6.11.

Definition. The set RAF of *rational functions* of one real variable is the set of functions defined recursively as follows:

Base cases:

- The identity function, $\mathrm{id}(r) ::= r$ for $r \in \mathbb{R}$ (the real numbers), is an RAF,
- any constant function on $\mathbb{R}$ is an RAF.

Constructor cases: If f, g are RAF's, then so is $f \circledast g$, where $\circledast$ is one of the operations

1. addition, $+$,
2. multiplication, $\cdot$, and
3. division $/$.

(a) Prove by structural induction that RAF is closed under composition. That is, using the induction hypothesis,

$$P(h) ::= \forall g \in \mathrm{RAF}\,.\, h \circ g \in \mathrm{RAF}, \tag{6.26}$$

prove that $P(h)$ holds for all $h \in \mathrm{RAF}$. Make sure to indicate explicitly

- each of the base cases, and
- each of the constructor cases. *Hint:* One proof in terms of $\circledast$ covers all three cases.

(b) Briefly indicate where a proof would break down using the very similar induction hypothesis

$$Q(g) ::= \forall h \in \mathrm{RAF}\,.\, h \circ g \in \mathrm{RAF}\,.$$

Problems for Section 6.2

Practice Problems

Problem 6.12.
Define the sets F_1 and F_2 recursively:

- F_1:
  - $5 \in F_1$,
  - if $n \in F_1$, then $5n \in F_1$.

- F_2:
  - $5 \in F_2$,
  - if $n, m \in F_1$, then $nm \in F_2$.

(a) Show that one of these definitions is technically *ambiguous*. (Remember that "ambiguous recursive definition" has a technical mathematical meaning which does not imply that the ambiguous definition is unclear.)

(b) Briefly explain what advantage unambiguous recursive definitions have over ambiguous ones.

(c) A way to prove that $F_1 = F_2$, is to show firat that $F_1 \subseteq F_2$ and second that $F_2 \subseteq F_1$. One of these containments follows easily by structural induction. Which one? What would be the induction hypothesis? (You do not need to complete a proof.)

Problem 6.13. (a) To prove that the set RecMatch, of matched strings of Definition 6.2.1 equals the set AmbRecMatch of ambiguous matched strings of Definition 6.2.2, you could first prove that

$$\forall r \in \text{RecMatch}.\ r \in \text{AmbRecMatch},$$

and then prove that

$$\forall u \in \text{AmbRecMatch}.\ u \in \text{RecMatch}.$$

Of these two statements, circle the one that would be simpler to prove by structural induction directly from the definitions.

(b) Suppose structural induction was being used to prove that AmbRecMatch $\subseteq$ RecMatch. Circle the one predicate below that would fit the format for a structural induction hypothesis in such a proof.

- $P_0(n) ::= |s| \leq n$ IMPLIES $s \in$ RecMatch.
- $P_1(n) ::= |s| \leq n$ IMPLIES $s \in$ AmbRecMatch.
- $P_2(s) ::= s \in$ RecMatch.
- $P_3(s) ::= s \in$ AmbRecMatch.
- $P_4(s) ::=$ ($s \in$ RecMatch IMPLIES $s \in$ AmbRecMatch).

(c) The recursive definition AmbRecMatch is ambiguous because it allows the $s \cdot t$ constructor to apply when s or t is the empty string. But even fixing that, ambiguity remains. Demonstrate this by giving two different derivations for the string "[][][] according to AmbRecMatch but only using the $s \cdot t$ constructor when $s \neq \lambda$ and $t \neq \lambda$.

Class Problems

Problem 6.14.

Let p be the string [ ]. A string of brackets is said to be *erasable* iff it can be reduced to the empty string by repeatedly erasing occurrences of p. For example, here's how to erase the string [[[]][]][]:

$$[[[]][]][] \to [[]] \to [] \to \lambda.$$

On the other hand the string [ ]][[[[[]] is not erasable because when we try to erase, we get stuck:][[[:

$$[]][[[[[]] \to][[[[] \to][[[\not\to$$

Let Erasable be the set of erasable strings of brackets. Let RecMatch be the recursive data type of strings of *matched* brackets given in Definition 6.2.1

(a) Use structural induction to prove that

$$\text{RecMatch} \subseteq \text{Erasable}.$$

(b) Supply the missing parts (labeled by "(*)") of the following proof that

$$\text{Erasable} \subseteq \text{RecMatch}.$$

Proof. We prove by strong induction that every length n string in Erasable is also in RecMatch. The induction hypothesis is

$$P(n) ::= \forall x \in \text{Erasable}.\ |x| = n \text{ IMPLIES } x \in \text{RecMatch}.$$

Base case:

(*) What is the base case? Prove that P is true in this case.

Inductive step: To prove $P(n+1)$, suppose $|x| = n+1$ and $x \in$ Erasable. We need to show that $x \in$ RecMatch.

Let's say that a string y is an *erase* of a string z iff y is the result of erasing a *single* occurrence of p in z.

Since $x \in$ Erasable and has positive length, there must be an erase, $y \in$ Erasable, of x. So $|y| = n - 1 \geq 0$, and since $y \in$ Erasable, we may assume by induction hypothesis that $y \in$ RecMatch.

Now we argue by cases:

Case (y is the empty string):

(*) Prove that $x \in$ RecMatch in this case.

Case ($y = [\, s \,] t$ for some strings $s, t \in$ RecMatch): Now we argue by subcases.

- **Subcase**($x = py$):
  (*) Prove that $x \in$ RecMatch in this subcase.
- **Subcase** (x is of the form $[\, s' \,] t$ where s is an erase of s'):
  Since $s \in$ RecMatch, it is erasable by part (b), which implies that $s' \in$ Erasable. But $|s'| < |x|$, so by induction hypothesis, we may assume that $s' \in$ RecMatch. This shows that x is the result of the constructor step of RecMatch, and therefore $x \in$ RecMatch.
- **Subcase** (x is of the form $[\, s \,] t'$ where t is an erase of t'):
  (*) Prove that $x \in$ RecMatch in this subcase.

(*) Explain why the above cases are sufficient.

This completes the proof by strong induction on n, so we conclude that $P(n)$ holds for all $n \in \mathbb{N}$. Therefore $x \in$ RecMatch for every string $x \in$ Erasable. That is, Erasable $\subseteq$ RecMatch. Combined with part (a), we conclude that

$$\text{Erasable} = \text{RecMatch}.$$

■

Problem 6.15. **(a)** Prove that the set RecMatch, of matched strings of Definition 6.2.1 is closed under string concatenation. Namely, if $s, t \in$ RecMatch, then $s \cdot t \in$ RecMatch.

(b) Prove AmbRecMatch $\subseteq$ RecMatch, where AmbRecMatch is the set of ambiguous matched strings of Definition 6.2.2.

(c) Prove that RecMatch $=$ AmbRecMatch.

Homework Problems

Problem 6.16.
One way to determine if a string has matching brackets, that is, if it is in the set, RecMatch, of Definition 6.2.1 is to start with 0 and read the string from left to right, adding 1 to the count for each left bracket and subtracting 1 from the count for each right bracket. For example, here are the counts for two sample strings:

[	]	]	[	[	[	[	[	]	]	]	]
0	1	0	−1	0	1	2	3	4	3	2	1	0

[	[	[	]	]	[	]	]	[	]
0	1	2	3	2	1	2	1	0	1	0

A string has a *good count* if its running count never goes negative and ends with 0. So the second string above has a good count, but the first one does not because its count went negative at the third step. Let

$$\text{GoodCount} ::= \{s \in \{\text{]}, \text{[}\}^* \mid s \text{ has a good count}\}.$$

The empty string has a length 0 running count we'll take as a good count by convention, that is, $\lambda \in \text{GoodCount}$. The matched strings can now be characterized precisely as this set of strings with good counts.

(a) Prove that GoodCount contains RecMatch by structural induction on the definition of RecMatch.

(b) Conversely, prove that RecMatch contains GoodCount.

Hint: By induction on the length of strings in GoodCount. Consider when the running count equals 0 for the second time.

Problems for Section 6.3

Homework Problems

Problem 6.17.
One version of the the Ackermann function, $A : \mathbb{N}^2 \to \mathbb{N}$, is defined recursively by the following rules:

$$A(m,n) ::= 2n, \qquad \text{if } m = 0 \text{ or } n \leq 1 \qquad \text{(A-base)}$$
$$A(m,n) ::= A(m-1, A(m,n-1)), \qquad \text{otherwise.} \qquad \text{(AA)}$$

Prove that if $B : \mathbb{N}^2 \to \mathbb{N}$ is a partial function that satisfies this same definition, then B is total and $B = A$.

Problems for Section 6.4

Practice Problems

Problem 6.18. **(a)** Write out the evaluation of

$$\text{eval}(\text{subst}(3x, x(x-1)), 2)$$

according to the Environment Model and the Substitution Model, indicating where the rule for each case of the recursive definitions of eval(,) and [:=] or substitution is first used. Compare the number of arithmetic operations and variable lookups.

(b) Describe an example along the lines of part (a) where the Environment Model would perform 6 fewer multiplications than the Substitution model. You need *not* carry out the evaluations.

(c) Describe an example along the lines of part (a) where the Substitution Model would perform 6 fewer multiplications than the Environment model. You need *not* carry out the evaluations.

Homework Problems

Problem 6.19. **(a)** Give a recursive definition of a function erase(e) that erases all the symbols in $e \in$ Aexp but the brackets. For example

$$\text{erase}([[\,3 * [\,x * x\,]\,] + [[\,2 * x\,] + 1\,]\,]) = [[[\,]\,][[\,2 * x\,] + 1\,]\,].$$

(b) Prove that erase(e) $\in$ RecMatch for all $e \in$ Aexp.

(c) Give an example of a small string $s \in$ RecMatch such that $[\,s\,] \neq \text{erase}(e)$ for any $e \in$ Aexp.

v

Problem 6.20.
We're going to characterize a large category of games as a recursive data type and then prove, by structural induction, a fundamental theorem about game strategies. The games we'll consider are known as *deterministic games of perfect information*,

because at each move, the complete game situation is known to the players, and this information completely determines how the rest of the game can be played. Games like chess, checkers, GO, and tic-tac-toe fit this description. In contrast, most card games do not fit, since card players usually do not know exactly what cards belong to the other players. Neither do games involving random features like dice rolls, since a player's move does not uniquely determine what happens next.

Chess counts as a deterministic game of perfect information because at any point of play, both players know whose turn it is to move and the location of every chess piece on the board.[2] At the start of the game, there are 20 possible first moves: the player with the White pieces can move one of his eight pawns forward 1 or 2 squares or one of his two knights forward and left or forward and right. For the second move, the Black player can make one of the 20 corresponding moves of his own pieces. The White player would then make the third move, but now the number of possible third moves depends on what the first two moves happened to be.

A nice way to think of these games is to regard each game situation as a game in its own right. For example, after five moves in a chess game, we think of the players as being at the start of a new "chess" game determined by the current board position and the fact that it is Black's turn to make the next move.

At the end of a chess game, we might assign a score of 1 if the White player won, -1 if White lost, and 0 if the game ended in a stalemate (a tie). Now we can say that White's objective is to maximize the final score and Black's objective is to minimize it. We might also choose to score the game in a more elaborate way, taking into account not only who won, but also how many moves the game took, or the final board configuration.

This leads to an elegant abstraction of this kind of game. We suppose there are two players, called the *max-player* and the *min-player*, whose aim is, respectively, to maximize and minimize the final score. A game will specify its set of possible first moves, each of which will simply be another game. A game with no possible moves is called an *ended game*, and will just have a final score. Strategically, all that matters about an ended game is its score. If a game is not ended, it will have a label `max` or `min` indicating which player is supposed to move first.

This motivates the following formal definition:

Definition. Let V be a nonempty set of real numbers. The class VG of *V-valued deterministic max-min games of perfect information* is defined recursively as fol-

[2] In order to prevent the possibility of an unending game, chess rules specify a limit on the number of moves, or a limit on the number of times a given board postion may repeat. So the number of moves or the number of position repeats would count as part of the game situation known to both players.

lows:

Base case: A value $v \in V$ is a VG, and is called an *ended game*.

Constructor case: If $\{G_0, G_1, \ldots\}$ is a nonempty set of VG's, and a is a label equal to `max` or `min`, then

$$G ::= (a, \{G_0, G_1, \ldots\})$$

is a VG. Each game G_i is called a possible *first move* of G.

In all the games like this that we're familiar with, there are only a finite number of possible first moves. It's worth noting that the definition of VG does not require this. Since finiteness is not needed to prove any of the results below, it would arguably be misleading to assume it. Later, we'll suggest how games with an infinite number of possible first moves might come up.

A *play* of a game is a sequence of legal moves that either goes on forever or finishes with an ended game. More formally:

Definition. A *play* of a game $G \in$ VG is defined recursively on the definition of VG:

Base case: (G is an ended game.) Then the length one sequence (G) is a *play* of G.

Constructor case: (G is not an ended game.) Then a *play* of G is a sequence that starts with a possible first move, G_i, of G and continues with the elements of a play of G_i.

If a play does not go on forever, its *payoff* is defined to be the value it ends with.

Let's first rule out the possibility of playing forever. Namely, every play will have a payoff.

(a) Prove that every play of a $G \in$ VG is a finite sequence that ends with a value in V. *Hint:* By structural induction on the definition of VG.

A *strategy* for a game is a rule that tells a player which move to make when it's his turn. Formally:

Definition. If a is one of the labels `max` or `min`, then an *a-strategy* is a function $s : \text{VG} \to \text{VG}$ such that

$$s(G) \text{ is } \begin{cases} \text{a first move of } G & \text{if } G \text{ has label } a, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Any pair of strategies for the two players determines a unique play of a game, and hence a unique payoff, in an obvious way. Namely, when it is a player's turn to move in a game G, he chooses the move specified by his strategy. A strategy for the max-player is said to *ensure* payoff v when, paired with *any* strategy for the min-player, the resulting payoff is *at least* v. Dually, a strategy for the min-player *caps* payoff at v when, paired with any strategy for the max-player, the resulting payoff is *at most* v.

Assuming for simplicity that the set V of possible values of a game is finite, the WOP (Section 2.4) implies there will be a strategy for the max-player that ensures the largest possible payoff; this is called the *max-ensured-value* of the game. Dually, there will also be a strategy for the min-player that caps the payoff at the smallest possible value, which is called the *min-capped-value* of the game.

The max-ensured-value of course cannot be larger than the min-capped-value. A unique value can be assigned to a game when these two values agree:

Definition. If the max-ensured-value and min-capped-value of a game are equal, their common value is called the *value of the game*.

So if both players play optimally in a game with that has a value, v, then there is actually no point in playing. Since the payoff is ensured to be at least v and is also capped to be at most v, it must be exactly v. So the min-player may as well skip playing and simply pay v to the max-player (a negative payment means the max-player is paying the min-player).

The punch line of our story is that the max-ensured-value and the min-capped-value are *always* equal.

Theorem (Fundamental Theorem for Deterministic Min-Max Games of Perfect Information)**.** *Let V be a finite set of real numbers. Every V-valued deterministic max-min game of perfect information has a value.*

(b) Prove this Fundamental Theorem for VG's by structural induction.

(c) Conclude immediately that in chess, there is a winning strategy for White, or a winning strategy for Black, or both players have strategies that guarantee at least a stalemate. (The only difficulty is that no one knows which case holds.)

So where do we come upon games with an infinite number of first moves? Well, suppose we play a tournament of n chess games for some positive integer n. This tournament will be a VG if we agree on a rule for combining the payoffs of the n individual chess games into a final payoff for the whole tournament.

There still are only a finite number of possible moves at any stage of the n-game chess tournament, but we can define a *meta-chess-tournament*, whose first move is

a choice of any positive integer n, after which we play an n-game tournament. Now the meta-chess-tournament has an infinite number of first moves.

Of course only the first move in the meta-chess-tournament is infinite, but then we could set up a tournament consisting of n meta-chess-tournaments. This would be a game with n possible infinite moves. And then we could have a *meta-meta*-chess-tournament whose first move was to choose how many meta-chess-tournaments to play. This meta-meta-chess-tournament will have an infinite number of infinite moves. Then we could move on to meta-meta-meta-chess-tournaments ....

As silly or weird as these meta games may seem, their weirdness doesn't disqualify the Fundamental Theorem: each of these games will still have a value.

(d) State some reasonable generalization of the Fundamental Theorem to games with an infinite set V of possible payoffs. *Optional*: Prove your generalization.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015

7 Infinite Sets

This chapter is about infinite sets and some challenges in proving things about them.

Wait a minute! Why bring up infinity in a Mathematics for *Computer Science* text? After all, any data set in a computer is limited by the size of the computer's memory, and there is a bound on the possible size of computer memory, for the simple reason that the universe is (or at least appears to be) bounded. So why not stick with *finite* sets of some large, but bounded, size? This is a good question, but let's see if we can persuade you that dealing with infinite sets is inevitable.

You may not have noticed, but up to now you've already accepted the routine use of the integers, the rationals and irrationals, and sequences of them—infinite sets, all. Further, do you really want Physics or the other sciences to give up the real numbers on the grounds that only a bounded number of bounded measurements can be made in a bounded universe? It's pretty convincing—and a lot simpler—to ignore such big and uncertain bounds (the universe seems to be getting bigger all the time) and accept theories using real numbers.

Likewise in computer science, it's implausible to think that writing a program to add nonnegative integers with up to as many digits as, say, the stars in the sky—billions of galaxies each with billions of stars—would be different from writing a program that would add *any* two integers, no matter how many digits they had. The same is true in designing a compiler: it's neither useful nor sensible to make use of the fact that in a bounded universe, only a bounded number of programs will ever be compiled.

Infinite sets also provide a nice setting to practice proof methods, because it's harder to sneak in unjustified steps under the guise of intuition. And there has been a truly astonishing outcome of studying infinite sets. Their study led to the discovery of fundamental, logical limits on what computers can possibly do. For example, in Section 7.2, we'll use reasoning developed for infinite sets to prove that it's impossible to have a perfect type-checker for a programming language.

So in this chapter, we ask you to bite the bullet and start learning to cope with infinity.

7.1 Infinite Cardinality

In the late nineteenth century, the mathematician Georg Cantor was studying the convergence of Fourier series and found some series that he wanted to say converged "most of the time," even though there were an infinite number of points where they didn't converge. As a result, Cantor needed a way to compare the size of infinite sets. To get a grip on this, he got the idea of extending the Mapping Rule Theorem 4.5.4 to infinite sets: he regarded two infinite sets as having the "same size" when there was a bijection between them. Likewise, an infinite set A should be considered "as big as" a set B when A surj B. So we could consider A to be "strictly smaller" than B, which we abbreviate as A strict B, when A is *not* "as big as" B:

Definition 7.1.1. A strict B iff NOT(A surj B).

On finite sets, this strict relation really does mean "strictly smaller." This follows immediately from the Mapping Rule Theorem 4.5.4.

Corollary 7.1.2. *For finite sets A, B,*

$$A \text{ strict } B \quad \textit{iff} \quad |A| < |B|.$$

Proof.

$$\begin{array}{lll} A \text{ strict } B & \text{iff} & \text{NOT}(A \text{ surj } B) \qquad\qquad (\text{Def } 7.1.1) \\ & \text{iff} & \text{NOT}(|A| \geq |B|) \qquad (\text{Theorem } 4.5.4.(4.5)) \\ & \text{iff} & |A| < |B|. \end{array}$$

■

Cantor got diverted from his study of Fourier series by his effort to develop a theory of infinite sizes based on these ideas. His theory ultimately had profound consequences for the foundations of mathematics and computer science. But Cantor made a lot of enemies in his own time because of his work: the general mathematical community doubted the relevance of what they called "Cantor's paradise" of unheard-of infinite sizes.

A nice technical feature of Cantor's idea is that it avoids the need for a definition of what the "size" of an infinite set might be—all it does is compare "sizes."

Warning: We haven't, and won't, define what the "size" of an infinite set is. The definition of infinite "sizes" requires the definition of some infinite sets called

ordinals with special well-ordering properties. The theory of ordinals requires getting deeper into technical set theory than we want to go, and we can get by just fine without defining infinite sizes. All we need are the "as big as" and "same size" relations, surj and bij, between sets.

But there's something else to watch out for: we've referred to surj as an "as big as" relation and bij as a "same size" relation on sets. Of course, most of the "as big as" and "same size" properties of surj and bij on finite sets do carry over to infinite sets, but *some important ones don't*—as we're about to show. So you have to be careful: don't assume that surj has any particular "as big as" property on *infinite* sets until it's been proved.

Let's begin with some familiar properties of the "as big as" and "same size" relations on finite sets that do carry over exactly to infinite sets:

Lemma 7.1.3. *For any sets,* A, B, C*,*

1. A surj B *iff* B inj A.
2. *If* A surj B *and* B surj C*, then* A surj C.
3. *If* A bij B *and* B bij C*, then* A bij C.
4. A bij B *iff* B bij A.

Part 1. follows from the fact that R has the $[\leq 1 \text{ out}, \geq 1 \text{ in}]$ surjective function property iff R^{-1} has the $[\geq 1 \text{ out}, \leq 1 \text{ in}]$ total, injective property. Part 2. follows from the fact that compositions of surjections are surjections. Parts 3. and 4. follow from the first two parts because R is a bijection iff R and R^{-1} are surjective functions. We'll leave verification of these facts to Problem 4.22.

Another familiar property of finite sets carries over to infinite sets, but this time some real ingenuity is needed to prove it:

Theorem 7.1.4. *[Schröder-Bernstein] For any sets* A, B*, if* A surj B *and* B surj A*, then* A bij B.

That is, the Schröder-Bernstein Theorem says that if A is at least as big as B and conversely, B is at least as big as A, then A is the same size as B. Phrased this way, you might be tempted to take this theorem for granted, but that would be a mistake. For infinite sets A and B, the Schröder-Bernstein Theorem is actually pretty technical. Just because there is a surjective function $f : A \to B$—which need not be a bijection—and a surjective function $g : B \to A$—which also need not be a bijection—it's not at all clear that there must be a bijection $e : A \to B$. The idea is to construct e from parts of both f and g. We'll leave the actual construction to Problem 7.11.

Another familiar set property is that for any two sets, either the first is at least as big as the second, or vice-versa. For finite sets this follows trivially from the Mapping Rule. It's actually still true for infinite sets, but assuming it was obvious would be mistaken again.

Theorem 7.1.5. *For* all *sets* A, B,

$$A \text{ surj } B \quad \text{OR} \quad B \text{ surj } A.$$

Theorem 7.1.5 lets us prove that another basic property of finite sets carries over to infinite ones:

Lemma 7.1.6.

$$A \text{ strict } B \quad \text{AND} \quad B \text{ strict } C \tag{7.1}$$

implies

$$A \text{ strict } C$$

for all sets A, B, C.

Proof. (of Lemma 7.1.6)

Suppose 7.1 holds, and assume for the sake of contradiction that NOT(A strict C), which means that A surj C. Now since B strict C, Theorem 7.1.5 lets us conclude that C surj B. So we have

$$A \text{ surj } C \quad \text{AND} \quad C \text{ surj } B,$$

and Lemma 7.1.3.2 lets us conclude that A surj B, contradicting the fact that A strict B. ■

We're omitting a proof of Theorem 7.1.5 because proving it involves technical set theory—typically the theory of ordinals again—that we're not going to get into. But since proving Lemma 7.1.6 is the only use we'll make of Theorem 7.1.5, we hope you won't feel cheated not to see a proof.

7.1.1 Infinity is different

A basic property of finite sets that does *not* carry over to infinite sets is that adding something new makes a set bigger. That is, if A is a finite set and $b \notin A$, then $|A \cup \{b\}| = |A| + 1$, and so A and $A \cup \{b\}$ are not the same size. But if A is infinite, then these two sets *are* the same size!

Lemma 7.1.7. *Let A be a set and $b \notin A$. Then A is infinite iff A* bij *$A \cup \{b\}$.*

Proof. Since A is *not* the same size as $A \cup \{b\}$ when A is finite, we only have to show that $A \cup \{b\}$ *is* the same size as A when A is infinite.

That is, we have to find a bijection between $A \cup \{b\}$ and A when A is infinite. Here's how: since A is infinite, it certainly has at least one element; call it a_0. But since A is infinite, it has at least two elements, and one of them must not equal to a_0; call this new element a_1. But since A is infinite, it has at least three elements, one of which must not equal both a_0 and a_1; call this new element a_2. Continuing in this way, we conclude that there is an infinite sequence $a_0, a_1, a_2, \ldots, a_n, \ldots$ of different elements of A. Now it's easy to define a bijection $e : A \cup \{b\} \to A$:

$$\begin{aligned} e(b) &::= a_0, \\ e(a_n) &::= a_{n+1} && \text{for } n \in \mathbb{N}, \\ e(a) &::= a && \text{for } a \in A - \{b, a_0, a_1, \ldots\}. \end{aligned}$$

■

7.1.2 Countable Sets

A set, C, is *countable* iff its elements can be listed in order, that is, the elements in C are precisely the elements in the sequence

$$c_0, c_1, \ldots, c_n, \ldots.$$

Assuming no repeats in the list, saying that C can be listed in this way is formally the same as saying that the function, $f : \mathbb{N} \to C$ defined by the rule that $f(i) ::= c_i$, is a bijection.

Definition 7.1.8. A set, C, is *countably infinite* iff $\mathbb{N}$ bij C. A set is *countable* iff it is finite or countably infinite.

We can also make an infinite list using just a finite set of elements if we allow repeats. For example, we can list the elements in the three-element set $\{2, 4, 6\}$ as

$$2, 4, 6, 6, 6, \ldots.$$

This simple observation leads to an alternative characterization of countable sets that does not make separate cases of finite and infinite sets. Namely, a set C is countable iff there is a list

$$c_0, c_1, \ldots, c_n, \ldots$$

of the elements of C, possibly with repeats.

Lemma 7.1.9. *A set, C, is countable iff* $\mathbb{N}$ surj C. *In fact, a nonempty set C is countable iff there is a* total *surjective function $g : \mathbb{N} \to C$.*

The proof is left to Problem 7.12.

The most fundamental countably infinite set is the set, $\mathbb{N}$, itself. But the set, $\mathbb{Z}$, of *all* integers is also countably infinite, because the integers can be listed in the order:

$$0, -1, 1, -2, 2, -3, 3, \ldots. \tag{7.2}$$

In this case, there is a simple formula for the nth element of the list (7.2). That is, the bijection $f : \mathbb{N} \to \mathbb{Z}$ such that $f(n)$ is the nth element of the list can be defined as:

$$f(n) ::= \begin{cases} n/2 & \text{if } n \text{ is even,} \\ -(n+1)/2 & \text{if } n \text{ is odd.} \end{cases}$$

There is also a simple way to list all *pairs* of nonnegative integers, which shows that $(\mathbb{N} \times \mathbb{N})$ is also countably infinite (Problem 7.16). From this, it's a small step to reach the conclusion that the set, $\mathbb{Q}^{\geq 0}$, of nonnegative rational numbers is countable. This may be a surprise—after all, the rationals densely fill up the space between integers, and for any two, there's another in between. So it might seem as though you couldn't write out all the rationals in a list, but Problem 7.10 illustrates how to do it. More generally, it is easy to show that countable sets are closed under unions and products (Problems 7.1 and 7.16) which implies the countability of a bunch of familiar sets:

Corollary 7.1.10. *The following sets are countably infinite:*

$$\mathbb{Z}^+, \mathbb{Z}, \mathbb{N} \times \mathbb{N}, \mathbb{Q}^+, \mathbb{Z} \times \mathbb{Z}, \mathbb{Q}.$$

A small modification of the proof of Lemma 7.1.7 shows that countably infinite sets are the "smallest" infinite sets, or more precisely that if A is an infinite set, and B is countable, then A surj B (see Problem 7.9).

Also, since adding one new element to an infinite set doesn't change its size, you can add any *finite* number of elements without changing the size by simply adding one element after another. Something even stronger is true: you can add a *countably* infinite number of new elements to an infinite set and still wind up with just a set of the same size (Problem 7.13).

By the way, it's a common mistake to think that, because you can add any finite number of elements to an infinite set and have a bijection with the original set, that you can also throw in infinitely many new elements. In general it isn't true that just because it's OK to do something any finite number of times, it also OK to do it an infinite number of times. For example, starting from 3, you can increment by 1 any finite number of times, and the result will be some integer greater than or equal to 3. But if you increment an infinite number of times, you don't get an integer at all.

7.1.3 Power sets are strictly bigger

Cantor's astonishing discovery was that *not all infinite sets are the same size*. In particular, he proved that for any set, A, the power set, $\text{pow}(A)$, is "strictly bigger" than A. That is,

Theorem 7.1.11. *[Cantor] For any set, A,*

$$A \text{ strict } \text{pow}(A).$$

Proof. To show that A is strictly smaller than $\text{pow}(A)$, we have to show that if g is a function from A to $\text{pow}(A)$, then g is *not* a surjection. To do this, we'll simply find a subset, $A_g \subseteq A$ that is not in the range of g. The idea is, for any element $a \in A$, to look at the set $g(a) \subseteq A$ and ask whether or not a happens to be in $g(a)$. First, define

$$A_g ::= \{a \in A \mid a \notin g(a)\}.$$

A_g is now a well-defined subset of A, which means it is a member of $\text{pow}(A)$. But A_g can't be in the range of g, because if it were, we would have

$$A_g = g(a_0)$$

for some $a_0 \in A$, so by definition of A_g,

$$a \in g(a_0) \quad \text{iff} \quad a \in A_g \quad \text{iff} \quad a \notin g(a)$$

for all $a \in A$. Now letting $a = a_0$ yields the contradiction

$$a_0 \in g(a_0) \quad \text{iff} \quad a_0 \notin g(a_0).$$

So g is not a surjection, because there is an element in the power set of A, specifically the set A_g, that is not in the range of g. ■

Cantor's Theorem immediately implies:

Corollary 7.1.12. $\text{pow}(\mathbb{N})$ *is uncountable.*

The bijection between subsets of an n-element set and the length n bit-strings, $\{0,1\}^n$, used to prove Theorem 4.5.5, carries over to a bijection between subsets of a countably infinite set and the infinite bit-strings, $\{0,1\}^\omega$. That is,

$$\text{pow}(\mathbb{N}) \text{ bij } \{0,1\}^\omega.$$

This immediately implies

Corollary 7.1.13. $\{0,1\}^\omega$ *is uncountable.*

More Countable and Uncountable Sets

Once we have a few sets we know are countable or uncountable, we can get lots more examples using Lemma 7.1.3. In particular, we can appeal to the following immediate corollary of the Lemma:

Corollary 7.1.14.

(a) If U is an uncountable set and A surj *U, then A is uncountable.*

(b) If C is a countable set and C surj *A, then A is countable.*

For example, now that we know that the set $\{0,1\}^\omega$ of infinite bit strings is uncountable, it's a small step to conclude that

Corollary 7.1.15. *The set $\mathbb{R}$ of real numbers is uncountable.*

To prove this, think about the infinite decimal expansion of a real number:

$$\begin{aligned}
\sqrt{2} &= 1.4142\ldots,\\
5 &= 5.000\ldots,\\
1/10 &= 0.1000\ldots,\\
1/3 &= 0.333\ldots,\\
1/9 &= 0.111\ldots,\\
4\frac{1}{99} &= 4.010101\ldots.
\end{aligned}$$

Let's map any real number r to the infinite bit string $b(r)$ equal to the sequence of bits in the decimal expansion of r, starting at the decimal point. If the decimal expansion of r happens to contain a digit other than 0 or 1, leave $b(r)$ undefined. For example,

$$\begin{aligned}
b(5) &= 000\ldots,\\
b(1/10) &= 1000\ldots,\\
b(1/9) &= 111\ldots,\\
b(4\tfrac{1}{99}) &= 010101\ldots\\
b(\sqrt{2}), b(1/3) &\text{ are undefined.}
\end{aligned}$$

Now b is a function from real numbers to infinite bit strings.[1] It is not a total function, but it clearly is a surjection. This shows that

$$\mathbb{R} \text{ surj } \{0,1\}^\omega,$$

and the uncountability of the reals now follows by Corollary 7.1.14.(a).

For another example, let's prove

Corollary 7.1.16. *The set $(\mathbb{Z}^+)^*$ of all finite sequences of positive integers is countable.*

To prove this, think about the prime factorization of a nonnegative integer:

$$20 = 2^2 \cdot 3^0 \cdot 5^1 \cdot 7^0 \cdot 11^0 \cdot 13^0 \cdots,$$
$$6615 = 2^0 \cdot 3^3 \cdot 5^1 \cdot 7^2 \cdot 11^0 \cdot 13^0 \cdots.$$

Let's map any nonnegative integer n to the finite sequence $e(n)$ of nonzero exponents in its prime factorization. For example,

$$\begin{aligned} e(20) &= (2,1), \\ e(6615) &= (3,1,2), \\ e(5^{13} \cdot 11^9 \cdot 47^{817} \cdot 103^{44}) &= (13,9,817,44), \\ e(1) &= \lambda, && \text{(the empty string)} \\ e(0) &\text{ is undefined.} \end{aligned}$$

Now e is a function from $\mathbb{N}$ to $(\mathbb{Z}^+)^*$. It is defined on all positive integers, and it clearly is a surjection. This shows that

$$\mathbb{N} \text{ surj } (\mathbb{Z}^+)^*,$$

and the countability of the finite strings of positive integers now follows by Corollary 7.1.14.(b).

---

[1] Some rational numbers can be expanded in two ways—as an infinite sequence ending in all 0's or as an infinite sequence ending in all 9's. For example,

$$5 = 5.000\cdots = 4.999\ldots,$$
$$\frac{1}{10} = 0.1000\cdots = 0.0999\ldots.$$

In such cases, define $b(r)$ to be the sequence that ends with all 0's.

Larger Infinities

There are lots of different sizes of infinite sets. For example, starting with the infinite set, $\mathbb{N}$, of nonnegative integers, we can build the infinite sequence of sets

$$\mathbb{N} \text{ strict } \text{pow}(\mathbb{N}) \text{ strict } \text{pow}(\text{pow}(\mathbb{N})) \text{ strict } \text{pow}(\text{pow}(\text{pow}(\mathbb{N}))) \text{ strict } \ldots.$$

By Cantor's Theorem 7.1.11, each of these sets is strictly bigger than all the preceding ones. But that's not all: the union of all the sets in the sequence is strictly bigger than each set in the sequence (see Problem 7.23). In this way you can keep going indefinitely, building "bigger" infinities all the way.

7.1.4 Diagonal Argument

Theorem 7.1.11 and similar proofs are collectively known as "diagonal arguments" because of a more intuitive version of the proof described in terms of on an infinite square array. Namely, suppose there was a bijection between $\mathbb{N}$ and $\{0,1\}^\omega$. If such a relation existed, we would be able to display it as a list of the infinite bit strings in some countable order or another. Once we'd found a viable way to organize this list, any given string in $\{0,1\}^\omega$ would appear in a finite number of steps, just as any integer you can name will show up a finite number of steps from 0. This hypothetical list would look something like the one below, extending to infinity both vertically and horizontally:

$$\begin{array}{lcccccccc}
A_0 & = & 1 & 0 & 0 & 0 & 1 & 1 & \cdots \\
A_1 & = & 0 & 1 & 1 & 1 & 0 & 1 & \cdots \\
A_2 & = & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\
A_3 & = & 0 & 1 & 0 & 0 & 1 & 0 & \cdots \\
A_4 & = & 0 & 0 & 1 & 0 & 0 & 0 & \cdots \\
A_5 & = & 1 & 0 & 0 & 1 & 1 & 1 & \cdots \\
\vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

But now we can exhibit a sequence that's missing from our allegedly complete list of all the sequences. Look at the diagonal in our sample list:

$$\begin{array}{lcccccccc}
A_0 & = & \mathbf{1} & 0 & 0 & 0 & 1 & 1 & \cdots \\
A_1 & = & 0 & \mathbf{1} & 1 & 1 & 0 & 1 & \cdots \\
A_2 & = & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & \cdots \\
A_3 & = & 0 & 1 & 0 & \mathbf{0} & 1 & 0 & \cdots \\
A_4 & = & 0 & 0 & 1 & 0 & \mathbf{0} & 0 & \cdots \\
A_5 & = & 1 & 0 & 0 & 1 & 1 & \mathbf{1} & \cdots \\
\vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

Here is why the diagonal argument has its name: we can form a sequence D consisting of the bits on the diagonal.

$$D = \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad \cdots,$$

Then, we can form another sequence by switching the **1**'s and **0**'s along the diagonal. Call this sequence C:

$$C = \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad \cdots.$$

Now if nth term of A_n is **1** then the nth term of C is **0**, and *vice versa*, which guarantees that C differs from A_n. In other words, C has at least one bit different from *every* sequence on our list. So C is an element of $\{0,1\}^\omega$ that does not appear in our list—our list can't be complete!

This diagonal sequence C corresponds to the set $\{a \in A \mid a \notin g(a)\}$ in the proof of Theorem 7.1.11. Both are defined in terms of a countable subset of the uncountable infinity in a way that excludes them from that subset, thereby proving that no countable subset can be as big as the uncountable set.

7.2 The Halting Problem

Although towers of larger and larger infinite sets are at best a romantic concern for a computer scientist, the *reasoning* that leads to these conclusions plays a critical role in the theory of computation. Diagonal arguments are used to show that lots of problems can't be solved by computation, and there is no getting around it.

This story begins with a reminder that having procedures operate on programs is a basic part of computer science technology. For example, *compilation* refers to taking any given program text written in some "high level" programming language like Java, C++, Python, ..., and then generating a program of low-level instructions that does the same thing but is targeted to run well on available hardware. Similarly, *interpreters* or *virtual machines* are procedures that take a program text designed to be run on one kind of computer and simulate it on another kind of computer. Routine features of compilers involve "type-checking" programs to ensure that certain kinds of run-time errors won't happen, and "optimizing" the generated programs so they run faster or use less memory.

The fundamental thing that just can't be done by computation is a *perfect* job of type-checking, optimizing, or any kind of analysis of the overall run time behavior of programs. In this section, we'll illustrate this with a basic example known as the *Halting Problem*. The general Halting Problem for some programming language

is, given an arbitrary program, to determine whether the program will run forever if it is not interrupted. If the program does not run forever, it is said to halt. Real programs may halt in many ways, for example, by returning some final value, aborting with some kind of error, or by awaiting user input. But it's easy to detect when any given program will halt: just run it on a virtual machine and wait till it stops. The problem comes when the given program does *not* halt—you may wind up waiting indefinitely without realizing that the wait is fruitless. So how could you detect that the program does *not* halt? We will use a diagonal argument to prove that if an analysis program tries to recognize the non-halting programs, it is bound to give wrong answers, or no answers, for an infinite number of the programs it is supposed to be able to analyze!

To be precise about this, let's call a programming procedure—written in your favorite programming language—a *string procedure* when it is applicable to strings over a standard alphabet—say, the 256 character ASCII alphabet. As a simple example, you might think about how to write a string procedure that halts precisely when it is applied to a *double letter* ASCII string, namely, a string in which every character occurs twice in a row. For example, `aaCC33`, and `zz++ccBB` are double letter strings, but `aa;bb`, `b33`, and `AAAAA` are not.

We'll call a set of strings *recognizable* if there is a string procedure that halts when it is applied to any string in that set and does not halt when applied to any string not in the set. For example, we've just agreed that the set of double letter strings is recognizable.

Let ASCII* be the set of (finite) strings of ASCII characters. There is no harm in assuming that every program can be written using only the ASCII characters; they usually are. When a string $s \in \text{ASCII}^*$ is actually the ASCII description of some string procedure, we'll refer to that string procedure as P_s. You can think of P_s as the result of compiling s.[2] It's technically helpful to treat *every* ASCII string as a program for a string procedure. So when a string $s \in \text{ASCII}^*$ doesn't parse as a proper string procedure, we'll define P_s to be some default string procedure—say one that never halts on any input.

Focusing just on string procedures, the general Halting Problem is to decide, given strings s and t, whether or not the procedure P_s halts when applied to t. We'll show that the general problem can't be solved by showing that a special case can't be solved, namely, whether or not P_s applied to s halts. So, let's define

[2]The string, $s \in \text{ASCII}^*$, and the procedure, P_s, have to be distinguished to avoid a type error: you can't apply a string to string. For example, let s be the string that you wrote as your program to recognize the double letter strings. Applying s to a string argument, say `aabbccdd`, should throw a type exception; what you need to do is compile s to the procedure P_s and then apply P_s to `aabbccdd`.

Definition 7.2.1.

$$\text{No-halt} ::= \{s \in \text{ASCII}^* \mid P_s \text{ applied to } s \text{ does not halt}\}. \tag{7.3}$$

We're going to prove

Theorem 7.2.2. *No-halt is not recognizable.*

We'll use an argument just like Cantor's in the proof of Theorem 7.1.11.

Proof. For any string $s \in \text{ASCII}^*$, let $f(s)$ be the set of strings recognized by P_s:

$$f(s) ::= \{t \in \text{ASCII}^* \mid P_s \text{ halts when applied to } t\}.$$

By convention, we associated a string procedure, P_s, with every string, $s \in \text{ASCII}^*$, which makes f a total function, and by definition,

$$s \in \text{No-halt IFF } s \notin f(s), \tag{7.4}$$

for all strings, $s \in \text{ASCII}^*$.

Now suppose to the contrary that No-halt was recognizable. This means there is some procedure P_{s_0} that recognizes No-halt, which is the same as saying that

$$\text{No-halt} = f(s_0).$$

Combined with (7.4), we get

$$s \in f(s_0) \quad \text{iff} \quad s \notin f(s) \tag{7.5}$$

for all $s \in \text{ASCII}^*$. Now letting $s = s_0$ in (7.5) yields the immediate contradiction

$$s_0 \in f(s_0) \quad \text{iff} \quad s_0 \notin f(s_0).$$

This contradiction implies that No-halt cannot be recognized by any string procedure. ■

So that does it: it's logically impossible for programs in any particular language to solve just this special case of the general Halting Problem for programs in that language. And having proved that it's impossible to have a procedure that figures out whether an arbitrary program halts, it's easy to show that it's impossible to have a procedure that is a perfect recognizer for *any* overall run time property.[3]

[3]The weasel word "overall" creeps in here to rule out some run time properties that are easy to recognize because they depend only on part of the run time behavior. For example, the set of programs that halt after executing at most 100 instructions is recognizable.

For example, most compilers do "static" type-checking at compile time to ensure that programs won't make run-time type errors. A program that type-checks is guaranteed not to cause a run-time type-error. But since it's impossible to recognize perfectly when programs won't cause type-errors, it follows that the type-checker must be rejecting programs that really wouldn't cause a type-error. The conclusion is that no type-checker is perfect—you can always do better!

It's a different story if we think about the *practical* possibility of writing programming analyzers. The fact that it's logically impossible to analyze perfectly arbitrary programs does not mean that you can't do a very good job analyzing interesting programs that come up in practice. In fact, these "interesting" programs are commonly *intended* to be analyzable in order to confirm that they do what they're supposed to do.

In the end, it's not clear how much of a hurdle this theoretical limitation implies in practice. But the theory does provide some perspective on claims about general analysis methods for programs. The theory tells us that people who make such claims either

- are exaggerating the power (if any) of their methods, perhaps to make a sale or get a grant, or
- are trying to keep things simple by not going into technical limitations they're aware of, or
- perhaps most commonly, are so excited about some useful practical successes of their methods that they haven't bothered to think about the limitations which must be there.

So from now on, if you hear people making claims about having general program analysis/verification/optimization methods, you'll know they can't be telling the whole story.

One more important point: there's no hope of getting around this by switching programming languages. Our proof covered programs written in some given programming language like Java, for example, and concluded that no Java program can perfectly analyze all Java programs. Could there be a C++ analysis procedure that successfully takes on all Java programs? After all, C++ does allow more intimate manipulation of computer memory than Java does. But there is no loophole here: it's possible to write a virtual machine for C++ in Java, so if there were a C++ procedure that analyzed Java programs, the Java virtual machine would be able to do it too, and that's impossible. These logical limitations on the power of computation apply no matter what kinds of programs or computers you use.

7.3 The Logic of Sets

7.3.1 Russell's Paradox

Reasoning naively about sets turns out to be risky. In fact, one of the earliest attempts to come up with precise axioms for sets in the late nineteenth century by the logician Gotlob Frege, was shot down by a three line argument known as *Russell's Paradox*[4] which reasons in nearly the same way as the proof of Cantor's Theorem 7.1.11. This was an astonishing blow to efforts to provide an axiomatic foundation for mathematics:

Russell's Paradox

Let S be a variable ranging over all sets, and define

$$W ::= \{S \mid S \notin S\}.$$

So by definition,

$$S \in W \text{ iff } S \notin S,$$

for every set S. In particular, we can let S be W, and obtain the contradictory result that

$$W \in W \text{ iff } W \notin W.$$

The simplest reasoning about sets crashes mathematics! Russell and his colleague Whitehead spent years trying to develop a set theory that was not contradictory, but would still do the job of serving as a solid logical foundation for all of mathematics.

Actually, a way out of the paradox was clear to Russell and others at the time: *it's unjustified to assume that W is a set*. The step in the proof where we let S be W has no justification, because S ranges over sets, and W might not be a set. In fact, the paradox implies that W had better not be a set!

[4] Bertrand Russell was a mathematician/logician at Cambridge University at the turn of the Twentieth Century. He reported that when he felt too old to do mathematics, he began to study and write about philosophy, and when he was no longer smart enough to do philosophy, he began writing about politics. He was jailed as a conscientious objector during World War I. For his extensive philosophical and political writing, he won a Nobel Prize for Literature.

But denying that W is a set means we must *reject* the very natural axiom that every mathematically well-defined collection of sets is actually a set. The problem faced by Frege, Russell and their fellow logicians was how to specify *which* well-defined collections are sets. Russell and his Cambridge University colleague Whitehead immediately went to work on this problem. They spent a dozen years developing a huge new axiom system in an even huger monograph called *Principia Mathematica*, but for all intents and purposes, their approach failed. It was so cumbersome no one ever used it, and it was subsumed by a much simpler, and now widely accepted, axiomatization of set theory by the logicians Zermelo and Fraenkel.

7.3.2 The ZFC Axioms for Sets

A *formula of set theory*[5] is a predicate formula that only uses the predicates "$x = y$" and "$x \in y$." The domain of discourse is the collection of sets, and "$x \in y$" is interpreted to mean that x and y are variables that range over sets, and x is one of the elements in y.

It's generally agreed that, using some simple logical deduction rules, essentially all of mathematics can be derived from some formulas of set theory called the Axioms of *Zermelo-Fraenkel Set Theory* with Choice (ZFC).

For example, since x is a subset of y iff every element of x is also an element of y, here's how we can express x being a subset of y with a formula of set theory:

$$(x \subseteq y) ::= \forall z.\, (z \in x \text{ IMPLIES } z \in y). \tag{7.6}$$

Now we can express formulas of set theory using "$x \subseteq y$" as an abbreviation for formula (7.6).

We're *not* going to be studying the axioms of ZFC in this text, but we thought you might like to see them—and while you're at it, get some practice reading quantified formulas:

Extensionality. Two sets are equal if they have the same members.

$$(\forall z.\, z \in x \text{ IFF } z \in y) \text{ IMPLIES } x = y.$$

Pairing. For any two sets x and y, there is a set, $\{x, y\}$, with x and y as its only elements:

$$\forall x, y.\, \exists u.\, \forall z.\, [z \in u \text{ IFF } (z = x \text{ OR } z = y)]$$

[5] Technically this is called a *first-order predicate formula* of set theory

Union. The union, u, of a collection, z, of sets is also a set:

$$\forall z.\, \exists u.\, \forall x.\, (\exists y.\, x \in y \text{ AND } y \in z) \text{ IFF } x \in u.$$

Infinity. There is an infinite set. Specifically, there is a nonempty set, x, such that for any set $y \in x$, the set $\{y\}$ is also a member of x.

Subset. Given any set, x, and any definable property of sets, there is a set containing precisely those elements $y \in x$ that have the property.

$$\forall x.\, \exists z.\, \forall y.\, y \in z \text{ IFF } [y \in x \text{ AND } \phi(y)]$$

where $\phi(y)$ is any assertion about y definable in the notation of set theory.

Power Set. All the subsets of a set form another set:

$$\forall x.\, \exists p.\, \forall u.\, u \subseteq x \text{ IFF } u \in p.$$

Replacement. Suppose a formula, ϕ, of set theory defines the graph of a function, that is,

$$\forall x, y, z.\, [\phi(x, y) \text{ AND } \phi(x, z)] \text{ IMPLIES } y = z.$$

Then the image of any set, s, under that function is also a set, t. Namely,

$$\forall s\, \exists t\, \forall y.\, [\exists x.\, \phi(x, y) \text{ IFF } y \in t].$$

Foundation. There cannot be an infinite sequence

$$\cdots \in x_n \in \cdots \in x_1 \in x_0$$

of sets each of which is a member of the previous one. This is equivalent to saying every nonempty set has a "member-minimal" element. Namely, define

$$\text{member-minimal}(m, x) ::= [m \in x \text{ AND } \forall y \in x.\, y \notin m].$$

Then the foundation axiom is

$$\forall x.\, x \neq \emptyset \text{ IMPLIES } \exists m.\, \text{member-minimal}(m, x).$$

Choice. Given a set, s, whose members are nonempty sets no two of which have any element in common, then there is a set, c, consisting of exactly one element from each set in s. The formula is given in Problem 7.28.

7.3.3 Avoiding Russell's Paradox

These modern ZFC axioms for set theory are much simpler than the system Russell and Whitehead first came up with to avoid paradox. In fact, the ZFC axioms are as simple and intuitive as Frege's original axioms, with one technical addition: the Foundation axiom. Foundation captures the intuitive idea that sets must be built up from "simpler" sets in certain standard ways. And in particular, Foundation implies that no set is ever a member of itself. So the modern resolution of Russell's paradox goes as follows: since $S \notin S$ for all sets S, it follows that W, defined above, contains every set. This means W can't be a set—or it would be a member of itself.

7.4 Does All This Really Work?

So this is where mainstream mathematics stands today: there is a handful of ZFC axioms from which virtually everything else in mathematics can be logically derived. This sounds like a rosy situation, but there are several dark clouds, suggesting that the essence of truth in mathematics is not completely resolved.

- The ZFC axioms weren't etched in stone by God. Instead, they were mostly made up by Zermelo, who may have been a brilliant logician, but was also a fallible human being—probably some days he forgot his house keys. So maybe Zermelo, just like Frege, didn't get his axioms right and will be shot down by some successor to Russell who will use his axioms to prove a proposition P and its negation $\overline{P}$. Then math as we understand it would be broken—this may sound crazy, but it has happened before.

  In fact, while there is broad agreement that the ZFC axioms are capable of proving all of standard mathematics, the axioms have some further consequences that sound paradoxical. For example, the Banach-Tarski Theorem says that, as a consequence of the *axiom of choice*, a solid ball can be divided into six pieces and then the pieces can be rigidly rearranged to give *two* solid balls of the same size as the original!

- Some basic questions about the nature of sets remain unresolved. For example, Cantor raised the question whether there is a set whose size is strictly between the smallest infinite set, $\mathbb{N}$ (see Problem 7.9), and the strictly larger set, $\text{pow}(\mathbb{N})$? Cantor guessed not:

Cantor's *Contiuum Hypothesis*: There is no set, A, such that

$$\mathbb{N} \text{ strict } A \text{ strict } \text{pow}(\mathbb{N}).$$

The Continuum Hypothesis remains an open problem a century later. Its difficulty arises from one of the deepest results in modern Set Theory—discovered in part by Gödel in the 1930's and Paul Cohen in the 1960's—namely, the ZFC axioms are not sufficient to settle the Continuum Hypothesis: there are two collections of sets, each obeying the laws of ZFC, and in one collection the Continuum Hypothesis is true, and in the other it is false. Until a mathematician with a deep understanding of sets can extend ZFC with persuasive new axioms, the Continuum Hypothesis will remain undecided.

- But even if we use more or different axioms about sets, there are some unavoidable problems. In the 1930's, Gödel proved that, assuming that an axiom system like ZFC is consistent—meaning you can't prove both P and $\overline{P}$ for any proposition, P—then the very proposition that the system is consistent (which is not too hard to express as a logical formula) cannot be proved in the system. In other words, no consistent system is strong enough to verify itself.

7.4.1 Large Infinities in Computer Science

If the romance of different-size infinities and continuum hypotheses doesn't appeal to you, not knowing about them is not going to limit you as a computer scientist. These abstract issues about infinite sets rarely come up in mainstream mathematics, and they don't come up at all in computer science, where the focus is generally on "countable," and often just finite, sets. In practice, only logicians and set theorists have to worry about collections that are "too big" to be sets. That's part of the reason that the 19th century mathematical community made jokes about "Cantor's paradise" of obscure infinities. But the challenge of reasoning correctly about this far-out stuff led directly to the profound discoveries about the logical limits of computation described in Section 7.2, and that really is something every computer scientist should understand.

Problems for Section 7.1

Practice Problems

Problem 7.1.
Prove that if A and B are countable sets, then so is $A \cup B$.

Problem 7.2.
Show that the set $\{0,1\}^*$ of finite binary strings is countable.

Problem 7.3.
Describe an example of two **un**countable sets A and B such that there is no bijection between A and B.

Problem 7.4.
Prove that if there is a total injective ([≥ 1 out, ≤ 1 in]) relation from $S \to \mathbb{N}$, then S is countable.

Problem 7.5.
For each of the following sets, indicate whether it is finite, countably infinite, or uncountable.

1. The set of solutions to the equation $x^3 - x = -0.1$.
2. The set of natural numbers $\mathbb{N}$.
3. The set of rational numbers $\mathbb{Q}$.
4. The set of real numbers $\mathbb{R}$.
5. The set of integers $\mathbb{Z}$.
6. The set of complex numbers $\mathbb{C}$.
7. The set of words in the English language no more than 20 characters long.
8. The powerset of the set of all possible bijections from $\{1, 2, \ldots, 10\}$ to itself.
9. An infinite set S with the property that there exists a total surjective function $f : \mathbb{N} \to S$.
10. A set $A \cup B$ where A is countable and B is uncountable.

Problem 7.6.
Circle the correct completions (there may be more than one)
A strict $\mathbb{N}$ IFF ...

- $|A|$ is undefined.
- A is countably infinite.
- A is uncountable.
- A is finite.
- $\mathbb{N}$ surj A.
- $\forall n \in \mathbb{N}, |A| \leq n$.
- $\forall n \in \mathbb{N}, |A| \geq n$.
- $\exists n \in \mathbb{N}. |A| \leq n$.
- $\exists n \in \mathbb{N}. |A| < n$.

Problem 7.7.
Let A to be some infinite set and B to be some countable set. We know from Lemma 7.1.7 that

$$A \text{ bij } (A \cup \{b_0\})$$

for any element $b_0 \in B$. An easy induction implies that

$$A \text{ bij } (A \cup \{b_0, b_1, \ldots, b_n\}) \tag{7.7}$$

for any finite subset $\{b_0, b_1, \ldots, b_n\} \subset B$.

Students sometimes think that (7.7) shows that A bij $(A \cup B)$. Now it's true that A bij $(A \cup B)$ for all such A and B for any countable set B (Problem 7.13), but the facts above do not prove it.

To explain this, let's say that a predicate $P(C)$ is *finitely discontinuous* when $P(A \cup F)$ is true for every *finite* subset $F \subset B$, but $P(A \cup B)$ is false. The hole in the claim that (7.7) implies A bij $(A \cup B)$ is the assumption (without proof) that the predicate

$$P_0(C) ::= [A \text{ bij } C]$$

is not finitely discontinuous. This assumption about P_0 is correct, but it's not completely obvious and takes some proving.

To illustrate this point, let A be the nonnegative integers and B be the nonnegative rational numbers, and remember that both A and B are countably infinite. Some of the predicates $P(C)$ below are finitely **d**iscontinuous and some are **n**ot. Indicate which is which.

1. C is finite.
2. C is countable.
3. C is uncountable.
4. C contains only finitely many non-integers.
5. C contains the rational number 2/3.
6. There is a maximum non-integer in C.
7. There is an $\epsilon > 0$ such that any two elements of C are ϵ apart.
8. C is countable.
9. C is uncountable.
10. C has no infinite decreasing sequence $c_0 > c_1 > \cdots$.
11. Every nonempty subset of C has a minimum element.
12. C has a maximum element.
13. C has a minimum element.

Class Problems

Problem 7.8.
Show that the set $\mathbb{N}^*$ of finite sequences of nonnegative integers is countable.

Problem 7.9. **(a)** Several students felt the proof of Lemma 7.1.7 was worrisome, if not circular. What do you think?

(b) Use the proof of Lemma 7.1.7 to show that if A is an infinite set, then A surj $\mathbb{N}$, that is, every infinite set is "as big as" the set of nonnegative integers.

Problem 7.10.
The rational numbers fill the space between integers, so a first thought is that there must be more of them than the integers, but it's not true. In this problem you'll show that there are the same number of positive rationals as positive integers. That is, the positive rationals are countable.

(a) Define a bijection between the set, $\mathbb{Z}^+$, of positive integers, and the set, $(\mathbb{Z}^+ \times \mathbb{Z}^+)$, of all pairs of positive integers:

$$\begin{array}{l}(1,1),(1,2),(1,3),(1,4),(1,5),\ldots\\(2,1),(2,2),(2,3),(2,4),(2,5),\ldots\\(3,1),(3,2),(3,3),(3,4),(3,5),\ldots\\(4,1),(4,2),(4,3),(4,4),(4,5),\ldots\\(5,1),(5,2),(5,3),(5,4),(5,5),\ldots\\\quad\vdots\end{array}$$

(b) Conclude that the set, $\mathbb{Q}^+$, of all positive rational numbers is countable.

Problem 7.11.
This problem provides a proof of the [Schröder-Bernstein] Theorem:

$$\text{If } A \text{ surj } B \text{ and } B \text{ surj } A, \text{ then } A \text{ bij } B. \tag{7.8}$$

(a) It is OK to assume that A and B are disjoint. Why?

(b) Explain why there are total injective functions $f : A \to B$, and $g : B \to A$.

Picturing the diagrams for f and g, there is *exactly one* arrow *out* of each element —a left-to-right f-arrow if the element is in A and a right-to-left g-arrow if the element is in B. This is because f and g are total functions. Also, there is *at most one* arrow *into* any element, because f and g are injections.

So starting at any element, there is a unique and unending path of arrows going forwards. There is also a unique path of arrows going backwards, which might be unending, or might end at an element that has no arrow into it. These paths are completely separate: if two ran into each other, there would be two arrows into the element where they ran together.

This divides all the elements into separate paths of four kinds:

i. paths that are infinite in both directions,

ii. paths that are infinite going forwards starting from some element of A.

iii. paths that are infinite going forwards starting from some element of B.

iv. paths that are unending but finite.

(c) What do the paths of the last type (iv) look like?

(d) Show that for each type of path, either

- the f-arrows define a bijection between the A and B elements on the path, or
- the g-arrows define a bijection between B and A elements on the path, or
- both sets of arrows define bijections.

For which kinds of paths do both sets of arrows define bijections?

(e) Explain how to piece these bijections together to prove that A and B are the same size.

Problem 7.12. (a) Prove that if a nonempty set, C, is countable, then there is a *total* surjective function $f : \mathbb{N} \to C$.

(b) Conversely, suppose that $\mathbb{N}$ surj D, that is, there is a not necessarily total surjective function $f : \mathbb{N}D$. Prove that D is countable.

Homework Problems

Problem 7.13.
Prove that if A is an infinite set and B is a countably infinite set that has no elements in common with A, then

$$A \text{ bij } (A \cup B).$$

Reminder: You may assume any of the results from class, MITx, or the text as long as you state them explicitly.

Problem 7.14.
In this problem you will prove a fact that may surprise you—or make you even more convinced that set theory is nonsense: the half-open unit interval is actually the "*same size*" as the nonnegative quadrant of the real plane![6] Namely, there is a bijection from $(0, 1]$ to $[0, \infty) \times [0, \infty)$.

(a) Describe a bijection from $(0, 1]$ to $[0, \infty)$.

Hint: $1/x$ almost works.

(b) An infinite sequence of the decimal digits $\{0, 1, \ldots, 9\}$ will be called *long* if it does not end with all 0's. An equivalent way to say this is that a long sequence

[6] The half-open unit interval, $(0, 1]$, is $\{r \in \mathbb{R} \mid 0 < r \leq 1\}$. Similarly, $[0, \infty) ::= \{r \in \mathbb{R} \mid r \geq 0\}$.

is one that has infinitely many occurrences of nonzero digits. Let L be the set of all such long sequences. Describe a bijection from L to the half-open real interval $(0, 1]$.

Hint: Put a decimal point at the beginning of the sequence.

(c) Describe a surjective function from L to L^2 that involves alternating digits from two long sequences. *Hint:* The surjection need not be total.

(d) Prove the following lemma and use it to conclude that there is a bijection from L^2 to $(0, 1]^2$.

Lemma 7.4.1. *Let A and B be nonempty sets. If there is a bijection from A to B, then there is also a bijection from $A \times A$ to $B \times B$.*

(e) Conclude from the previous parts that there is a surjection from $(0, 1]$ to $(0, 1]^2$. Then appeal to the Schröder-Bernstein Theorem to show that there is actually a bijection from $(0, 1]$ to $(0, 1]^2$.

(f) Complete the proof that there is a bijection from $(0, 1]$ to $[0, \infty)^2$.

Exam Problems

Problem 7.15.
Prove that if $A_0, A_1, \ldots, A_n, \ldots$ is an infinite sequence of countable sets, then so is

$$\bigcup_{n=0}^{\infty} A_n$$

Problem 7.16.
Let A and B be countably infinite sets:

$$A = \{a_0, a_1, a_2, a_3, \ldots\}$$
$$B = \{b_0, b_1, b_2, b_3, \ldots\}$$

Show that their product, $A \times B$, is also a countable set by showing how to list the elements of $A \times B$. You need only show enough of the initial terms in your sequence to make the pattern clear—a half dozen or so terms usually suffice.

Problem 7.17. (a) Prove that if A and B are countable sets, then so is $A \cup B$.

(b) Prove that if C is a countable set and D is infinite, then there is a bijection between D and $C \cup D$.

Problem 7.18.

Let $\{0,1\}^*$ be the set of finite binary sequences, $\{0,1\}^\omega$ be the set of infinite binary sequences, and F be the set of sequences in $\{0,1\}^\omega$ that contain only a **f**inite number of occurrences of `1`'s.

(a) Describe a simple surjective function from $\{0,1\}^*$ to F.

(b) The set $\overline{F} ::= \{0,1\}^\omega - F$ consists of all the infinite binary sequences with *infinitely* many `1`'s. Use the previous problem part to prove that $\overline{F}$ is uncountable.

Hint: We know that $\{0,1\}^*$ is countable and $\{0,1\}^\omega$ is not.

Problem 7.19.

Let $\{0,1\}^\omega$ be the set of infinite binary strings, and let $B \subset \{0,1\}^\omega$ be the set of infinite binary strings containing infinitely many occurrences of 1's. Prove that B is uncountable. (We have already shown that $\{0,1\}^\omega$ is uncountable.)

Hint: Define a suitable function from $\{0,1\}^\omega$ to B.

Problem 7.20.

A real number is called *quadratic* when it is a root of a degree two polynomial with integer coefficients. Explain why there are only countably many quadratic reals.

Problem 7.21.

Describe which of the following sets have bijections between them:

$\mathbb{Z}$ (integers),	$\mathbb{R}$ (real numbers),
$\mathbb{C}$ (complex numbers),	$\mathbb{Q}$ (rational numbers),
pow($\mathbb{Z}$) (all subsets of integers),	pow($\emptyset$),
pow(pow($\emptyset$)),	$\{0,1\}^*$ (finite binary sequences),
$\{0,1\}^\omega$ (infinite binary sequences)	$\{\mathbf{T},\mathbf{F}\}$ (truth values)
pow($\{\mathbf{T},\mathbf{F}\}$),	pow($\{0,1\}^\omega$)

Problems for Section 7.2

Class Problems

Problem 7.22.
Let $\mathbb{N}^\omega$ be the set of infinite sequences of nonnegative integers. For example, some sequences of this kind are:

$$(0, 1, 2, 3, 4, \ldots),$$
$$(2, 3, 5, 7, 11, \ldots),$$
$$(3, 1, 4, 5, 9, \ldots).$$

Prove that this set of sequences is uncountable.

Problem 7.23.
There are lots of different sizes of infinite sets. For example, starting with the infinite set, $\mathbb{N}$, of nonnegative integers, we can build the infinite sequence of sets

$$\mathbb{N} \text{ strict } \text{pow}(\mathbb{N}) \text{ strict } \text{pow}(\text{pow}(\mathbb{N})) \text{ strict } \text{pow}(\text{pow}(\text{pow}(\mathbb{N}))) \text{ strict } \ldots.$$

where each set is "strictly smaller" than the next one by Theorem 7.1.11. Let $\text{pow}^n(\mathbb{N})$ be the nth set in the sequence, and

$$U ::= \bigcup_{n=0}^{\infty} \text{pow}^n(\mathbb{N}).$$

(a) Prove that

$$U \text{ surj } \text{pow}^n(\mathbb{N}), \tag{7.9}$$

for all $n > 0$.

(b) Prove that

$$\text{pow}^n(\mathbb{N}) \text{ strict } U$$

for all $n \in \mathbb{N}$.

Now of course, we could take $U, \text{pow}(U), \text{pow}(\text{pow}(U)), \ldots$ and keep on in this way building still bigger infinities indefinitely.

Problem 7.24.
The method used to prove Cantor's Theorem that the power set is "bigger" than the

set, leads to many important results in logic and computer science. In this problem we'll apply that idea to describe a set of binary strings that can't be described by ordinary logical formulas. To be provocative, we could say that we will describe an undescribable set of strings!

The following logical formula illustrates how a formula can describe a set of strings. The formula

$$\text{NOT}[\exists y.\, \exists z.\, s = y1z], \qquad \text{(no-1s}(s)\text{)}$$

where the variables range over the set, $\{0,1\}^*$, of finite binary strings, says that the binary string, s, does not contain a 1.

We'll call such a predicate formula, $G(s)$, about strings a *string formula*, and we'll use the notation strings(G) for the set of binary strings with the property described by G. That is,

$$\text{strings}(G) ::= \{s \in \{0,1\}^* \mid G(s)\}.$$

A set of binary strings is *describable* if it equals strings(G) for some string formula, G. So the set, 0^*, of finite strings of 0's is describable because it equals strings(no-1s).[7]

The idea of representing data in binary is a no-brainer for a computer scientist, so it won't be a stretch to agree that any string formula can be represented by a binary string. We'll use the notation G_x for the string formula with binary representation $x \in \{0,1\}^*$. The details of the representation don't matter, except that there ought to be a display procedure that can actually display G_x given x.

Standard binary representations of formulas are often based on character-by-character translation into binary, which means that only a sparse set of binary strings actually represent string formulas. It will be technically convenient to have *every* binary string represent some string formula. This is easy to do: tweak the display procedure so it displays some default formula, say no-1s, when it gets a binary string that isn't a standard representation of a string formula. With this tweak, *every* binary string, x, will now represent a string formula, G_x.

Now we have just the kind of situation where a Cantor-style diagonal argument can be applied, namely, we'll ask whether a string describes a property of *itself*! That may sound like a mind-bender, but all we're asking is whether $x \in \text{strings}(G_x)$.

For example, using character-by-character translations of formulas into binary, neither the string 0000 nor the string 10 would be the binary representation of a formula, so the display procedure applied to either of them would display no-1s.

[7] no-1s and similar formulas were examined in Problem 3.25, but it is not necessary to have done that problem to do this one.

That is, $G_{0000} = G_{10} =$ no-1s and so $\text{strings}(G_{0000}) = \text{strings}(G_{10}) = 0^*$. This means that

$$0000 \in \text{strings}(G_{0000}) \quad \text{and} \quad 10 \notin \text{strings}(G_{10}).$$

Now we are in a position to give a precise mathematical description of an "undescribable" set of binary strings, namely, let

Theorem. *Define*

$$U ::= \{x \in \{0,1\}^* \mid x \notin \text{strings}(G_x)\}. \tag{7.10}$$

The set U is not describable.

Use reasoning similar to Cantor's Theorem 7.1.11 to prove this Theorem.

Homework Problems

Problem 7.25.
For any sets, A, and B, let $[A \to B]$ be the set of total functions from A to B. Prove that if A is not empty and B has more than one element, then NOT(A surj $[A \to B]$).

Hint: Suppose that σ is a function from A to $[A \to B]$ mapping each element $a \in A$ to a function $\sigma_a : A \to B$. Pick any two elements of B; call them 0 and 1. Then define

$$\text{diag}(a) ::= \begin{cases} 0 \text{ if } \sigma_a(a) = 1, \\ 1 \text{ otherwise.} \end{cases}$$

Exam Problems

Problem 7.26.
Let $\{1,2,3\}^\omega$ be the set of infinite sequences containing only the numbers 1, 2, and 3. For example, some sequences of this kind are:

$$(1,1,1,1...),$$
$$(2,2,2,2...),$$
$$(3,2,1,3...).$$

Prove that $\{1,2,3\}^\omega$ is uncountable.

Hint: One approach is to define a surjective function from $\{1,2,3\}^\omega$ to the power set $\text{pow}(\mathbb{N})$.

Problems for Section 7.3

Class Problems

Problem 7.27.
Forming a pair (a, b) of items a and b is a mathematical operation that we can safely take for granted. But when we're trying to show how all of mathematics can be reduced to set theory, we need a way to represent the pair (a, b) as a set.

(a) Explain why representing (a, b) by $\{a, b\}$ won't work.

(b) Explain why representing (a, b) by $\{a, \{b\}\}$ won't work either. *Hint:* What pair does $\{\{1\}, \{2\}\}$ represent?

(c) Define

$$\text{pair}(a, b) ::= \{a, \{a, b\}\}.$$

Explain why representing (a, b) as $\text{pair}(a, b)$ uniquely determines a and b. *Hint:* Sets can't be indirect members of themselves: $a \in a$ never holds for any set a, and neither can $a \in b \in a$ hold for any b.

Problem 7.28.
The axiom of choice says that if s is a set whose members are nonempty sets that are *pairwise disjoint* —that is no two sets in s have an element in common —then there is a set, c, consisting of exactly one element from each set in s.

In formal logic, we could describe s with the formula,

$$\begin{aligned} &\text{pairwise-disjoint}(s) \\ ::= &\forall x \in s.\, x \neq \emptyset \text{ AND} \\ &\forall x, y \in s.\, x \neq y \text{ IMPLIES } x \cap y = \emptyset. \end{aligned}$$

Similarly we could describe c with the formula

$$\text{choice-set}(c, s) ::= \quad \forall x \in s.\, \exists! z.\, z \in c \cap x.$$

Here "$\exists!\, z$." is fairly standard notation for "there exists a *unique* z."

Now we can give the formal definition:

Definition (Axiom of Choice)**.**

$$\forall s.\, \text{pairwise-disjoint}(s) \text{ IMPLIES } \exists c.\, \text{choice-set}(c, s).$$

The only issue here is that set theory is technically supposed to be expressed in terms of *pure* formulas in the language of sets, which means formula that uses only the membership relation, $\in$, propositional connectives, the two quantifies $\forall$ and $\exists$, and variables ranging over all sets. Verify that the axiom of choice can be expressed as a pure formula, by explaining how to replace all impure subformulas above with equivalent pure formulas.

For example, the formula $x = y$ could be replaced with the pure formula $\forall z.\, z \in x$ IFF $z \in y$.

Problem 7.29.

Let $R : A \to A$ be a binary relation on a set, A. If $a_1 \; R \; a_0$, we'll say that a_1 is "R-smaller" than a_0. R is called *well founded* when there is no infinite "R-decreasing" sequence:

$$\cdots R\, a_n \; R \cdots R\, a_1 \; R\, a_0, \tag{7.11}$$

of elements $a_i \in A$.

For example, if $A = \mathbb{N}$ and R is the $<$-relation, then R is well founded because if you keep counting down with nonnegative integers, you eventually get stuck at zero:

$$0 < \cdots < n - 1 < n.$$

But you can keep counting up forever, so the $>$-relation is not well founded:

$$\cdots > n > \cdots > 1 > 0.$$

Also, the $\leq$-relation on $\mathbb{N}$ is not well founded because a *constant* sequence of, say, 2's, gets $\leq$-smaller forever:

$$\cdots \leq 2 \leq \cdots \leq 2 \leq 2.$$

(a) If B is a subset of A, an element $b \in B$ is defined to be *R-minimal in B* iff there is no R-smaller element in B. Prove that $R : A \to A$ is well founded iff every nonempty subset of A has an R-minimal element.

A logic *formula of set theory* has only predicates of the form "$x \in y$" for variables x, y ranging over sets, along with quantifiers and propositional operations. For example,

$$\text{isempty}(x) ::= \forall w.\ \text{NOT}(w \in x)$$

is a formula of set theory that means that "x is empty."

(b) Write a formula, member-minimal(u, v), of set theory that means that u is $\in$-minimal in v.

(c) The Foundation axiom of set theory says that $\in$ is a well founded relation on sets. Express the Foundation axiom as a formula of set theory. You may use "member-minimal" and "isempty" in your formula as abbreviations for the formulas defined above.

(d) Explain why the Foundation axiom implies that no set is a member of itself.

Homework Problems

Problem 7.30. (a) Explain how to write a formula, $\text{Subset}_n(x, y_1, y_2, \ldots, y_n)$, of set theory [8] that means $x \subseteq \{y_1, y_2, \ldots, y_n\}$.

(b) Now use the formula Subset_n to write a formula, $\text{Atmost}_n(x)$, of set theory that means that x has at most n elements.

(c) Explain how to write a formula, Exactly_n, of set theory that means that x has exactly n elements. Your formula should only be about twice the length of the formula Atmost_n.

(d) The obvious way to write a formula, $D_n(y_1, \ldots, y_n)$, of set theory that means that $y_1, \ldots, y_n$ are distinct elements is to write an AND of subformulas "$y_i \neq y_j$" for $1 \leq i < j \leq n$. Since there are $n(n-1)/2$ such subformulas, this approach leads to a formula D_n whose length grows proportional to n^2. Describe how to write such a formula $D_n(y_1, \ldots, y_n)$ whose length only grows proportional to n.

Hint: Use Subset_n and Exactly_n.

Exam Problems

Problem 7.31. (a) Explain how to write a formula $\text{Members}(p, a, b)$ of set theory[9] that means $p = \{a, b\}$.

Hint: Say that everything in p is either a or b. It's OK to use subformulas of the form "$x = y$," since we can regard "$x = y$" as an abbreviation for a genuine set theory formula.

A *pair* (a, b) is simply a sequence of length two whose first item is a and whose second is b. Sequences are a basic mathematical data type we take for granted, but when we're trying to show how all of mathematics can be reduced to set theory, we need a way to represent the ordered pair (a, b) as a set. One way that will work[10]

[8] See Section 7.3.2.

[9] See Section 7.3.2.

[10] Some similar ways that don't work are described in problem 7.27.

is to represent (a, b) as

$$\text{pair}(a, b) ::= \{a, \{a, b\}\}.$$

(b) Explain how to write a formula $\text{Pair}(p, a, b)$, of set theory [11] that means $p = \text{pair}(a, b)$.

Hint: Now it's OK to use subformulas of the form "$\text{Members}(p, a, b)$."

(c) Explain how to write a formula $\text{Second}(p, b)$, of set theory that means p is a pair whose second item is b.

Problems for Section 7.4

Homework Problems

Problem 7.32.
For any set x, define $\text{next}(x)$ to be the set consisting of all the elements of x, along with x itself:

$$\text{next}(x) ::= x \cup \{x\}.$$

So by definition,

$$x \in \text{next}(x) \text{ and } x \subset \text{next}(x). \tag{7.12}$$

Now we give a recursive definition of a collection, Ord, of sets called *ordinals* that provide a way to count infinite sets. Namely,

Definition.

$$\begin{aligned} &\emptyset \in \text{Ord}, \\ &\text{if } \nu \in \text{Ord}, \text{ then } \text{next}(\nu) \in \text{Ord}, \\ &\text{if } S \subset \text{Ord}, \text{ then } \bigcup_{\nu \in S} \nu \in \text{Ord}. \end{aligned}$$

There is a method for proving things about ordinals that follows directly from the way they are defined. Namely, let $P(x)$ be some property of sets. The *Ordinal Induction Rule* says that to prove that $P(\nu)$ is true for all ordinals ν, you need only show two things

- If P holds for all the members of $\text{next}(x)$, then it holds for $\text{next}(x)$, and
- if P holds for all members of some set S, then it holds for their union.

[11] See Section 7.3.2.

That is:

Rule. ***Ordinal Induction***

$$\frac{\begin{array}{c}\forall x.\ (\forall y \in next(x).\ P(y)) \ \text{IMPLIES}\ \ P(next(x)),\\ \forall S.\ (\forall x \in S.\ P(x)) \ \text{IMPLIES}\ \ P(\bigcup_{x\in S} x)\end{array}}{\forall \nu \in Ord.\ P(\nu)}$$

The intuitive justification for the Ordinal Induction Rule is similar to the justification for strong induction. We will accept the soundness of the Ordinal Induction Rule as a basic axiom.

(a) A set x is *closed under membership* if every element of x is also a subset of x, that is

$$\forall y \in x.\ y \subset x.$$

Prove that every ordinal ν is closed under membership.

(b) A sequence

$$\cdots \in \nu_{n+1} \in \nu_n \in \cdots \in \nu_1 \in \nu_0 \tag{7.13}$$

of ordinals ν_i is called a *member-decreasing* sequence starting at ν_0. Use Ordinal Induction to prove that no ordinal starts an infinite member-decreasing sequence.[12]

[12] Do not assume the Foundation Axiom of ZFC (Section 7.3.2) which says that there isn't *any* set that starts an infinite member-decreasing sequence. Even in versions of set theory in which the Foundation Axiom does not hold, there cannot be any infinite member-decreasing sequence of ordinals.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015