Introduction

This handout presents the “proof systems” that you are asked to connect to recursive functions in your homework. In this handout, we present only proofs about natural numbers. For another view of the same ideas, but focused on proofs about computation, consult Sections 1.4.1 and 1.6 of *Programming Languages: Build, Prove, and Compare* (pages 19 and 58).

Formal judgment and sequents

A regular person says something like “7 is a natural number.” A semanticist\(^1\) also says “7 is a natural number,” but when they write it down, they write something like this:

\[ \vdash 7 \text{ nat}. \]

Roughly speaking, what they mean is, “even without having to make any assumptions, I claim that 7 is a natural number.” The notation is an example of a *sequent* from mathematical logic, and the general form is like this:

\[ \text{context} \vdash \text{statement}. \]

A *context* usually contains assumptions, and because 7 is a natural number regardless of assumptions, the sequent “\( \vdash 7 \text{ nat} \)” doesn’t need any assumptions.

A sequent is just one form of *formal judgment*, which is how a semanticist states a claim precisely. Formal judgments play a major role in the second homework (and in programming languages more generally), and sequents are used to express judgments in type systems, which we study in mid-semester.

When we speak a sequent out loud, we don’t usually pronounce the \( \vdash \) symbol, but when we need to talk about the symbol, we call it “the turnstile.”

Proofs and inference rules

A sequent is just a claim, and as in real life, we can make claims that aren’t true, like “the square root of 2 is a natural number.” We’d like to distinguish good claims from bad ones, but in programming languages, truth is usually impossible to establish. Instead, we focus on claims that can be *proved*. To *prove* a judgment, as opposed to just stating it, we use *proof system*. If the proof system is designed properly, all provable claims are true, and therefore, no false claims are provable. For example, I can write “\( \vdash \sqrt{2} \text{ nat} \),” but I’d better not be able to prove it. (Thanks to Mr. Gödel and Mr. Turing, not all true claims are provable.)

The proof systems we use are composed of *inference rules*. An inference rule can be identified by its long horizontal line. Below the line you will find a single judgment, the *conclusion*. Above the line you will find some number of judgments, called *premises*. The rule means “if you can prove every premise (above the line), you may apply this rule, after which you are considered to have proven the conclusion (below the line).” For example, if you can prove that \( m \) is a natural number, you can also prove that \( m + 1 \) is a natural number. The rule is

\[^1\text{Someone who studies the meanings of languages.}\]
called Successor:

\[
\text{Successor} \quad \vdash_P m \text{ nat} \\
\quad \vdash_P (m + 1) \text{ nat}
\]

The capital \( p \) dangling off the turnstile is a way to label this rule as belonging to a particular proof system—one of the five in the title of the handout.

A good way to read the Successor rule is, “if you want to prove that \( m + 1 \) is a natural number, you first have to prove that \( m \) is a natural number.” This reading is good because it’s like writing a recursive function: if you want to compute a function of \( m + 1 \), you might first recursively call the function on \( m \).\(^2\) This trick is pretty good, and it covers every natural number except zero (the only one that can’t be written in the form \( m + 1 \), where \( m \) is also a natural number). But zero is also a natural number, and it needs a proof rule:

\[
\text{Zero} \quad \vdash_P 0 \text{ nat}
\]

Another good way to read this rule is this “if you want to prove that 0 is a natural number, there’s nothing else you have to prove first—you’re done.” It’s a bit like writing a recursive function: when you get an argument of 0, you don’t have to make a recursive call.

Five proof systems

When you write recursive functions on integers, you have a lot of recursion patterns to choose from. Most of these patterns correspond to some sort of proof system. Here are five example proof systems. All except the last are based on numbering systems; the last is based on parity (even versus odd).

\(^2\)Perhaps you are more accustomed to think “if I want to compute a function of \( n \), I might first recursively call the function on \( n - 1 \).” I like this thinking, but I wouldn’t want to write the Successor rule this way. When writing a specification like this, we use \( m \) and \( m + 1 \) because then the rule works for any natural number \( m \)—the rule is not restricted to, say, natural numbers greater than zero.

Peano numerals

The simplest and most standard way to characterize the natural numbers is the system named for mathematician Giuseppe Peano: a natural number is either zero or is the successor of some other natural number. You may have studied this idea in math class. Here is the proof system, identified with a subscript \( p \) on the turnstile. The rules are the two rules you’ve already seen, but under different names:

\[
\text{PeanoZero} \quad \vdash_P 0 \text{ nat}
\]

\[
\text{Successor} \quad \vdash_P m \text{ nat} \\
\quad \vdash_P (m + 1) \text{ nat}
\]

Binary “numbers”

Computer scientists say “binary number,” but a mathematician would blanch—the binary system is just another kind of numeral. A natural number is either zero or is twice a natural number \( m \) plus a bit \( b \).

\[
\text{BinaryZero} \quad \vdash_B 0 \text{ nat}
\]

\[
\text{BinaryNat} \quad \vdash_B m \text{ nat} \\
\quad \vdash_B b \text{ bit} \\
\quad \vdash_B (2 \times m + b) \text{ nat}
\]

A bit is either zero or one:

\[
\text{BitZero} \quad \vdash 0 \text{ bit}
\]

\[
\text{BitOne} \quad \vdash 1 \text{ bit}
\]

Bits are bits regardless of proof system, so I don’t decorate the turnstile.

The decimal system

The decimal (also called Arabic) numerals have a proof system very similar to “binary numbers.” A natural number is either zero or is ten times a natural number \( m \) plus a decimal digit \( d \).

\[
\text{DecimalZero} \quad \vdash_D 0 \text{ nat}
\]

\[
\text{DecimalNat} \quad \vdash_D m \text{ nat} \\
\quad \vdash_D d \text{ digit} \\
\quad \vdash_D (10 \times m + d) \text{ nat}
\]
Proving that $d$ is a digit requires ten highly repetitive rules:

| Digit0 | ⊢ $d$ digit |
| Digit1 | ⊢ $1$ digit |
| Digit2 | ⊢ $2$ digit |
| Digit3 | ⊢ $3$ digit |
| Digit4 | ⊢ $4$ digit |
| Digit5 | ⊢ $5$ digit |
| Digit6 | ⊢ $6$ digit |
| Digit7 | ⊢ $7$ digit |
| Digit8 | ⊢ $8$ digit |
| Digit9 | ⊢ $9$ digit |

Another view of the decimal system

A computation inspired by the decimal system might not want to work all the way down to zero—it might want to stop at the last digit. (For you to figure out: is the “last” digit here the least significant or the most significant?)

Parity

This strange little proof system relies on numbers being even or odd:

EvenParity ⊢ $E\ 0\ n$ nat

OddParity ⊢ $E\ 1\ n$ nat

SameParity ⊢ $E\ m\ n$ nat

The insight that is expressed here is that 0 is even, 1 is odd, and no matter whether $m$ is even or odd, so is $m + 2$.

Turning proof systems into recursive functions

The point of all these proof systems is that each one gives you a pattern of recursion that you can use to design functions. For example, let’s suppose I want to design a function that sums the natural numbers from 0 to $n$. I choose the Peano proof system. According to the proof system, any natural number $n$ is either zero or is $m + 1$, where $m$ is a natural number. I can distinguish those cases by testing $n = 0$. If $n = 0$, I’m done, and otherwise $n = m + 1$, therefore $m = n - 1$, and I make my recursive call on $n - 1$:

```c
int sumto(unsigned n) { // not tested
if (n == 0) {
    return 0;
} else {
    return n + sumto(n - 1);
}
}
```

This design applies a general strategy:

1. Look at all the rules in the proof system.
2. Look at your data, and find out which rule has that data in its conclusion.
3. If the rule has no premises, you have a base case.
4. Otherwise, for each premise with the same form of judgment, you have a recursive call.

You’ll apply this strategy again and again.

Here’s an example with the parity system, in which a natural number $n$ is either 0, or 1, or $m + 2$:

```c
typedef enum { EVEN, ODD } Parity;
Parity parity (unsigned n) { // not tested
if (n == 0) {
    return EVEN;
} else if (n == 1) {
    return ODD;
} else {
    return parity(n - 2);
}
}
```
Finally, here’s a sketch for you to apply yourself. You’re using the Decimal2 system with input $n$. If $n < 10$, then rule Decimal2Digit applies. If $n \geq 10$, then rule Decimal2Nat applies. In rule Decimal2Digit, even though there is a premise above the line, it’s not a nat judgment, so there’s no recursive call: $n < 10$ is a base case. In rule Decimal2Nat, there is $\vdash D_2 m \text{nat}$ above the line, so you do need to make a recursive call, and you’ll need to compute $m = n/10$ and $d = m \text{ mod } 10$. 