1 Introduction

This handout presents the technical basis for the key steps of a software-design process that you will use throughout the term: writing algebraic laws and turning them into code. The key steps build on concepts of “proof systems,” data, “algebraic laws,” and functions. This handout explores only proofs, laws, and functions that compute with 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).

2 Formal judgment and sequents

A regular person says something like “7 is a natural number.” A semanticist 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.”

3 Proofs and inference rules

A sequent is just a claim. As in real life, some claims are good, like “7 is a natural number,” but some claims are bad, like “the square root of 2 is a natural number.” We’d like to distinguish good claims from bad ones. Truth is always good, but truth is usually impossible to establish. Instead, we focus on provability. To prove a judgment, as opposed to just stating it, we use a 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. (Not all true claims are provable; if you have heard of “Gödel’s incompleteness theorem,” it constructs a claim that is true but not 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 called Successor:

\[
\text{Successor} \quad \vdash p \, m \text{ nat} \\
\vdash p \, (m + 1) \text{ nat}
\]

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

Footnotes:

1 Someone who studies the meanings of languages.

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 specifica-
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:

\[
\frac{}{\vdash \text{P0 nat}} \quad \frac{}{\vdash \text{P} (m + 1) \text{ nat}}
\]

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.

### 4 Five proof systems

When you write a recursive function on natural numbers, you have a lot of ways to structure the recursion. Ideally, the recursive structure of your function follows from the inductive structure you use to describe the input data. And an inductive structure of natural numbers can be described by a 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).

#### 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:

\[
\frac{}{\vdash \text{P0 nat}} \\
\frac{}{\vdash \text{P} (m + 1) \text{ nat}}
\]

#### 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 \( b \).

\[
\frac{}{\vdash \text{B0 nat}} \\
\frac{}{\vdash \text{B} (2 \times m + b) \text{ nat}}
\]

A bit is either zero or one:

\[
\frac{}{\vdash 0 \text{ bit}} \\
\frac{}{\vdash 1 \text{ bit}}
\]

Bits are bits regardless of proof system, so when I write \( \vdash 0 \text{ bit} \) or \( \vdash 1 \text{ bit} \), I don’t decorate the turnstile.

#### A decimal system for arithmetic

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 \( d \).

\[
\frac{}{\vdash \text{D0 nat}} \\
\frac{}{\vdash \text{D} (10 \times m + d) \text{ nat}}
\]

Proving that \( d \) is a digit requires ten highly repetitive rules:

\[
\frac{}{\vdash 0 \text{ digit}} \\
\frac{}{\vdash 1 \text{ digit}} \\
\frac{}{\vdash 2 \text{ digit}} \\
\frac{}{\vdash 3 \text{ digit}} \\
\frac{}{\vdash 4 \text{ digit}} \\
\frac{}{\vdash 5 \text{ digit}} \\
\frac{}{\vdash 6 \text{ digit}} \\
\frac{}{\vdash 7 \text{ digit}} \\
\frac{}{\vdash 8 \text{ digit}} \\
\frac{}{\vdash 9 \text{ digit}}
\]

This proof system is great for guiding implementations of arithmetic on natural numbers, including addition, subtraction, multiplication, and division.

#### A decimal system for numerals

The \text{DECIMAL} proof system is useful for arithmetic, but it is not at all good for looking at properties of numerals. For example, if you want to know if a number \( n \) is represented by a numeral that is all 4’s, you should avoid the \text{DECIMAL} system.\(^3\) Instead, you

\(^3\)Using the \text{DECIMAL} system has the same effect as starting every numeral with a leading zero.
<table>
<thead>
<tr>
<th>Proof system</th>
<th>Left-hand side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peano</td>
<td>(f 0) = · · ·</td>
</tr>
<tr>
<td></td>
<td>(f (m + 1)) = · · ·</td>
</tr>
<tr>
<td>Binary</td>
<td>(f 0) = · · ·</td>
</tr>
<tr>
<td></td>
<td>(f (2 × m + b)) = · · ·</td>
</tr>
<tr>
<td>Decimal (arithmetic)</td>
<td>(f (10 × m + d)) = · · ·</td>
</tr>
<tr>
<td>Decimal (numeral)</td>
<td>(f d) = · · ·</td>
</tr>
<tr>
<td></td>
<td>(f (10 × m + d)) = · · ·</td>
</tr>
<tr>
<td>Parity</td>
<td>(f 0) = · · ·</td>
</tr>
<tr>
<td></td>
<td>(f 1) = · · ·</td>
</tr>
<tr>
<td></td>
<td>(f (m + 2)) = · · ·</td>
</tr>
</tbody>
</table>

Table 1: Forms of laws for a one-argument function f

should prefer this DecNumeral system:

\[
\text{DecNumeralDigit} \quad \frac{\vdash d \text{ digit}}{\vdash_{\text{DN}} d \text{ nat}} \\
\text{DecNumeralNat} \quad \frac{m \neq 0 \vdash d \text{ digit}}{\vdash_{\text{DN}} (10 \times m + d) \text{ nat}}
\]

Parity

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

\[
\text{EvenParity} \quad \frac{}{\vdash_E 0 \text{ nat}} \quad \text{OddParity} \quad \frac{}{\vdash_E 1 \text{ nat}} \\
\text{SameParity} \quad \frac{\vdash_E m \text{ nat}}{\vdash_E (m + 2) \text{ nat}}
\]

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.

5 From proof system to algebraic specification

A proof system is a starting point for designing recursive functions. Design begins by looking at the forms of natural numbers as they appear in the conclusions of the rules. For example, the Peano system has natural numbers of the forms “0” and “(m + 1).” The binary system has natural numbers of the forms “0” and “(2 × m + b).”

Once you know the forms, the next step in designing a function is to specify what the function is supposed to do for each form. The specification is written as a set of equations called algebraic laws. These laws are stylized: there is typically one law for each form of each input, and the left-hand side of the law applies the function to those forms.

Writing left-hand sides is mechanistic: a left-hand side is determined by the name of the function being defined, the number of arguments it expects, and the form of each argument. As examples, forms of laws for all one-argument natural-number functions appear in Table 1.

The laws in Table 1 are missing their right-hand sides. Right-hand sides require thought: a right-hand side specifies what a function must do in one particular case. When writing a right-hand side, you can get a valuable hint from the underlying proof system:

1. Look at the proof rule whose conclusion has the form of input used on the law’s left-hand side.

2. If the rule has no premises, you have a base case. The right-hand side should not make any recursive calls.

3. If the rule has premises, each premise with the same form of judgment represents a potential recursive call. A premise with a different form of judgment may represent a call to a helper function.

If there is more than one input, you may have to consider more than one proof rule. Some examples appear below.

Example: is a number even?

To design a function even?, which tells if a natural number is even, we can reasonably use the Peano, binary, or parity system. The Peano system has two rules, PeanoZero and Successor. In the zero case, there is no premise above the line, and I ought to be able to tell whether zero is even without making a recursive call. In the successor case, there is a judgment \( \vdash_{\text{P}} m \text{ nat} \) above the line, and I should consider a recursive call (even? m). With these considerations in mind, I propose these laws:

\[
\text{(even? 0) = true} \\
\text{(even? (m + 1)) = ¬(even? m)}
\]

where \( ¬ \) is the “logical not” operator.

Using the binary-numeral system, I’m pedantic enough to want a helper function even-bit?, which is based on the proof system for the judgment form

\[
\text{Parity} \quad \frac{}{\vdash_E 0 \text{ nat}} \quad \text{OddParity} \quad \frac{}{\vdash_E 1 \text{ nat}} \\
\text{SameParity} \quad \frac{\vdash_E m \text{ nat}}{\vdash_E (m + 2) \text{ nat}}
\]
\(\vdash \text{b bit}\):

\[
\begin{align*}
\text{even? } 0 &= \text{true} \\
\text{even? } (2 \times m + b) &= \text{even-bit? } b \\
\text{even-bit? } 0 &= \text{true} \\
\text{even-bit? } 1 &= \text{false} \\
\text{even? } (m + 2) &= \text{even? } m
\end{align*}
\]

This is the system you would be using if you were coding even? “in the normal way.” To see why, answer this question: if \(n = 2 \times m + b\), how do you get \(b\) from \(n\)?

Here are the laws for even? using the parity system:

\[
\begin{align*}
\text{even? } 0 &= \text{true} \\
\text{even? } 1 &= \text{false} \\
\text{even? } (m + 2) &= \text{even? } m
\end{align*}
\]

I wouldn’t want to implement even? using either of the decimal proof systems. These systems don’t really fit a computation of even?, and no sane person would try to use them—there’s no point in coding even? on a digit \(d\) when you could more easily test \(n\) directly.

Example: Multiplication

The decimal proof systems are useless for parity, but for problems like multiplication, they work well. Since multiplication is a two-argument function, here is the general form of laws for a two-argument function \(g\), using the forms of input from the decimal-arithmetic proof system:

\[
\begin{align*}
g(0 \; 0) &= \ldots \\
g(10 \times m + d \; 0) &= \ldots \\
g(0 \; (10 \times m' + d')) &= \ldots \\
g(10 \times m + d \; (10 \times m' + d')) &= \ldots
\end{align*}
\]

Table 2: Identifying the form of \(n\) and extracting its parts

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Proof system} & \text{Form of } n & \text{Test for form} & \text{Parts } n \text{ is formed from} \\
\hline
\text{Peano} & 0 & n = 0 & m = n - 1 \\
& (m + 1) & n \neq 0 & m = n - 1 \\
\hline
\text{Binary} & 0 & n = 0 & m = n \div 2 \quad b = n \mod 2 \\
& (2 \times m + b) & n \neq 0 & m = n \div 2 \quad b = n \mod 2 \\
\hline
\text{Decimal (arithmetic)} & 0 & n = 0 & m = n \div 10 \quad b = n \mod 10 \\
& (10 \times m + b) & n \neq 0 & m = n \div 10 \quad b = n \mod 10 \\
\hline
\text{Decimal (numerals)} & d & n < 10 & d = n \\
& (10 \times m + b) & n \geq 10 & m = n \div 10 \quad b = n \mod 10 \\
\hline
\text{Parity} & 0 & n = 0 & m = n - 2 \\
& 1 & n = 1 & m = n - 2 \\
& (m + 2) & n \neq 0 \land n \neq 1 & m = n - 2 \\
\hline
\end{array}
\]

Example: “All fours”

Suppose I want to write a function all-fours?, which tells me if an argument’s decimal representation uses only the digit 4. That is, it likes 4, 44, 444, and so on. I don’t want the DECIMAL system, since 0 is the wrong base case. Instead, I want DECIMALNAT:

\[
\begin{align*}
\text{all-fours? } d &= (d = 4) \\
\text{all-fours? } (10 \times m + d) &= \text{all-fours? } m \land d = 4,
\end{align*}
\]

where \(\land\) is the “logical and” symbol.

6 From algebraic laws to recursive function

When the algebraic laws are complete, we write the code. Conceptually, the code emerges in response to three questions:

1. We ask of each input, how were you formed?
2. Once we know the form of an input, we proceed to ask from what parts were you formed?
3. Finally, when we know how each input was formed and from what parts, we can ask which algebraic law applies and what does the law say we are supposed to do with the parts?
The first two questions can be answered using the tests and equations shown in Table 2 on page 4. The third question is answered by selecting the algebraic law whose left-hand side has the right form, then using the right-hand side.

**Detailed example**

As a first example, let’s implement function *is_even*, in C, using the parity system. Here are the laws:

\[
\begin{align*}
(is \ even \ 0) &= true \\
(is \ even \ 1) &= false \\
(is \ even \ (m + 2)) &= (is \ even \ m)
\end{align*}
\]

Table 2 on page 4 shows that we can distinguish the forms of \(n\) using tests for 0 and 1, so the first draft of our function uses *if* statements to distinguish three cases: one for each law.

```c
bool is_even (unsigned n) {
    if (n == 0) {
        ...
    } else if (n == 1) {
        ...
    } else {
        ...
    }
}
```

In the first two cases, \(n\) isn’t formed from any other parts, and we can knock off the cases by filling in the right-hand sides of the laws:

```c
bool is_even (unsigned n) {
    if (n == 0) {
        return true;
    } else if (n == 1) {
        return false;
    } else {
        unsigned m = n - 2;
        ...
    }
}
```

In the final case, the law mentions \(m\), which is computed as \(n - 2\):

```c
bool is_even (unsigned n) {
    if (n == 0) {
        return true;
    } else if (n == 1) {
        return false;
    } else {
       (unsigned m = n - 2;
        ...
    }
}
```

With \(m\) computed, we use the right-hand side of the law to write a recursive call:

```c
bool is_even (unsigned n) {
    if (n == 0) {
        return true;
    } else if (n == 1) {
        return false;
    } else {
        unsigned m = n - 2;
        return is_even(m);
    }
}
```

In practice, I probably would not bother with \(m\) in the third case, writing instead *is_even(n-2)*.

**Condensed example**

As another example, suppose I want to design a function that sums the natural numbers from 0 to \(n\). I choose the Peano proof system, and I write these laws:

\[
\begin{align*}
(sumto \ 0) &= 0 \\
(sumto \ (m + 1)) &= (sumto \ m) + (m + 1)
\end{align*}
\]

I distinguish case \(n = 0\) from case \(n = (m + 1)\) by testing \(n = 0\), and when \(n = (m + 1)\), I get the “part” \(m\) by computing \(m = n - 1\):

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

In this code, I don’t bother with an explicit \(m\).

### 7 Complete process examples

The preceding sections of this handout look at proof systems, forms of data, and algebraic laws, which are the technical core of our recommended software process. This section works two more examples, showing all 9 steps of the complete process.

**Design of a factorial function**

1. **Understand the forms of data.** To design a factorial function, I choose the Peano system, with forms 0 and \((m + 1)\). Choosing the right system is not always obvious, but I’ve written factorial functions before, and I know the Peano system will work.
Design of a power function

Same drill, but now I define a function of two arguments, \( x \) and \( n \), to compute \( x^n \). And I do something sophisticated: I know that this computation depends only on the form of \( n \), not on the form of \( x \). So have only one form to analyze, and I get by with just two cases instead of four or more.

1. **Understand the forms of data.** Again, I choose the Peano system, with forms 0 and \((m + 1)\).

2. **Write a sample input for each form.** Again, I choose examples 0 and 4.

3. **Choose a name.** I choose power, a noun that describes what the function returns.

4. **Write the contract.** This contract is not trivial: we need to know which argument is the base and which is the exponent.

\[
\text{;; (power x n) returns } x \text{ raised to the } n \text{th power, where } n \text{ is a natural number}
\]

5. **Write examples.** My examples:

\[
\begin{align*}
\text{(check-expect (power 3 0) 1)} \\
\text{(check-expect (power 3 4) 81)}
\end{align*}
\]

(Unit tests are indented eight spaces.)

6. **Generalize to algebraic laws.** In math form,\n
\[
\begin{align*}
x^0 &= 1 \\
x^{(m+1)} &= x^m \times x
\end{align*}
\]

When you’re designing, math form is always legitimate and often helpful.

In code form,
(power x 0) == 1
(power x (+ m 1)) == (* (power x m) x)

7. **Code the case analysis.** It’s the same proof system, the same form of input, and therefore the same case analysis as for *factorial*:

```
(define power (x n)
  (if (= n 0)
      ... ; zero case
      ...)) ; successor case
```

8. **Finish the function.** Again, instead of *m*, I write *n - 1*:

```
(define power (x n)
  (if (= n 0)
      1
      (* (power x (- n 1)) x)))
```

9. **Revisit unit tests.** My tests cover every form of input, and there are no Booleans in play. And they pass.

Here’s the complete solution in file `power.imp`:

```
;; (power x n) returns x raised to the
;; nth power, where n is a natural number
(define power (x n)
  (if (= n 0)
      1
      (* (power x (- n 1)) x)))

(check-expect (power 3 0) 1)
(check-expect (power 3 4) 81)
```