4.6 Reasoning Tools

The use of proof assistants is now common in work on programming language semantics. Proof assistants can be used to capture the semantics of languages as well as proofs about those semantics, and the tool can then automatically check that those proofs are correct. These tools can also be useful for learning programming language semantics, because they provide immediate feedback on whether definitions are well-formed and whether the reasoning in proofs are correct.

In this section, we show how SASyLF, a proof assistant designed for programming language education, can be used to capture the semantics of portions of the Calculator language from the text, and automatically check proofs about its semantics. SASyLF is unique among proof assistants in that it uses a textual syntax that is as close as possible to the notation we use for definitions nad proofs in this textbook. That means you don't have to spend a lot of time learning the syntax for the tool; you can pick it up quickly based on what you already know from the text.

Earlier in the chapter, we assumed certain things as a starting point, such as the definition of numbers and arithmetic operators. In proof assistants, we want to make sure all the definitions we build on are correct, and so it is common to formalize these ideas in the same framework that we use to formalize the programming language. Starting with arithmetic will also help in understanding how to use the proof assistant in a simple setting, before we encounter bound variables and other challenges.

4.6.1 Formalizing Numbers and Arithmetic

EXAMPLE 4.23
Representing Numbers

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 z represent the number zero. And if n is a number, then s n is the successor of that number. Using an abstract grammar, this can be written as:

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

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

SASyLF provides a way to describe abstract grammars, but must be able to parse them so the tool can read expressions in the language being formalized. We describe a language's syntax to SASyLF by first declaring the set of terminals in the language, and then a syntax made up of rules in BNF form:

```
terminals
    z s

syntax

n ::= z
  |   s n
```

SASyLF checks that the nonterminals defined in the syntax are distinct from the terminals, and that all symbols used on the right hand side are either nonterminals, terminals, or symbols like + which are implicitly assumed to be terminals.

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 SASyLF, we define this using the `judgment` keyword, the name of the judgment, and the judgment's syntax, as follows:

```
judgment sum: n + n = n
```

SASyLF checks to make sure that the name of the judgment is unique and that the symbols used in the judgment are declared terminals or nonterminals. We start to define this judgment by writing 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, $n + \mathtt{z} = n$.

In SASyLF, we define the rules for a judgment immediately after declaring the judgment. The rules are given as a line of two or more dashes, with premises on separate lines above the dashed line, the conclusion underneath, and the name of the rule to the right. Whenever a rule uses a nonterminal, we use the name of the nonterminal, with numbers or primes appended to distinguish it from others. SASyLF figures out the type of the nonterminal from its name, so that a nonterminal named `n3` must be an instance of the nonterminal `n` for numbers.

```
--------- sum-z
z + n = n

n1 + n2 = n3
----------------- sum-s
s n1 + n2 = s n3
```

SASyLF checks to make sure that each of the rules has a conclusion that is an instance of the judgment we are defining, and that each premise of each rule can be parsed as some judgment (not necessarily the one we are defining). ■

4.6.2 Derivations and Simple Theorems

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 $\texttt{s z} + \texttt{s s z} = \texttt{s s s z}$. 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 $\texttt{z} + \texttt{s s z} = \texttt{s s z}$. 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: $\texttt{s z} + \texttt{s s z} = \texttt{s s s z}$.

Unfortunately SASyLF doesn't have syntax that directly mimics derivations, mainly because capturing the branching structure of derivation trees in a literal syntax isn't practical. Instead, you can declare a simple theorem stating that a fact is true. To prove the theorem, instead of giving a derivation, you state the intermediate facts in the tree one at a time, starting with the leaves of the derivation, and giving a justification for each fact. For example, we can write the above derivation as a theorem and its proof in SASyLF as follows:

```
theorem one-plus-two-is-three:
    exists s z + s s z = s s s z.
    d1: z + s s z = s s z by rule sum-z
    d2: s z + s s z = s s s z by rule sum-s on d1
end theorem
```

The above theorem simply states that there exists a derivation of `s z + s s z = s s s z`. It is proved by giving the derivation. Each line of the proof is an assertion of a fact, a justification, and a name for the derivation of that fact. The name comes first; we often use names starting with d (for derivation) and with sequential numbers, so `d1`, `d2`, etc. Then there is the fact being asserted, and finaly the justification. Here we are justifying by rule. We give the name of the rule and, if the rule has any premises, we give the name of the derivation of each premise. So the justification `by rule sum-s on d1` means that we are applying rule `sum-s` using `d1` as its one and only premise. When we run SASyLF, the tool checks that the judgment asserted in the `exists` clause is well-formed. It then verifies each line of the proof, making sure that applying the given rule to the given premises yields the given result. Finally, it checks that the last line of the proof is exactly the judgment whose existence the theorem asserts. ■

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, $n + \mathtt{z} = n$. 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 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.

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.

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. ■

The example above motivates and explains mathematical induction well, but it's too complex for an initial example with a proof assistant, so we'll start with simpler examples to illustrate the basic ideas.

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, $\mathtt{s}n$). 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.

Now let's look at how this proof can be expressed in SASyLF:

```
theorem sum-z-rh:
    forall n
    exists n + z = n.
    proof by induction on n:
        case z is
            d1: z + z = z by rule sum-z
        end case
        case s n' is
            d1: n' + z = n' by induction hypothesis on n'
            d2: s n' + z = s n' by rule sum-s on d1
        end case
    end innduction
end theorem
```

This time, we're stating a theorem that must be true for all numbers `n`. Therefore, the theorem states that for all `n` there exists a derivation that `n + z = n`. In general, SASyLF proves forall-exists theorems: that is, theorems that start with zero or more forall clauses and end with one or more exists clauses. This time, we can't prove the theorem simply by giving a derivation, because there will be different derivations for different input numbers `n`. Instead, we prove the theorem by induction on `n`, just like the paper version of the proof. There are two cases, one for each syntax of `n`. In the first case we prove the result with the `sum-z` rule, as expected. In the second case, we assume that `n` is of the form `s n'`. Note that we name the number `n'` differently from `n` so that we distinguish them; SASyLF keeps track of the fact that `n` is equivalent to `s n'`. In fact, if we use `n` later in this case, SASyLF will substitute `n` with `s n'` in its internal reasoning.

Now we prove the second case. We first use the induction hypothesis; the syntax for doing is is like a rule, but we have to provide a "premise" for the induction hypothesis. This "premise" is just the input to the theorem. The "premise" must be a subtree of `n`, the thing we are doing induction on, in order for this to be a valid use of the induction hypothesis. In this case, it is valid; `n` is equivalent to `s n'` so `n'` is a subtree of `n`. We finish the case by applying the `sum-s` rule.

■

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 $\mathtt{z}$, 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 $\mathtt{z}$ 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!

Let's look at how we can prove this theorem in SASyLF:

```
theorem sum-s-rh:
    forall d: n1 + n2 = n3
    exists n1 + s n2 = s n3.
    proof by induction on d:
        case rule
            -------------- sum-z
            d1: z + n2 = n2
            where n1 := z and n3 := n2
        is
            _: z + s n2 = s n2 by rule sum-z
        end case
        case rule
            d1: n1' + n2 = n3'
            ---------------------- sum-s
            d: s n1' + n2 = s n3'
            where n1 := s n1' and n3 := s n3'
        is
            d3: n1' + s n2 = s n3' by induction hypothesis on d1
            proof by rule sum-s on d3
        end case
    end induction
end theorem
```

Now the theorem assumes not just a number `n` but a judgment `n1 + n2 = n3`, so we put that in the `forall` clause and also give the derivation of that input judgment a name `d`. The proof is by induction on that derivation. Now the cases are rules instead of syntax; we put the entire rule in the case, labeling each premise and conclusion with a name like `d1` and appropriately substituting variables into the rule so that the rule makes sense. In particular, the conclusion should match the input derivation, except that some parts of it might be constrained by the rule. For example, in the first case, the rule constrains `n1` to be `z` and `n3` to be the same as `n2`. We therefore write the conclusion as `z + n2 = n2`. It is helpful to remember that `n1` from the input derivation is `z` in this case, and we can document this with a `where` clause; similarly the where clause documents that `n3` is `n2`. Where clauses are optional because SASyLF can infer these based on how the rule was given, but they make the proof more readable and so they are recommended. SASyLF checks each case to make sure it is a valid instantiation of the rule. It also checks that the conclusion of the rule matches the input derivation in as general a way as possible given that this rule is being used. Finally, SASyLF checks that the where clauses are correct. In the remainder of the case, the variables `n1` and `n3` that appear on the left side of a where clause are substituted with the right hand side. For clarity, we do that already when we write down the proof for this case. The proof of the first case is easy; we just apply the `sum-z` rule. In cases like this one where we do not care about the name of a derivation, we can use the wildcard _ for the name.

The second case is only a little bit more interesting. We apply the induction hypothesis; since the theorem we are proving takes a judgment as input, we pass in a derivation of that judgment as the premise of the induction hypothesis. SASyLF checks that we are using the induction hypothesis on a subtree of the derivation `d` that we are doing induction over; in this case, `d1` is indeed a subtree because it is a premise of the `sum-s` case that is coming from a case analysis of `d`. Using the rule `sum-s` completes the proof. As a convenience, the keyword `proof` always means the derivation that we are trying to prove; this makes the last line in the proof of the case a bit shorter.

■

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. ■

EXAMPLE 4.31
Proving that addition is deterministic

Let's see how one can prove slightly more complicated and interesting theorems in SASyLF, and look at the process by which one proves theorems in practice. One example is proving that addition is deterministic. To formalize this we need a way to formalize equality. We can do so with a judgment expressed in SASyLF as follows:

```
judgment equal: n = n

----- eq
n = n
```

Now we can state the theorem as follows: if two numbers `n1` and `n2` sum to `n3` and also to `n4`, then `n3` and `n4` must be the same. To prove it, we can use induction over either of the input `forall` derivations in the theorem—we arbitrarily pick the first:

```
theorem add-deterministic:
    forall d1: n1 + n2 = n3
    forall d2: n1 + n2 = n4
    exists n3 = n4.
    proof by induction on d1:
    end induction
end theorem
```

Given the above input, SASyLF issues an error that we have not considered all the cases in the induction. We can write the cases ourselves, and initially it may be good practice to do so. After learning how to do this, however, it's very convenient to use the Eclipse plugin for SASyLF and apply a quick fix (right-click on the error, choose quick fix, and press "Finish" when the dialog comes up for the "insert missing cases" fix) to automatically generate all the necessary cases.

To fill in the proof, we need a high-level strategy. If you are not yet comfortable writing fluidly in SASyLF, it may be helpful to outline a proof strategy on paper first, and then see how to write it in the tool; later, it may be natural to go right to the tool. In this case, to prove the theorem, we need to show that the derivation `d2` identically mirrors the derivation `d1` that we are doing induction over; this will then imply that `n4` is the same as `n3`. The first case that was generated for us is:

```
case rule
    --------------- sum-z
    _: z + n3 = n3
is
    proof by unproved
end case
```

It may be helpful to add a clause `where n1 := z and n2 := n3` to document what we know about `n1` and `n2` as a result of the assumption that the `sum-z` rule was applied. We need to show the derivation `d2` is also of the form `z + n3 = n3`. To do that, we can case-analyze on `d2`, as follows:

```
proof by case analysis on d2:
    case rule
        --------------- sum-z
        _: z + n4 = n4
    is
        proof by unproved
    end case
end case analysis
```

Here, the case in the case analysis can also be generated by the SASyLF quick fix once we write the `proof by case analysis on d2: ... end case analysis` part. It may seem that we have not made any progress. But think about the substitution implicit in this case. The case uses `n4` for where the original derivation `d2` used both `n2` and `n4`, meaning that in this case we have substituted `n4` for `n2`. If we write the obvious `where n2 := n4` SASyLF says this is redundant, as `n2` has already been replaced with `n3`, as per the where clause above. Instead, we need to reflect that `n3`, which was the replacement for `n2`, has itself been replaced with `n4`. So the appropriate where clause is `where n3 := n4`. This is exactly what we need to show: that `n3` and `n4` are the same. To fulfill our proof obligation, we produce a *witness* of this fact by constructing a judgment `n3 = n4` as follows:

```
proof by case analysis on d2:
    case rule
        --------------- sum-z
        _: z + n4 = n4
        where n3 := n4
    is
        _: n3 = n4 by rule eq
    end case
end case analysis
```

This completes the case. The next case is:

```
case rule
    d3: n0 + n2 = n5
    ----------------------- sum-s
    _: (s n0) + n2 = (s n5)
    where n1 := s n0 and n3 := s n5
is
    proof by unproved
end case
```

Once again we need to do case analysis on `d2` to show that it exactly mirrors the derivation `d1`. There is only one case possible in the inner case analysis. We get:

```
proof by case analysis on d2:
    case rule
        d4: n0 + n2 = n6
        ---------------------- sum-s
        _: (s n0) + n2 = (s n6)
        where n4 := s n6
    is
        proof by unproved
    end case
end case analysis
```

It turns out we can clean this up a little bit. This case analysis is degenerate, in the sense that there is only one case, and all we really need from the case is what the premise is (i.e. `d4`) and the substitutions in the where clause generated (i.e. `where n4 := s n6`). SASyLF provides a convenient construct for this. We use the principle of *inversion* to get the premise(s) of a rule, along with substitutions, in a case analysis that has only one case. It's call "inversion" because we are using case analysis to "invert" the rule, going from the conclusion to the premises (rather than the other way around, which is the way we usually work when constructing a derivation). The syntax looks like this:

```
d4: n0 + n2 = n6 by inversion on d2 where n4 := s n6
proof by unproved
```

This means exactly the same thing as the previous proof snippet, but it is 2 lines long instead of 10 and avoids adding extra indentation levels to the proof when there is only one case. We work towards finishing the proof by adding a use of induction, which will allow us to conclude that the derivation `d4` is equivalent to the earlier derivation `d3`:

```
d4: n0 + n2 = n6 by inversion on d2 where n4 := s n6
d5: n5 = n6 by induction hypothesis on d3, d4
proof by unproved
```

Now we have a judgment that states that `n5` equals `n6`. But there's nothing special about the `n5 = n6` judgment that would automatically tell SASyLF that `n5` and `n6` are identical as variables. We can, however, argue to SASyLF that they must be by doing a case analysis on `d5`, the equality judgment. There will be only one case and the conclusion is `n6 = n6`, meaning that `n5` has been substituted with `n6`. We can write this formally as:

```
d4: n0 + n2 = n6 by inversion on d2 where n4 := s n6
d5: n5 = n6 by induction hypothesis on d3, d4
proof by case analysis on d5:
    case rule
        ----------- eq
        _: n6 = n6
        where n5 := n6
    is
        proof by unproved
    end case
end case analysis
```

Once again, we have a case analysis with only one case, so can we use inversion? Yes, but it's slightly awkward because there is no premise that we "get" from applying inversion; instead, all we really care about is the knowledge that `n5 := n6`. There's another form of the inversion construct, `use inversion`, where all we get out is the where clause, for expressing this:

```
d4: n0 + n2 = n6 by inversion on d2 where n4 := s n6
d5: n5 = n6 by induction hypothesis on d3, d4
use inversion on d5 where n5 := n6
proof by unproved
```

Again, this is equivalent to the immediately preceeding proof snippet. Finally, we can finish the proof by using the `eq` rule to show that `n3` (i.e. the successor of `n5`) is equal to `n4` (i.e. the succcesor of `n6`). Here's the entire proof:

```
theorem add-deterministic:
    forall d1: n1 + n2 = n3
    forall d2: n1 + n2 = n4
    exists n3 = n4.
    proof by induction on d1:
        case rule
            --------------- sum-z
            _: z + n3 = n3
            where n1 := z and n2 := n3
        is
            proof by case analysis on d2:
                case rule
                    --------------- sum-z
                    _: z + n4 = n4
                    where n3 := n4
                is
                    _: n3 = n4 by rule eq
                end case
            end case analysis
        end case

        case rule
            d3: n0 + n2 = n5
            ----------------------- sum-s
            _: (s n0) + n2 = (s n5)
            where n1 := s n0 and n3 := s n5
        is
            d4: n0 + n2 = n6 by inversion on d2 where n4 := s n6
            d5: n5 = n6 by induction hypothesis on d3, d4
            use inversion on d5 where n5 := n6
            _: s n5 = s n6 by rule eq
        end case
    end induction
end theorem
```

We can see that the use of the inversion construct saved us from writing down a lot of cases, but the underlying ideas are the same ones we've been using in prior proofs: induction, case analysis, substitutions, and inference rules.

■