4.6 Formal Definitions and Reasoning

One of the goals of this course is to teach formal (i.e. mathematical) reasoning about programming languages and compilers. In fact, here at CMU that is an explicit goal of the Logic and Languages constrained elective category. Today we'll learn some basic intellectual tools that we'll use to formally define aspects of a programming language or compiler, and to reason about those formal definitions. Tomorrow in recitation, you'll learn about a proof assistant tool, SASyLF, that will help you write down those definitions and proofs in a rigorous way and can automatically check them for mistakes.

You might ask about the title of this course: what's pragmatic about formalism and proof? Perhaps surprisingly, a lot! Writing little formalisms like the ones we'll learn in this class were used to figure out how to correctly add generics to Java—types like List<Int>. More recently, the WebAssembly platform, which for example allows safely and efficiently running C programs in browers, was defined entirely using a formal specification. We can be certain that if we have formalized a design and proved it correct, it won't have certain flaws–like security vulnerabilities in the case of mobile code in Java or WebAssembly. That's a very pragmatic thing for end users to care about! Based on the recent history, it's likely that future programming languages will be defined in terms of formalisms like the ones you'll be learning. More broadly, learning how to formalize definitions and do proofs is a skill that can help you think more precisely about the semantics of the languages you work in and the programs you write.

4.6.1 Formalizing Numbers and Arithmetic

EXAMPLE 4.23
Representing Numbers

Before tackling programming languages, let's look at something simpler. We'll start by defining natural numbers, i.e. the non-negative integers that we use to count with: $0, 1, 2, \ldots$. To reason about numbers, we need to represent them somehow. We will use abstract syntax to do this—defined using the same tools (abstract grammars) that are using to define the abstract syntax of programming languages.

Natural numbers can be defined inductively: a number is either zero or a successor of some other number. For example, the number one is the successor of zero, and the number two is the successor of one. We can define this syntactically as follows. Let $\mathtt{z}$ represent the number zero. And if n is a number, then $\mathtt{s}\ n$ is the successor of that number. Using an abstract grammar, this can be written as:

$$n \longrightarrow \mathtt{z} \mid \mathtt{s}\ n$$

Now we can represent the number 3 with the string "$\mathtt{s\ s\ s\ z}$." But instead of thinking about strings, we'd like to think of this as an abstract syntax tree, with the $\mathtt{s}$ elements forming the root and (single) branch, and $\mathtt{z}$ at the leaf. ■

EXAMPLE 4.24

Formalizing Addition

Now that we have formalized natural numbers, we'd like to reason about them. Most interesting properties of numbers rely on operators such as addition. Let's start by writing down some formal syntax for relating an addition expression to its result. We'll do that with a judgment of the form $n_1 + n_2 = n_3$, which means exactly what it looks like: that when you add the number n_1 to the number n_2, you get the number n_3. Let's call this judgment "sum." In the sum judgment, n_1, n_2, and n_3 are metavariables: we will replace them with actual numbers when we instantiate the judgment. For example, if n_1 is 0 (or "`z`") and n_2 is 1 (or "`s z`") then n_3 will be 1 (again, "`s z`" in our formalization). Note that we are using a convention that metavariables are named after the nonterminal representing their syntactic category: n_1 is a number (the subscript 1 distinguishes it from other numbers in the same judgment).

Of course, the judgment $n_1+n_2 = n_3$ is true if we instantiate it as $\mathtt{z}+\mathtt{s\ z} = \mathtt{s\ z}$, but it's not true if we instantiate it in other ways. For example, $\mathtt{z} + \mathtt{z} = \mathtt{s\ z}$ is not true. We can define when a judgment is true using inference rules. An inference rule is written as follows:

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{C}\ \textit{rule-name}$$

Where $P_1 \ldots P_n$ are judgments called *premises* and C is a judgment called the *conclusion*. We write the name of the rule to the right of the line. The inference rule means that if all the premises are true, then the conclusion is true. A special case of an inference rule is an *axiom*, which is a rule that has no premises. Let's write an axiom for adding zero to a number:

$$\frac{}{\mathtt{z} + n = n}\ \textit{sum-z}$$

This rule states that if you add `z` (zero) to any number n, the result is n. We name the rule *sum-z*, which helps us remember that it defines the sum judgment for the case where we are adding zero to a number.

Of course, we also need to define addition when we are adding numbers other than zero. Let's therefore define another rule:

$$\frac{n_1 + n_2 = n_3}{(\mathtt{s}\,n_1) + n_2 = \mathtt{s}\,n_3}\ \textit{sum-s}$$

We'll call this rule *sum-s*, because it's the successor case of sum. If we have established that $n_1 + n_2 = n_3$, then we know that we can add 1 to both sides, thus $(\mathtt{s}\,n_1) + n_2 = \mathtt{s}\,n_3$.

You might think that we need more rules–what if the second number is zero? But in fact these are all the rules we need to define addition for natural numbers. As we will see, we can use inductive reasoning to show other interesting properties of addition, such as that for all numbers n, $n + \mathtt{z} = n$. ■

4.6.2 Derivations and Provability

EXAMPLE 4.25

Derivation of $1 + 2 = 3$

How can we prove concrete facts like $1 + 2 = 3$ using this system? First of all, let's encode the numbers in our system. $1 + 2 = 3$ can be written as `s z` + `s s z` = `s s s z`. Now we can use inference rules to conclude what we need. We'll build a derivation tree, which has the thing we want to prove at the bottom, and applies rules to each judgment until we get to axioms at the leaves of the tree. For our example fact, the derivation will look like this:

$$\dfrac{\dfrac{}{\texttt{z} + \texttt{s s z} = \texttt{s s z}}\ \textit{sum-z}}{\texttt{s z} + \texttt{s s z} = \texttt{s s s z}}\ \textit{sum-s}$$

You can read the reasoning from the top down. We can apply the axiom *sum-z*, instantiating the number n with `s s z`, to conclude that `z` + `s s z` = `s s z`. We can then use that as a premise of the rule *sum-s*: n_1 will be `z`, n_2 will be `s s z`, and n_3 will be `s s z`. If we plug n_1, n_2, and n_3 into the conclusion of the `sum-s` rule, we get the desired result: `s z` + `s s z` = `s s s z`.

We say that a judgment J is *provable* if there exists a well-formed derivation that has J as its conclusion. Well-formed means that every step in the derivation is a valid instance of one of the inference rules in our formal system. ■

4.6.3 Mathematical Induction

Now, we'd like to prove some properties! Let's start with the property we mentioned earlier: for all n, $n + \texttt{z} = n$. This is "obviously" true in mathematics, but is it true in our formalization of addition? Let's find out!

We'll use a technique called mathematical induction to do this. Mathematical Induction is a technique for properties about natural numbers. One such property is the one above: that adding zero to any number n yields that same number, n.

In a proof by induction, we show that some property P is true in two parts. In the first part, called the *base case*, we show that the property is true for the number 0—which we can write as $P(0)$.[1]

In the second part, called the inductive case, we show that when the property is true for some number k, then it must be true for the number $k + 1$. More formally, we show that $P(k)$ implies $P(k + 1)$. Together, these show that the property is true for every natural number n. We know this must be true because for a given n we can apply the base case plus n instances of the inductive case to show that the property is true. The nice thing is that we do not have to actually construct the concrete proofs for each individual n (which is good because there are an infinite

1 Sometimes we want to prove a property for all numbers greater than 1, in which case our base case is $P(1)$. In general induction can start with any fixed number, in which case we prove the property for all numbers greater than or equal to the starting number.

number of such n's, and the concrete proofs get larger with each n). One generic proof suffices to prove the property for all numbers.

EXAMPLE 4.26

Proof that the sum of numbers $1 \ldots n$ is $\frac{n(n+1)}{2}$

To illustrate induction, let's prove a property from algebra: the sum of numbers from 1 to n, which we can write formally as $\sum_{i=1}^{n} i$, is equal to $\frac{n(n+1)}{2}$. We want to prove this property for all $n \geq 1$. In a proof by induction, we can start from either 0 or 1; since the property we want to prove is about natural numbers grater than or equal to 1, our base case will be $n = 1$.

We check the property for the base case P(1), which we can get by substituting 1 for n in the statement of the property. Here, the sum of numbers from 1 to 1, written $\sum_{i=1}^{1} i$, is just 1, so the property we need to prove is $1 = \frac{1(1+1)}{2}$. We can simplify $1 = \frac{1(1+1)}{2} = \frac{1(2)}{2} = \frac{2}{2} = 1$ and we are done with the base case.

Now for the inductive case, we assume that the property holds for some arbitrary number k, and we need to prove it for the number $k+1$. We assume P(k), which is that $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$. We need to prove P(k+1), which is that $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$. Starting with our assumption, we can add $k+1$ to both sides to get $\sum_{i=1}^{k} i + k + 1 = \frac{k(k+1)}{2} + k + 1$. on the left we merge $k+1$ into the sum to get $\sum_{i=1}^{k+1} i = \frac{k(k+1)}{2} + k + 1$. On the right we rewrite $k+1$ as the fraction $\frac{2k+2}{2}$ and combine it with the existing fraction to get $\sum_{i=1}^{k+1} i = \frac{k(k+1)+2k+2}{2}$. Now we multiply out $k(k+1)$ on the right to get $\sum_{i=1}^{k+1} i = \frac{k^2+k+2k+2}{2}$. We simplify to get $\sum_{i=1}^{k+1} i = \frac{k^2+3k+2}{2}$. Now we factor to get $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$. which is what we had to show, so we have proved the inductive case and also finished the proof. ■

For reasoning about programming languages—as well as the simpler case of addition for natural numbers—we'll use a variant of induction called structural induction. Structural induction works over some inductively defined structure: like our natural number syntax. The base cases are the base case of our syntax: for natural numbers, that's `z`. So to prove some property $P(n)$ for all n, the base case will be to show that $P(\texttt{z})$ holds. Then, for the inductive case, we show that we can prove that $P(n)$ holds if we assume $P(n')$ holds for all n' that are smaller than n. What does it mean for n' to be smaller than n? In structural induction, n' is smaller than n if n' is a substructure of n. In the case of natural numbers, if $n = \texttt{s}n'$, then n' is clearly a substructure of n, in the sense that it is a subtree of the abstract syntax tree that represents n. This also matches our intuition from mathematical intuition over natural numbers: $\texttt{s}n'$ means $n' + 1$ and so n is greater than n'. Just as in mathematical induction we reason from smaller numbers to larger ones, in structural induction we reason from smaller structures to larger structures.

Another way to look at this is that we are doing induction over trees. We complete an inductive proof by first proving base cases that cover all the possible leaves of the tree (our numbers form trees with a single linear branch, so there is only one leaf, i.e. `z`) and then proving inductive cases that apply to interior notes (in the case of numbers, this is the case where we assume the property for a subtree n and prove it for one node up the tree, $\texttt{s}n$). The inductive case moves the proof up the tree one step at a time, until we've proved the property for a whole number such as `s s s z`. The nice thing is, we can write a generic case of the proof for $\mathtt{s}n$, without knowing exactly what n is, and then apply that case multiple times. Thus we can prove a property of `s s s z` with only two cases (one for `z` and one for `s`), rather than four—the case for `s` can conceptually be "applied" three times to work up first to `s z`, then `s s z`, and finally `s s s z`.

Later, when we prove properties of expressions that have numbers, variables, addition, and multiplication, we can work in a similar way: proving base cases for numbers and variables and inductive cases for addition (which adds two subexpressions) and multiplication (which multiplies two subexpressions) we have proved a property for expressions of arbitrary size that are build out of these parts.

EXAMPLE 4.27

Proof that $n + \mathtt{z} = \mathtt{z}$

Let's take the simple case of zero/successor numbers first, and prove the property that for all n, $n + \mathtt{z} = n$. We can give this property a name, *sum-z-rh*, for it is a property of the sum judgment when you add `z` on the right hand side of $+$. The proof is by structural induction on n:

Base case ($n = \mathtt{z}$)**:** We need to show that $\mathtt{z} + \mathtt{z} = \mathtt{z}$. We can prove this by applying the *sum-z* rule where $n = \mathtt{z}$:

$$\frac{}{\mathtt{z} + \mathtt{z} = \mathtt{z}}\ \textit{sum-z}$$

Inductive case ($n = \mathtt{s}n'$)**:** We need to show that $n + \mathtt{z} = n$. Rewriting in terms of n', we have $\mathtt{s}n' + \mathtt{z} = \mathtt{s}n'$. Now, we are allowed to assume that the property we are proving is true for substructures of n. We call this assumption the induction hypothesis. One such substructure is n'. Thus we have $n' + \mathtt{z} = n'$ by applying the induction hypothesis to n'. Now we can finish the proof by applying the rule *sum-s*:

$$\frac{n' + \mathtt{z} = n'}{\mathtt{s}n' + \mathtt{z} = \mathtt{s}n'}\ \textit{sum-s}$$

Of course, this is not a complete derivation, but that's OK. When we assume the induction hypothesis, we are really assuming there is some derivation $\mathcal{D}$ that can be used to prove that $n' + \mathtt{z} = n'$. What we did in the last step is apply the rule *sum-s* with the conclusion of the entire derivation $\mathcal{D}$ as its premise, giving us an extended derivation with the desired conclusion. This kind of proof is often called an *constructive proof* because the proof conceptually constructs a complete derivation of the thing that is being proved.

■

4.6.4 Induction Over Derivations

EXAMPLE 4.28

Proving that if we add one to the right of a sum, we add one to the result

Syntax definitions are inductive structures–but so are derivations. That means we can do induction over them. This is useful to prove many properties. For example, consider the property that's symmetric to *sum-s*: for all n_1, n_2, and n_3 such that

$n_1 + n_2 = n_3$, we have $n1 + \mathtt{s}\ n_2 = \mathtt{s}\ n_3$. Let's call this *sum-s-rh* (it's a property of the sum judgment when you add an s to the right hand side of the +) You can actually prove this by induction on n_1, but it's a bit complicated to do so. Let's instead assume there is some derivation $\mathcal{D}$ of $n_1 + n_2 = n_3$, and do induction over that derivation. The derivation $\mathcal{D}$ must end with the application of some rule: either *sum-z* or *sum-s*, since those are the only two rules that can be used to derive a sum judgement. We'll finish the proof by considering each rule as a case.

Case $\dfrac{}{\mathtt{z} + n = n}\ \textit{sum-z}$: If we are applying rule *sum-z*, then n_1 must be z, and n_2 and n_3 must be the same number n, because otherwise it doesn't match the rule. Plugging the substituion $[\mathtt{z}/n_1, n/n_2, n/n_3]$ into the thing we have to show, we get $\mathtt{z} + \mathtt{s}\ n = \mathtt{s}\ n$ as the desired result. Here the notation $[\mathtt{z}/n_1, n/n_2, ...]$ means substitute z for n_1, n for n_2, etc. But we can just use the *sum-z* rule to show this:

$$\frac{}{\mathtt{z} + \mathtt{s}\ n = \mathtt{s}\ n}\ \textit{sum-z}$$

which finishes our case.

Case $\dfrac{n_1' + n_2 = n_3'}{\mathtt{s}\ n_1' + n_2 = \mathtt{s}\ n_3'}\ \textit{sum-s}$: Once again, we have a substitution: if we are using the *sum-s* rule to derive $n_1 + n_2 = n_3$, then n_1 must be $\mathtt{s}\ n_1'$ (for some number n_1') and similarly n_3 must be some number $\mathtt{s}\ n_3'$. Now, notice that if the derivation $\mathcal{D}$ ended with the above application of *sum-s*, there must be some derivation $\mathcal{D}'$ of the property $n_1'+n_2 = n_3'$ that is in the premise of the rule. $\mathcal{D}'$ is a *subderivation* of $\mathcal{D}$: it's a part of the derivation of $\mathcal{D}$. We are doing induction on the derivation $\mathcal{D}$, so we can assume the induction hypothesis about any subderivation, in particular the subderivation $\mathcal{D}'$. Thus we have $n_1' + \mathtt{s}\ n_2 = \mathtt{s}\ n_3'$ by applying the induction hypothesis to $\mathcal{D}'$.

Notice that now we can use rule *sum-s* as follows:

$$\frac{n_1' + \mathtt{s}\ n_2 = \mathtt{s}\ n_3'}{\mathtt{s}\ n_1' + \mathtt{s}\ n_2 = \mathtt{s}\ \mathtt{s}\ n_3'}\ \textit{sum-s}$$

But notice that this result, $\mathtt{s}\ n_1' + \mathtt{s}\ n_2 = \mathtt{s}\ \mathtt{s}\ n_3'$, is exactly what you get if you apply the substitution $[\mathtt{s}\ n_1'/n_1, \mathtt{s}\ n_3'/n_3]$ to the thing we were trying to prove, which was $n_1 + \mathtt{s}\ n_2 = \mathtt{s}\ n_3$. Thus we are done! ■

Once we have proved a property like *sum-s-rh*, we can use it just like a rule to prove other theorems. For example, we might want to prove that + is commutative. We can do so using structural induction, the rules *sum-z* and *sum-s*, and the theorems above: *sum-z-rh* and *sum-s-rh*. In fact, theorems like "+ is commutative" are the interesting ones; properties like *sum-z-rh* are mostly useful to prove commutativity, and so we call them *lemmas*: properties that are useful in proving a more interesting theorem.

4.6.5 Proofs by Induction Over Syntax

Proofs about numbers are fun, but can we prove things about programs? Consider the following grammar for expressions:

$$e \longrightarrow n \mid x \mid e+e \mid e*e$$

Let's define the *literals* of an expression to be all the n's and x's within it, and let's define the *operators* to be all the $+$'s and $*$'s within it. An interesting property is that the number of literals is always the number of operators plus one. Can we prove it?

EXAMPLE 4.29

Defining example properties of expressions: how many literals and operators they contain

First, let's define some rules that formalize the notion of literals and operators. First of all, we have a judgment $Lit(e) = n$ for defining the literals of an expression e. The rules are:

$$\frac{}{Lit(n) = 1}\ \textit{Lit-n}$$

$$\frac{}{Lit(x) = 1}\ \textit{Lit-x}$$

$$\frac{Lit(e_1) = n_1 \quad Lit(e_2) = n_2 \quad n_1 + n_2 = n_3}{Lit(e_1 + e_2) = n_3}\ \textit{Lit+}$$

$$\frac{Lit(e_1) = n_1 \quad Lit(e_2) = n_2 \quad n_1 + n_2 = n_3}{Lit(e_1 * e_2) = n_3}\ \textit{Lit*}$$

We can also define rules for an operator judgment, Ops(e) = n:

$$\frac{}{Ops(n) = 0}\ \textit{Ops-n}$$

$$\frac{}{Ops(x) = 0}\ \textit{Ops-x}$$

$$\frac{Ops(e_1) = n_1 \quad Ops(e_2) = n_2 \quad n_1 + n_2 + 1 = n_3}{Ops(e_1 + e_2) = n_3}\ \textit{Ops+}$$

$$\frac{Ops(e_1) = n_1 \quad Ops(e_2) = n_2 \quad n_1 + n_2 + 1 = n_3}{Ops(e_1 * e_2) = n_3}\ \textit{Ops*}$$

If we assume that numbers n are defined as before, we can take 0 as an abbreviation for `z` and 1 as an abbreviation for `s z`. In the rest of this subsection, I'll use numbers as in math, but remember that we could define and reason about them entirely using rules like *sum-s*. ■

EXAMPLE 4.30

Proving that in any expression, the number of literals is one greater than the number of operators

Now let's prove the property. For all expressions e, $Lit(e) = Ops(e) + 1$. We'll prove it by induction on e. This induction is a bit more interesting because e is tree-structured...the syntax $e_1 + e_2$ has two smaller bits of syntax within it, e_1 and e_2. So when we do the inductive step for the case of $e_1 + e_2$, we can apply the induction hypothesis twice: once for e_1 and once for e_2. This is OK because both e_1 and e_2 are subtrees of $e_1 + e_2$. The proof goes by case analysis on the last syntactic production used to construct e:

Case $e = x$:
$Lit(x) = 1$ by rule *Lit-x*
$Ops(x) = 0$ by rule *Ops-x*
So $Lit(x) = 1 = Ops(x) + 1 = 0 + 1 = 1$ (as mentioned above, we are doing the math in one step, rather than appealing to the judgments defining $+$)

Case $e = n$: $Lit(n) = 1$ by rule *Lit-n*
$Ops(n) = 0$ by rule *Ops-n*
So $Lit(n) = 1 = Ops(n) + 1 = 0 + 1 = 1$ (note: this case is analogous to that for $e = x$)

Case $e = e_1 + e_2$: $Lit(e_1) = Ops(e_1) + 1$ by the induction hypothesis applied to e_1
$Lit(e_2) = Ops(e_2) + 1$ by the induction hypothesis applied to e_2
$Lit(e_1 + e_2) = Lit(e_1) + Lit(e_2)$ by the rule *Lit+*
$Ops(e_1 + e_2) = Ops(e_1) + Ops(e_2) + 1$ by the rule *Ops+*
So $Lit(e_1+e_2) = Lit(e_1)+Lit(e_2) = Ops(e_1)+1+Ops(e_2)+1 = Ops(e_1+e_2)+1$ which is the result we needed to prove.

Case $e = e_1 * e_2$: The proof is analogous to the case for $e_1 + e_2$, above.
This concludes the proof. ∎