1 Recursively defining collections by rules

We often want to define a collection of objects by a series of recursive rules[1]. For example, here is a set of rules to define the collection Nat of natural numbers:

$$\frac{}{\mathsf{Z} : \mathsf{Nat}}\ \mathsf{Znat} \qquad\qquad \frac{n : \mathsf{Nat}}{\mathsf{S}(n) : \mathsf{Nat}}\ \mathsf{Snat}$$

The rule Znat says that we can always conclude that the symbol Z is a Nat; the rule Snat says that we can conclude if n is a Nat then the application of the symbol S to n is also a Nat[2]. We say that these rules are *recursive* because the rule Snat has a premise $(n : \mathsf{Nat})$ that only makes sense using the rules we are in the process of defining. What does Nat contain? Consider four possibilities:

A	incomplete	$\mathsf{Nat_A}$	$\equiv$	$\{\mathsf{Z}, \mathsf{S}(\mathsf{Z})\}$
B	uninvertible	$\mathsf{Nat_B}$	$\equiv$	$\{\mathsf{Z}, \mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{Z})), \ldots, \infty, \mathsf{S}(\infty), \mathsf{S}(\mathsf{S}(\infty)), \ldots\}$
C	inductive	$\mathsf{Nat_C}$	$\equiv$	$\{\mathsf{Z}, \mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{Z})), \ldots\}$
D	coinductive	$\mathsf{Nat_D}$	$\equiv$	$\{\mathsf{Z}, \mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{Z})), \ldots, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\ldots)))))\}$

Choice (A) is not right because it is not *complete*. That is, rule Snat tells us that if $n : \mathsf{Nat}$, then $\mathsf{S}(n) : \mathsf{Nat}$; choice (A) does not obey this rule since $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$ is not in $\mathsf{Nat_A}$. For the same kind of reason we know that Nat is nonempty: rule Znat tells us that Z must be in Nat. Choice (B) in some sense has the opposite problem: it contains too much. When we say that a collection is defined by a set of rules, we need the objects to be *invertible*. That is, every object in the collection must have been constructed using that series of rules, so if $n : \mathsf{nat}$, then we must be able to conclude that $n = \mathsf{Z}$ or that there exists some $n' : \mathsf{Nat}$ such that $n = S(n')$ (if n was constructed with Snat). Of course, neither of these conclusions is satisfied for ∞, so we rule out $\mathsf{Nat_B}$.

For the remainder of this course, we will require that all collections of objects defined by rules be both complete and invertible[3]. Observe that both $\mathsf{Nat_C}$ and $\mathsf{Nat_D}$ are complete and invertible, but that they are not equal.

Choice (C) defines the *minimal* (or *least*) complete and invertible collection satisfying the pair of rules Znat and Snat. In contrast, choice (D) defines the *maximal* (*greatest*) complete and invertible collection satisfying the rules Znat and Snat. Here, we use "least" and "greatest" to refer to the subset ordering; *i.e.*, if $\mathbb{S}$ is a collection of sets, then $S \in \mathbb{S}$ is the least element if

$$\forall S' : \mathbb{S}(S \subseteq S')$$

whereas $S \in \mathbb{S}$ is the greatest element if

$$\forall S' : \mathbb{S}(S' \subseteq S)$$

[1] We use "collection", "set", and "type" interchangeably to indicate a group of objects.

[2] In set theory, this would be written $\mathsf{Z} \in \mathsf{Nat}$; here we use notation from a related body of mathematics called *type theory*, in which we indicate that an object v belongs to a type τ by writing $v : \tau$. To a first approximation, objects are elements and types are sets; $v \in \tau \approx v : \tau$.

[3] Collections defined by rules must be complete in both set theory and type theory. Such collections need not be invertible in set theory, while in type theory they must be invertible.

Given a set of rules R, we say that the least collection satisfying R is the *inductively defined* set generated by R. In contrast, the greatest collection satisfying R is the *coinductively defined* set generated by R.

There is another way of looking at the difference between inductively defined sets and coinductively defined sets. Recall that a collection C is invertible with respect to a set of rules R if every element $c : C$ can be constructed using the rules. In an inductively defined set, we require that element's construction be finite—or, to say it another way, every element can be built from a terminating computation applying the rules in R. In contrast, in a coninductively defined set, we also include all elements whose construction is infinite—that is, we also include all elements that can be built using nonterminating computation.

This understanding may help us understand the difference between $\mathsf{Nat_C}$ and $\mathsf{Nat_D}$. The collection $\mathsf{Nat_C}$ contains all of the objects generated by a finite application of the rules Znat and Snat; the "..." in the definition of $\mathsf{Nat_C}$ just indicates that we keep going using the rule Snat, including elements like $\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))$, etc. While we are defining an infinite collection of elements, each individual element is just some finite number of applications of S applied to the terminal element Z. In contrast, the collection $\mathsf{Nat_D}$ has two occurrences of "..."; the first one is indicating that just like in the case of $\mathsf{Nat_C}$, we keep applying rule Snat to include elements like $\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))$; therefore,

$$\mathsf{Nat_C} \subset \mathsf{Nat_D}$$

However, $\mathsf{Nat_D}$ contains another element, which we will call ω, and which is indicated in the definition by $\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\ldots)))))$. The element ω is generated by an infinite application of the rule Snat; alternatively you can think of ω as the result of an unterminating computation. Since ω requires an infinite construction, $\omega \notin \mathsf{Nat_C}$. Observe that $\mathsf{Nat_D}$ is complete since

$$\omega \quad = \quad \mathsf{S}(\omega)$$

Exercise 1. Prove $\mathsf{Nat_D}$ is invertible: $\forall n : \mathsf{Nat_D}\big(n = \mathsf{Z} \ \vee \ \exists n'(n = \mathsf{S}(n'))\big)$.

Observation. Suppose we wanted to construct $\mathsf{Nat_B}$. Define the rule

$$\frac{}{\infty : \mathsf{nat}} \ \infty\mathsf{nat}$$

The inductively defined set generated by the rules Znat, Snat, and $\infty\mathsf{nat}$ is $\mathsf{Nat_B}$.

Convention and Notation. In this course we concentrate on inductively defined sets, so unless we specify otherwise, a set defined by a set of rules is assumed to be inductively defined. We also use a shorthand notation to write down a series of recursive definitions as follows:

$$\mathbb{C} \quad = \quad G_1(\ldots) \mid \ \ldots \ \mid G_n(\ldots)$$

Here, $\mathbb{C}$ is the type of objects we wish to define and $G_1 \ldots G_n$ are a (finite) collection of generators (constructors) separated by the distinguished symbol

"|". The "..." inside a $G_i(\ldots)$ indicate the type of the parameters (arguments) to the generators; the type of parameters **can** include $\mathbb{C}$ itself. By convention, if a generator takes no arguments, the "()" are omitted. For example:

$$\mathsf{Nat_C} \quad = \quad \mathsf{Z} \mid \mathsf{S}(\mathsf{Nat_C})$$

In contrast, we would define the type $\mathsf{Nat_B}$ by:

$$\mathsf{Nat_B} \quad = \quad \mathsf{Z} \mid \infty \mid \mathsf{S}(\mathsf{Nat_B})$$

To define the type $\mathsf{Nat_D}$, we use the same pattern as $\mathsf{Nat_C}$,

$$\mathsf{Nat_D} \quad = \quad \mathsf{Z} \mid \mathsf{S}(\mathsf{Nat_D})$$

but explicitly specify that the rules should be interpreted coninductively.

1.1 Defining collections by rules in Coq

Coq makes it quite easy to define inductive types; in fact the notation used is inspired from the notation just explained. Here is how to define the type $\mathsf{Nat_C}$:

```
Inductive NatC : Type :=
  | Z : NatC
  | S : NatC -> NatC.
```

That is, Coq should define a new inductively-defined `Type`[4] named `NatC` with two production rules (generators): `Z`, which takes no parameters; and `S`, which takes a `NatC` as an argument[5]. It is simple to define a few elements of `NatC`:

```
Definition Zero : NatC := Z.
Definition One : NatC := S Zero.
(* or S(Zero), if you like extra "(" and ")" *)
Definition Two : NatC := S One.
```

Similarly, we can define the type $\mathsf{Nat_B}$ as:

```
Inductive NatB : Type :=
  | NB_Z : NatB
  | Infinity : NatB
  | NB_S : NatB -> NatB.
```

[4]Coq uses the keyword `Type` to define a new type (set, collection) of objects. You may have also noticed that Coq has a related keyword, `Set`. They keyword `Set` is a way of letting Coq know that the object being defined is "simple" in a certain technical sense. Everything that is of type `Set` could have been defined instead to have `Type`, but not the reverse; that is, in some sense, `Set` $\subsetneq$ `Type`. Things start to get complicated when you realize that the type of `Set` is actually `Type`—that is, `Set:Type`. The type of `Type` is *almost* `Type` too—that is, `Type:Type` is well-defined in Coq, but the interpretation is a little subtle. Things can start getting tricky pretty fast; fortunately we are not likely to run into the thornier areas in this course; we will just use `Type` everywhere and trust that Coq will prevent us from getting into trouble.

[5]Technically, it's possible to omit the initial "|" before the first generator (`Z` in this case), but most people put it in so that the generators line up nicely.

Notice that we name the first and third generators for `NatB` "`NB_Z`" and "`NB_S`" so that there is no naming conflict with the generators for `NatC`. As with `NatC`, it is easy to define a few elements of `NatB` such as:

```
Definition NB_Zero : NatB := NB_Z.
Definition NB_Two := NB_S (NB_S NB_Zero).
(* Notice we left out the ": NatB" - Coq was able to guess it. *)
```

Coq also lets you define the coinductively defined set $\mathsf{Nat_D}$ in a very similar way:

```
CoInductive NatD : Type :=
  | ND_Z : NatD
  | ND_S : NatD -> NatD.
```

Notice that here we use the keyword `CoInductive` rather than `Inductive`. Defining the simpler elements of `NatD` is no harder than in the previous cases:

```
Definition ND_Zero := ND_Z.
Definition ND_One := ND_S ND_Zero.
```

The interesting question, of course, is how you define the infinite element ω. Since Coq scripts must be finite, we cannot just write

```
Definition omega : NatD := ND_S (ND_S (ND_S (ND_S (ND_S ...))))
```

We will show how to properly define ω in section 2.2.

1.2 More examples of defining sets by rules.

Since examples are very helpful, here are a few more recursively defined types:

Example 1: Binary trees (without data).

$$\frac{}{\bullet : \mathsf{Tree}}\ \mathsf{Leaftree} \qquad \frac{t_l : \mathsf{Tree} \qquad t_r : \mathsf{Tree}}{t_l \quad t_r \ : \mathsf{Tree}}\ \mathsf{Nodetree}$$

Using our notational shorthand, we can write:

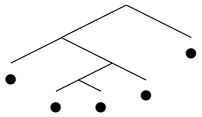
$$\mathsf{Tree} \quad = \quad \bullet \quad | \quad \mathsf{Tree} \quad \mathsf{Tree}$$

Here is how we write these definitions in Coq:

```
Inductive Tree : Type :=
  | Leaf : Tree
  | Node : Tree -> Tree -> Tree.
```

Exercise 2. Write down the set of objects inductively defined by the set of rules Ltree and Ntree in the style of $\mathsf{Nat_C}$ above.

Exercise 3. Define the following tree in Coq:



Example 2: Formulas in propositional logic. Here we use $\mathsf{Nat_C}$ for atoms:

$$\frac{p : \mathsf{Nat_C}}{p : \mathsf{Formula}}\ \mathsf{Atom} \qquad \frac{F_1 : \mathsf{Formula} \qquad F_2 : \mathsf{Formula}}{F_1 \ \wedge\ F_2 \ : \ \mathsf{Formula}}\ \mathsf{Conjunction}$$

$$\frac{F_1 : \mathsf{Formula} \qquad F_2 : \mathsf{Formula}}{F_1 \ \vee\ F_2 \ : \ \mathsf{Formula}}\ \mathsf{Disjunction} \qquad \frac{F : \mathsf{Formula}}{\neg F \ : \ \mathsf{Formula}}\ \mathsf{Negation}$$

$$\frac{F_1 : \mathsf{Formula} \qquad F_2 : \mathsf{Formula}}{F_1 \rightarrow F_2 \ : \ \mathsf{Formula}}\ \mathsf{Implication}$$

We can define this collection in our shorthand notation by starting with

$$\mathsf{Formula} \ = \ \mathsf{Atom}(\mathsf{Nat_C}) \mid \ldots$$

Exercise 4. Fill in the "..." in the above shorthand definition.

Although you have already seen this done in Coq homework two (and Coq quiz two), here is the Coq implementation for the above definitions:

```
Inductive Formula : Type :=
  | Atom : NatC -> Formula
  | Conj : Formula -> Formula -> Formula
  | Disj : Formula -> Formula -> Formula
  | Neg : Formula -> Formula
  | Impl : Formula -> Formula -> Formula.
```

Example 3: Polymorphic lists. Things can now get a bit more complicated. We can define the collection of lists of some type τ with the following rules:

$$\frac{}{\mathsf{nil}_\tau \ : \ \mathsf{list}(\tau)}\ \mathsf{Nil} \qquad \frac{v : \tau \qquad L : \mathsf{list}(\tau)}{v ::_\tau L \ : \ \mathsf{list}(\tau)}\ \mathsf{Cons}$$

Rather than defining a single type, here we define a **family** of types indexed by another type τ[6]. For example, if $\tau = \mathsf{Nat_C}$, then we have the set of lists with elements $\mathsf{Nat_C}$. We can define an element of type $\mathsf{list}(\mathsf{NatC})$ as follows:

$$\mathsf{aNatList} : \mathsf{list}(\mathsf{Nat_C}) \ \equiv \ \mathsf{Z} ::_{\mathsf{Nat_C}} \mathsf{S(Z)} ::_{\mathsf{Nat_C}} \mathsf{Z} ::_{\mathsf{Nat_C}} \mathsf{nil}_{\mathsf{Nat_C}}$$

[6]The technical term for the type of lists of arbitrary type is *polymorphic*.

Having to explicitly write the type τ for the generators nil_τ and $::_\tau$ can be a real pain on the eyes. Usually they can be easily guessed, and we can just write:

$$\mathsf{aNatList} : \mathsf{list}(\mathsf{Nat_C}) \;\equiv\; \mathsf{Z} :: \mathsf{S}(\mathsf{Z}) :: \mathsf{Z} :: \mathsf{nil}$$

Still, you have to be a little careful; let's consider a more complicated case. The type of lists of lists of naturals is $\mathsf{list}(\mathsf{list}(\mathsf{NatC}))$. Consider the element

$$\mathsf{aNatListList} : \mathsf{list}(\mathsf{list}(\mathsf{Nat_C})) \;\equiv\; \mathsf{aNatList} :: \mathsf{nil} :: \mathsf{nil}$$

This looks weird; what is the second nil doing? Things get easier to disambiguate at the cost of added clutter when we put in the explicit type parameters:

$$\mathsf{aNatListList} : \mathsf{list}(\mathsf{list}(\mathsf{Nat_C})) \;\equiv\; \mathsf{aNatList} ::_{\mathsf{list}(\mathsf{Nat_C})} \boxed{\mathsf{nil}_{\mathsf{Nat_C}}} ::_{\mathsf{list}(\mathsf{Nat_C})} \mathsf{nil}_{\mathsf{list}(\mathsf{Nat_C})}$$

The first nil (boxed) has the type parameter $\mathsf{Nat_C}$—that is, the boxed nil has type $\mathsf{list}(\mathsf{Nat_C})$. Since $\mathsf{aNatListList}$ is a list whose elements have type $\mathsf{list}(\mathsf{Nat_C})$, the first (boxed) nil is allowed to be an element. In contrast, the second nil has type parameter $\mathsf{list}(\mathsf{Nat_C})$—that is, has type $\mathsf{list}(\mathsf{list}(\mathsf{Nat_C}))$, so it ends $\mathsf{aNatListList}$.

Convention for type parameters. In general, we will omit the type parameters to avoid clutter; however, remember that they are still in the background.

Polymorphic lists in Coq. In Coq it is easy to define polymorphic lists:

```
Inductive PolyList (A : Type) : Type :=
  | Nil : PolyList A
  | Cons : A -> PolyList A -> PolyList A.
```

Now we define the type of lists of naturals, and its element aNatList, as follows:

```
Definition aNatList : PolyList NatC :=
  Cons NatC Z (Cons NatC (S Z) (Cons NatC Z (Nil NatC))).
```

This is clear enough, but again, the type parameters are adding a lot of pain. Fortunately, we can tell Coq that we want it to guess them by writing:

```
Implicit Arguments PolyList [A].
Implicit Arguments Nil [A].
Implicit Arguments Cons [A].
```

These instructions tell Coq to try hard to guess the parameter `A` when the user writes `PolyList`, `Nil`, and `Cons`. Now we can just write:

```
Definition aNatList2 : PolyList :=
  Cons Z (Cons (S Z) (Cons Z Nil)).
```

What is nice about the situation is that now we can easily define aNatListList:

```
Definition aNatListList :=
  Cons aNatList (Cons Nil Nil).
```

Notice how Coq automatically figured out that the first `Nil` has type `PolyList NatC`; Coq also figured out that `aNatListList` has type `PolyList (PolyList NatC)`. Coq is usually **very** good at guessing types, but sometimes it is not enough. For example, consider the following:

```
Definition aNil : PolyList :=
  Nil.
```

If you try this, Coq does not have enough information to guess which type you want, and it gives you the following cryptic error message:

```
Error: Cannot infer the implicit parameter A of Nil.
```

You need to give it some kind of hint by using the symbol `@`, as follows:

```
Definition aNil : PolyList :=
  @ Nil NatC.
```

By using `@`, you tell Coq to turn off implicit arguments for the next function (constructors like `Nil` are a special kind of function). Of course, instead of providing the explicit parameter for `Nil`, you could provide it for `PolyList`:

```
Definition aNil : @ PolyList NatC :=
  Nil.
```

In either case, Coq has enough information to guess what is missing.

Example 4: Streams of natural numbers. Consider the following rule:

$$\frac{n : \mathsf{nat} \qquad s : \mathsf{stream}}{n \mathbin{@} s \ : \ \mathsf{stream}} \ \mathsf{Strm}$$

Let stream be the set coinductively defined by the rule Strm. Observe that one element $\mathsf{Sconst_Z} : \mathsf{stream}$ is the constant stream generated by Z:

$$\mathsf{Sconst_Z} \quad \equiv \quad \mathsf{Z} \mathbin{@} \mathsf{Z} \mathbin{@} \mathsf{Z} \mathbin{@} \ldots$$

Another is the increasing stream starting from Z, $\mathsf{Sinc_Z} : \mathsf{stream}$:

$$\mathsf{Sinc_Z} \quad \equiv \quad \mathsf{Z} \mathbin{@} \mathsf{S(Z)} \mathbin{@} \mathsf{S(S(Z))} \mathbin{@} \ldots$$

To define streams in Coq is pretty simple:

```
CoInductive NatStream : Type :=
  NS : NatC -> NatStream -> NatStream.
```

We will defer the Coq definitions of $\mathsf{Sconst_Z}$ and $\mathsf{Sinc_Z}$ until section 2.2.

1.3 Closing observations on sets defined by rules

We close with a few observations. You probably noticed that the sets $\mathsf{Nat_C}$ and $\mathsf{Nat_D}$ differed by only one element, and you might have wondered if there were any sets "in the middle". For the case of the rules Znat and Snat, the answer is clearly no; in general, however, it can happen.

Exercise 5. **($\star$)** Give a series of rules and a complete and invertible set of objects satisfying those rules that is **neither** the least nor greatest.

Built-in inductive types in Coq. Our type `NatC` is actually almost identical to a built-in type `nat` in Coq; the only difference is that the built-in type uses `O` (`0` is also ok in most contexts) instead of `Z` to indicate the base object of type `nat`. Similarly, our type `PolyList` is very similar to the built-in type `list`. If you want to use `list`, then you need to import the list library in Coq by writing

```
Require Import List.
```

The built-in list type uses `nil` for the terminal element and `cons` to attach an element to the head of an existing list; for convenience the notation "`::`" can be used *infix* (that is, in the middle) instead of `cons`, as follows:

```
Definition anEmptyList : list nat := nil.
Definition aListoOfLists := (3 :: nil) :: foo :: nil.
```

The dangers of recursively defined objects. One must be a little careful when defining a set by recursive rules. Consider the following attempt:

$$\mathbb{S} \quad = \quad \mathsf{S}_1 \mid \mathsf{S}_2(\mathbb{S} \to \mathbb{S})$$

Here we use "$A \to B$" to indicate the type of **functions** from some type A to B. This is an attempt to inductively define a set with two generators: S_1, which takes no arguments (like Z for $\mathsf{Nat_C}$); and S_2, which takes one argument, which must be a function from $\mathbb{S}$ to $\mathbb{S}$. There is a simple set-theoretic argument[7] that says that the cardinality (*i.e.*, size) of a set $\mathbb{S}$ is **strictly** lower than the cardinality of functions from $\mathbb{S}$ to $\mathbb{S}$. In other words, the above definition implies that $\mathbb{S}$ has a **strictly** higher cardinality than itself—the set theoretic version of asking for an integer n such that $n < n$. Of course, that is impossible!

One of the advantages of using Coq is that it can verify that a desired recursive definition is sound[8]. Let's see what Coq does with $\mathbb{S}$:

```
Inductive Unsound : Type :=
  | S1 : Unsound
  | S2 : (Unsound -> Unsound) -> Unsound.
```

[7] For example, see `http://en.wikipedia.org/wiki/Cantor\%27s_diagonal_argument`.

[8] Coq's verification is sound—everything it accepts is well-defined. However, in this place it is incomplete—there are some sound definitions that it rejects—but it is still pretty useful.

That is, define the constructor `S1` that takes no arguments, and a second constructor `S2` that takes a single argument, which itself must be a function from `Unsound` to `Unsound`. Coq rejects the definition with the error message:

```
Error: Non strictly positive occurrence of "Unsound" in
 "(Unsound -> Unsound) -> Unsound".
```

You are unlikely to run into this problem as long as your recursive definitions avoid function types, but it is still something to be aware of. If you have a definition that you are unsure about, try defining it in Coq.

Exercise 6. **(⋆⋆)** Explain what Coq is doing here, and define (on paper) a recursive definition that is sound but the `Inductive` keyword of Coq rejects.

2 Defining functions on sets defined by rules

Once we have a recursively defined set like Nat, we want to be able to define functions (operations) on it[9]. For example, we would like to define the predicate that is true ($\top$) when given Z, and false ($\bot$) otherwise. Here is a first attempt:

$$\mathsf{isZero}(n) \quad \equiv \quad \begin{cases} \top & \text{when } n = \mathsf{Z} \\ \bot & \text{when } n = \mathsf{S}(\mathsf{Z}) \end{cases}$$

Here we use a notation that may be familiar from mathematics courses; we break the input n into *cases*. In the first case, when $n = \mathsf{Z}$, isZero returns $\top$; in the second case, when $n = \mathsf{S}(\mathsf{Z})$, isZero returns $\bot$. It is worth stressing that breaking the input into cases only works because we have an invertible set.

Coq can take cases using the `match` construction, as follows:

```
Definition isZero (n : NatC) : Prop :=
  match n with
   | Z => True
   | S Z => False
  end.
```

Hopefully the basic pattern is clear: we are taking cases on `n`, indicated by `match n`, and have defined two cases separated by "`|`". In the first case, when `n = Z`, we evaluate to `True` (*i.e.,* $\top$); while in the second, when `n = S Z`, we evaluate to `False` ($\bot$). We finish the case analysis with the keyword `end`.

The `match...end` construction is quite powerful and we will in no way cover all of what it can do. Instead, we will focus on the simpler use cases... and what can go wrong. In fact, the code above results in the following error:

```
Error: Non exhaustive pattern-matching:
  no clause found for pattern S (S _)
```

[9] From here on, we will write Nat to mean the inductively defined set from the rules Znat and Snat—that is, $\mathsf{Nat} = \mathsf{Nat}_{\mathsf{C}}$.

Coq is complaining that the `match` is not covering all of the cases. Coq is right: our definition of isZero evaluates correctly on Z and S(Z), but is undefined on other input such as S(S(Z)). Coq insists (and it is a good idea in general) that every function is *total*—that is, a function must be defined on every input[10].

With this in mind, we try the following definition:

$$\mathsf{isZero}(n) \quad \equiv \quad \begin{cases} \top & \text{when } n = \mathsf{Z} \\ \bot & \text{when } n = \mathsf{S}(n') \end{cases}$$

We again break n into cases, but this time our second case covers more choices. In fact, by invertibility, we know that this breakdown covers all the possibilities. It is simple to code this definition in Coq; this time there are no complaints.

```
Definition isZero (n : NatC) : Prop :=
  match n with
   | Z => True
   | S n' => False
  end.
```

No complaints, but there are a few subtle things going on here. First of all, `n'` is introduced as a variable binding; that means that the second case (`False`) is able to use `n'` if it wants to. Here is an example of using that power:

```
Definition isOne (n : NatC) : Prop :=
  match n with
   | Z => False
   | S n' => isZero n'
  end.
```

Here we define a function that returns `False` on zero (`Z`) and when passed some `S(n')` returns `isZero(n')`. When `n'` is `Z` (*i.e.*, `n = S Z`), then the result is `True`; otherwise (*i.e.*, when `n = S (S n'')` for some `n''`) the result is `False`. The result is a function that is `True` on `S Z` and `False` otherwise. Here is the same function written using the mathematical notation:

$$\mathsf{isOne}(n) \quad \equiv \quad \begin{cases} \bot & \text{when } n = \mathsf{Z} \\ \mathsf{isZero}(n') & \text{when } n = \mathsf{S}(n') \end{cases}$$

The way variables are bound can be a little tricky. Consider:

```
Definition isOne' (n : NatC) : Prop :=
  match n with
   | Z => False
   | S n' => isZero n
  end.
```

[10] In a setting like ours where we require that functions be total, it is straightforward to model a *partial* function—a function that does not need to evaluate to a result on every input—from total functions: just evaluate all of the undefined inputs to some special-purpose object indicating that the function does not cover this input.

The code for `isOne'` is **almost** the same as the code for `isOne` above. The difference is in the second case; here, we use `isZero n` instead of `isZero n'`. The result is a function that is **always** `False`. Another very similar example is:

```
Definition isOne'' (n : NatC) : Prop :=
  match n with
   | Z => False
   | S n => isZero n
  end.
```

The difference here is again in the second case. Instead of binding the new variable to a fresh variable such as `n'`, here we reuse the variable `n`. Just like with quantifiers, variables in patterns bind to the closest binding. In other words, `isOne''` behaves like `isOne`, and **not** like `isOne'`.

There are other wrinkles to case analysis; one of these is the notion of *wildcards*, inspired from the following alternative definition of isZero:

$$\mathsf{isZero}(n) \quad \equiv \quad \begin{cases} \top & \text{when } n = \mathsf{Z} \\ \bot & \text{otherwise} \end{cases}$$

Since the isZero function does not use the n' in the second case, we can write "otherwise" rather than "$n = \mathsf{S}(n')$". Coq generalizes this idea as follows:

```
Definition isZero (n : NatC) : Prop :=
  match n with
   | Z => True
   | S _ => False
  end.
```

The wildcard "_" tells Coq that this case should be used regardless of what is in this spot (such as `Z` or `S Z` or `n`); no variable names are introduced. Cases are evaluated top to bottom: the first pattern, then the second, etc. until Coq has found a match. Therefore, the following code has exactly the same effect:

```
Definition isZero (n : NatC) : Prop :=
  match n with
   | Z => True
   | _ => False
  end.
```

Here the "_" will match anything—it is the way to say, "otherwise". Since the only case left is `S n'`, the result is identical. **Not** identical is the following:

```
Definition isZero (n : NatC) : Prop :=
  match n with
   | _ => False
   | Z => True
  end.
```

The problem here is that it is **impossible** to reach the second case since the initial case matches `Z` and everything else! Thus Coq rejects the definition:

```
Error: This clause is redundant.
```

There is one final thing to be aware of when using `match`. Consider the following reasonable pair of definitions for defining the isTwo predicate:

$$\mathsf{two} \quad \equiv \quad \mathsf{S}(\mathsf{S}(\mathsf{Z}))$$

$$\mathsf{isTwo}(n) \quad \equiv \quad \begin{cases} \top & \text{when } n = \mathsf{two} \\ \bot & \text{otherwise} \end{cases}$$

Here's a reasonable attempt to code this in Coq:

```
Definition two : NatC := S (S Z).
Definition isTwo (n : NatC) : Prop :=
  match n with
   | two => True
   | _ => False
  end.
```

Unfortunately, Coq returns a somewhat mysterious error message:

```
Error: This clause is redundant.
```

What is going on? It turns out that Coq is interpreting `two` as a fresh variable name instead of as a possible pattern to match. Therefore, the first clause matches all of the cases—that is, in the first case, the fresh variable `two` can be `Z` or `S Z` or anything else—leaving nothing for the second case, which is therefore redundant. It turns out that Coq **requires** that the pattern only include the generators (constructors) defined when the inductive type was created—in this case, `Z` and `S`. Given this, one way to write the `isTwo` function is as follows:

```
Definition isTwo (n : NatC) : Prop :=
  match n with
   | S (S Z) => True
   | _ => False
  end.
```

This works just fine, and the wildcard in the second case matches everything except `S (S Z)`. You might wonder why Coq does not allow for more general pattern matching. There are several technical reasons. Two of the simpler are the desire to check for missing cases and redundancy when the definition is written (as opposed to when it is used); and that Coq requires that there be a computable test for which case is taken. In general, equality is not computable (if I give you two functions `f` and `g` from `NatC` to `NatC`, it might take an infinite amount of time to determine if they are equal).

In the case of the `isTwo` function, however, we can use a neat trick. We take advantage of the fact that the result of the `isTwo` function is `Prop`, which also happens to be the type of equality facts. Thus we can write:

```
Definition isTwo (n : NatC) : Prop :=
  match n with
   | n' => n' = two
  end.
```

In this case, the match is so simple that our trick does not buy us much; in fact, the above code is equivalent to the following much more obvious definition:

```
Definition isTwo (n : NatC) : Prop :=
  n = two.
```

Still, in general it is useful to remember that if your result is `Prop` you can test for equality on the right-hand side of the pattern in addition to the left.

2.1 Fixpoints

We now understand the details of defining a function by cases. Let us use our newfound understanding to investigate recursive functions, such as the following:

$$\mathsf{isEven}(n) \quad \equiv \quad \begin{cases} \top & \text{when } n = \mathsf{Z} \\ \bot & \text{when } n = \mathsf{S}(\mathsf{Z}) \\ \mathsf{isEven}(n') & \text{when } n = \mathsf{S}(\mathsf{S}(n')) \end{cases}$$

We first observe that this function is total because all of the cases are covered. The tricky issue is that there is a recursive call in the function; is this sound? Here it depends again on what what we want our functions to be. In mathematics it is common to define functions as subsets of the cross product of the domain and codomain (range) with certain properties. In computer science, we very often want much stronger properties. By default, Coq requires that functions be both *computable* and *terminating*[11]. Unrestricted recursion makes it easy to define nonterminating functions, so Coq makes recursive functions a little trickier to define. Here is our first natural-looking attempt:

```
Definition isEven (n : NatC) : Prop :=
  match n with
   | Z => True
   | S Z => False
   | S (S n') => isEven n'
  end.
```

Unfortunately, Coq rejects this code, and complains as follows:

```
Error: The reference isEven
  was not found in the current environment.
```

In other words, the keyword `Definition` does not let us use the object we are defining recursively. Instead, we need a related keyword, `Fixpoint`, as follows:

[11] It is possible to define uncomputable functions, for example by enabling axioms such as description or unrestricted choice—but the additional power comes at a price.

```
Fixpoint isEven (n : NatC) {struct n} : Prop :=
  match n with
   | Z => True
   | S Z => False
   | S (S n') => isEven n'
  end.
```

This definition works. The unexpected item is the {`struct n`}. What is this for? It tells Coq that the *principal argument* of `isEven` is `n`. The principal argument is used to verify that the recursion will terminate. The reason is that since objects in inductively defined sets have a finite construction, as long as we restrict ourselves to only recursing on subparts of our original expression, then we know we will eventually reach a base case, when we will terminate[12].

When defining a recursive function, Coq always needs to know what the principal argument is so that it can verify that the function terminates. Fortunately, Coq is pretty good at guessing; in fact in most cases we can leave the {`struct ...`} out. In the example we gave above, Coq is able to examine the structure of this function and determine that `n` is the principal argument (ok, in this case it was easy since there was only one argument). When the user adds the recursive definition, Coq can then respond:

```
isEven is recursively defined (decreasing on 1st argument)
```

The 1st argument being `n`. What happens if we try a nonterminating function?

```
Fixpoint unguarded (n : NatC) : Prop :=
  unguarded n.
```

Coq rejects this definition with the following error message:

```
Error: Recursive definition of unguarded is ill-formed.
  In environment unguarded : NatC -> Prop n : NatC
  Recursive call to unguarded has principal argument
  equal to "n" instead of a subterm of n.
```

A second example: map. If you are familiar with a functional programming language (Lisp, Scheme, ML, Haskell, etc.) then you are probably familiar with a function called map, defined on polymorphic lists. Here is the definition:

$$\mathsf{map}_{(A,B)}(f : A \to B,\ L : \mathsf{list}(A)) \ : \ \mathsf{list}(B)$$
$$\equiv \begin{cases} \mathsf{nil}_B & \text{when } L = \mathsf{nil}_A \\ f(a) \ ::_B \ \mathsf{map}_{(A,B)}(f, L') & \text{when } L = a ::_A L' \end{cases}$$

We put in the types of the arguments f and L, as well as the type of the result of map (*i.e.*, $\mathsf{list}(B)$). We also added the explicit type operators on the list

[12] Note that we do not require that the subpart be direct subpart: that is, we can recurse on `n'` although it is only an indirect subpart (*i.e.*, a subpart of a subpart) of `n`. Coq also allows the definition of recursive functions with more sophisticated termination arguments. If you like, investigate the `Program` keyword in the user manual for detail.

constructors nil and "::". The map function takes a list L of As and a function f from A to B and uses f to transform L into a list of Bs. Of course, very often $A = B$, but even so f can be very useful. For example, here we use map to increment (*i.e.*, apply the generator/function S to) every nat in a list[13]:

$$\mathsf{map}(\mathsf{S},\ \mathsf{Z} :: \mathsf{S}(\mathsf{Z}) :: \mathsf{Z} :: \mathsf{nil}) \quad = \quad \mathsf{S}(\mathsf{Z}) :: \mathsf{S}(\mathsf{S}(\mathsf{Z})) :: \mathsf{S}(\mathsf{Z}) :: \mathsf{nil}$$

Let us take a look at the code for this function in Coq:

```
Fixpoint map (A B : Type) (f : A -> B) (L: list A) : list B :=
  match L with
    | nil => nil
    | a :: L' => (f a) :: map A B f L'
  end.
```

Observe that `map` terminates since the recursive call occurs on `L'`, which is the tail (back part) of the initial list `L`. Indeed, Coq reports that:

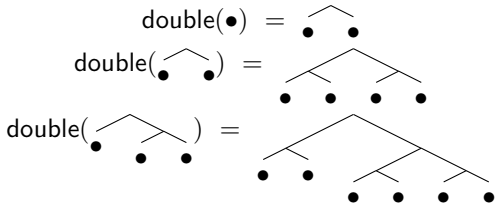
```
map is recursively defined (decreasing on 4th argument)
```

Of course, we want Coq to guess the type arguments `A` and `B`:

```
Implicit Arguments map [A B].
```

Exercise 7. Define (paper and Coq) an addition function on naturals.

Exercise 8. Define (paper and Coq) a function that doubles all of the leaves in a tree. For example, this function should behave as follows:



2.2 Cofixpoints

Both inductive and coinductive types have constructors/generators (*e.g.*, for nat, the constructors are Z and S), which are ways to **introduce** objects of the type. Both inductive and coinductive types are invertible, which means that case analysis as covered in section 2 can **eliminate** (use) objects of the type. We have already seen several examples of case analysis on inductive types; let us first examine a case analysis on a coinductive type. Recall the definition of stream given by the coinductive interpretation of the following rule:

$$\frac{n : \mathsf{nat} \qquad s : \mathsf{stream}}{n \mathbin{@} s \ : \ \mathsf{stream}}\ \mathsf{Strm}$$

[13] Here we will once again revert to hiding the type arguments to avoid clutter.

It is quite natural to define a function $\mathsf{head} : \mathsf{stream} \to \mathsf{nat}$ that returns the natural at the head of the stream as follows:

$$\mathsf{head}(s) \quad \equiv \quad \Big\{ n \quad \text{when } s = n \mathbin{@} s'$$

This is straightforward. Here is the (previously presented) Coq defining streams:

```
CoInductive NatStream : Type :=
  NS : NatC -> NatStream -> NatStream.
```

Defining the function to get the head of a stream is equally straightforward:

```
Definition head (ns : NatStream) : NatC :=
  match ns with
   | Strm n _ => n
  end.
```

So far inductive and coinductive types seem similar: both use generators for introduction and case analysis for elimination. However, as we have seen, inductive types also have a more powerful kind of elimination rule: fixpoints. A fixpoint is able to **eliminate** (use) a complete inductive object, instead of just a single layer of a complete object like case analysis does. Unfortunately, we cannot use fixpoints to eliminate coinductive types since the existence of infinite objects means that our functions might not terminate.

Instead of fixpoints, coinductive types are associated with *cofixpoints*. Cofixpoints **introduce** (build) coinductive types. Thus, inductive types get a more powerful elimination technique (fixpoints), while coinductive types get a more powerful introduction technique (cofixpoints). The lack of a powerful elimination technique poses a serious challenge for those interested in using coinductive types. Indeed, coinductive types are a little like hell: easy to get in to, but hard to escape. Still, from time to time they come in handy.

Let us see an example of a cofixpoint in action. Recall that we wanted to define the constant stream generated by Z, which looks something like this:

$$\mathsf{Sconst}_{\mathsf{Z}} \quad \equiv \quad \mathsf{Z} \mathbin{@} \mathsf{Z} \mathbin{@} \mathsf{Z} \mathbin{@} \ldots$$

A cofixpoint is just what we need; we simply write:

$$\mathsf{Sconst}_{\mathsf{Z}} \quad \equiv \quad \mathsf{Z} \mathbin{@} \mathsf{Sconst}_{\mathsf{Z}}$$

This looks like a very bad idea since it appears to contain unguarded recursion; recall that for fixpoints, we were only allowed to make a recursive call on **subparts** of our input. For a cofixpoint, we also have a guard, but it is in some sense the opposite: we are only allowed to recurse after we have been "wrapped" in a coinductive generator, in the case of $\mathsf{Sconst}_{\mathsf{Z}}$, the recursive call occurs in the context of the stream constructor "@". Thus, the result of a cofixpoint **must** be a coinductive type[14]. Here is the Coq code defining this constant stream:

[14] If you are familiar with lazy functional languages like Haskell, you are probably aware of computation on demand. The idea here is similar: here we have generation on demand; the corecursive function is able to generate as many elements of the stream as required.

```
CoFixpoint SconstZ : NatStream :=
  Strm Z ScontZ.
```

Coq responds with:

```
SconstZ is corecursively defined
```

If the recursive call does not occur in the context of a coinductive constructor, Coq will reject the definition just like an unguarded recursive call in a fixpoint.

Cofixpoints can also take arguments like normal functions. We also wanted to define the increasing stream starting from Z, $\mathsf{Sinc}_{\mathsf{Z}} : \mathsf{stream}$:

$$\mathsf{Sinc}_{\mathsf{Z}} \quad \equiv \quad \mathsf{Z} \mathbin{@} \mathsf{S}(\mathsf{Z}) \mathbin{@} \mathsf{S}(\mathsf{S}(\mathsf{Z})) \mathbin{@} \ldots$$

We break the definition into two parts. First we will define a generalized corecursive function that generates an increasing stream from some starting value:

$$\mathsf{incStream}(n) \quad \equiv \quad n \mathbin{@} \mathsf{incStream}(\mathsf{S}(n))$$

Now it is easy to define the increasing stream that starts from Z:

$$\mathsf{Sinc}_{\mathsf{Z}} \quad \equiv \quad \mathsf{incStream}(\mathsf{Z})$$

Finally, here is the Coq code for these definitions:

```
CoFixpoint incStream (n : NatC) : NatStream :=
  Strm n (incStream (S n)).
Definition SincZ : NatStream :=
  incStream Z.
```

Exercise 9. Define (on paper and in Coq) the infinite element ω in the coinductive interpretation of the natural numbers (rules Znat and Snat).

3 Proof techniques for recursive types

We have seen how to recursively define sets, and how to define recursive functions over such sets. What remains is to learn the proof techniques for reasoning about recursively defined sets. These techniques are analogous to the elimination techniques available—defining functions by cases and defining fixpoints.

3.1 Case analysis

Properties of both inductive and coinductive types can be proved by a technique called *case analysis*. Suppose we define a function called dec that decrements a natural number (with the convention that $\mathsf{dec}(0) = 0$)[15].

$$\mathsf{dec}(n) \quad \equiv \quad \begin{cases} Z & \text{when } n = \mathsf{Z} \\ n' & \text{when } n = \mathsf{S}(n') \end{cases}$$

[15] To cut down on clutter, we will start using the arabic numerals to indicate naturals; $0 = \mathsf{Z}$, $1 = \mathsf{S}(\mathsf{Z})$, $2 = \mathsf{S}(\mathsf{S}(\mathsf{Z}))$, etc. In addition, we assume standard arithmetical facts like $1 > 0$.

Let us prove that if n is nonzero, then n is the successor of its decrement, *i.e.*,

$$\forall n \ \big((n \neq 0) \Rightarrow (n = \mathsf{S}(\mathsf{dec}(n)))\big)$$

To prove this by case analysis, we must show that it is true when $n = \mathsf{Z}$ and also when $n = \mathsf{S}(n')$. We begin by clearly stating the technique we will use:

Proof. By case analysis on the structure of n.

Next, we break out each individual case, and prove the goal, as follows:

1. $(n = \mathsf{Z})$. Here the goal reduces to:

   $$0 \neq 0 \ \Rightarrow \ (0 = \mathsf{S}(\mathsf{dec}(n)))$$

   Fortunately, since the premise of the implication is false, this is very easy.

2. $(n = \mathsf{S}(n'))$. Here the goal reduces to:

   $$\mathsf{S}(n') \neq 0 \ \Rightarrow \ (\mathsf{S}(n') \ = \ \mathsf{S}(\mathsf{dec}(\mathsf{S}(n'))))$$

   Of course here the premise is true, so we need to prove the conclusion. Unfolding the definition of dec yields:

   $$\mathsf{S}(n') \neq 0 \ \Rightarrow \ (\mathsf{S}(n') \ = \ \mathsf{S}(\begin{cases} \mathsf{Z} & \text{when } \mathsf{S}(n') = \mathsf{Z} \\ \mathsf{n}'' & \text{when } \mathsf{S}(n') = \mathsf{S}(n'') \end{cases}))$$

   Which is just the same as

   $$\mathsf{S}(n') \neq 0 \ \Rightarrow \ (\mathsf{S}(n') \ = \ \mathsf{S}(n'))$$

   The conclusion is true by reflexivity, so we are done with case 2.

Thus by case analysis we have proved our goal.

Observation. We can use case analysis on any invertible set; dec is well-defined even if the naturals had been coinductively defined instead of inductively defined, and in that case our proof would have worked exactly the same way.

3.2 Case analysis in Coq.

Doing case analysis in Coq is pretty straightforward; the most common tactic used is `destruct`. Let us see an example. Here is the code for decrement:

```
Definition dec (n : NatC) : NatC :=
  match n with
   | Z => Z
   | S n' => n'
  end.
```

We now state our desired theorem:

```
Lemma Sdec: forall n,
  n <> Z -> n = S(dec n).
```

After the usual "`Proof. intros.`", Coq shows the following proof state:

```
1 subgoal
n : NatC
H : n <> Z
______________________________________(1/1)
n = S (dec n)
```

We tell Coq to prove the goal by case analysis on `n` by typing "`destruct n.`":

```
2 subgoals
H : Z <> Z
______________________________________(1/2)
Z = S (dec Z)
______________________________________(2/2)
S n = S (dec (S n))
```

The first goal corresponds to the case when `n=Z` and is easy sice we have a hypothesis that `Z <> Z`; we use "`contradiction H. trivial.`". The second goal corresponds to the case when `n=S n0`. However, Coq does some on-the-fly variable renaming that can be a bit confusing:

```
1 subgoal
n : NatC
H : S n <> Z
______________________________________(1/1)
S n = S (dec (S n))
```

You might have instead expected to see something more along the lines of:

```
1 subgoal
n0 : NatC
H : S n0 <> Z
______________________________________(1/1)
S n0 = S (dec (S n0))
```

However, Coq has determined that the old `n` is no longer needed (since it is just equal to `S(`the new `n)`. Therefore, Coq decided (a little confusingly) to use the same string ("`n`") to refer to the new `n` in the second goal. In any event, the proof is almost done. We use "`simpl. trivial.`" to finish, and then:

```
Proof completed.
```

Observation. Just as in a paper proof, the same proof script would work if `NatC` had been coinductively defined instead of inductively defined.

3.3 Induction

Just as inductively defined types have a more powerful elimination form called fixpoints, they also come come with a more powerful proof technique called *structural induction*. In a very important sense, structural induction is defining a fixpoint in a proof. The technique of structural induction is among the most important in mathematics. We illustrate the technique with some examples.

Example 5. Recall the inductive definition of trees:

$$\mathsf{Tree} \quad = \quad \bullet \quad | \quad \overset{\mathsf{Tree}\quad\mathsf{Tree}}{\wedge}$$

Define the fixpoint $\mathsf{leaves} : \mathsf{Tree} \to \mathsf{nat}$ that counts the leaves of a tree:

$$\mathsf{leaves}(t) \quad \equiv \quad \begin{cases} 1 & \text{when t} = \bullet \\ \mathsf{leaves}(t_l) + \mathsf{leaves}(t_r) & \text{when } t = \overset{t_l \quad t_r}{\wedge} \end{cases}$$

Define a second fixpoint $\mathsf{nodes} : \mathsf{Tree} \to \mathsf{nat}$ that counts the nodes of a tree:

$$\mathsf{nodes}(t) \quad \equiv \quad \begin{cases} 0 & \text{when t} = \bullet \\ 1 + \mathsf{nodes}(t_l) + \mathsf{nodes}(t_r) & \text{when } t = \overset{t_l \quad t_r}{\wedge} \end{cases}$$

We want to prove that for any tree t, the number of leaves in t is greater than the number of nodes in t. That is,

$$\forall t \ \big(\mathsf{leaves}(t) > \mathsf{nodes}(t)\big)$$

The key to doing a good induction proof is to write out all of the steps and avoid shortcuts. Doing it right takes time, but at the end you know that you have a dependable result. Start out with a clear statement of your goal (the previous equation) and the technique you would like to apply.

Proof. By induction on the structure of the tree t.

Next, clearly state your induction hypothesis. You hypothesis must be a **predicate** (function to truth values) on the variable you are inducting over and should be equal to your goal if that variable is universally quantified over; *i.e.*,

$$\mathsf{IH}(t) \quad \equiv \quad \mathsf{leaves}(t) > \mathsf{nodes}(t)$$

Now you need to take cases on the variable you are inducting over. State each case clearly, showing what t is on this case and what the induction hypothesis is on any subparts. The point of induction is that you can use the induction hypothesis on the subparts to build the proof on the combined object. In each case, you need to prove $\mathsf{IH}(t)$ at the end[16]. If you do so, then induction lets you conclude that the induction hypothesis holds for all variables, that is,

$$\forall t \ \mathsf{IH}(t)$$

[16]For induction proofs, we do not require a three-column proof such as we use in natural deduction; however, you may choose to do one if you find it a useful format.

With this in mind, we do the cases for this proof as follows:

1. $(t = \bullet)$. Since $\bullet$ has no subparts, we do not have any induction hypothesis. Our goal is to prove $\mathsf{IH}(\bullet)$, that is, $\mathsf{leaves}(\bullet) > \mathsf{nodes}(\bullet)$. This is direct since $\mathsf{leaves}(\bullet) = 1$ and $\mathsf{nodes}(\bullet) = 0$ and $1 > 0$. We are done with case 1.

2. $(t = \overset{\frown}{t_l \quad t_r})$. We can assume IH on the subparts of t; that is, we can assume $\mathsf{IH}(t_l)$ and $\mathsf{IH}(t_r)$. Our goal is to prove $\mathsf{IH}(t)$; that is, $\mathsf{leaves}(t) \geq \mathsf{nodes}(t)$. Since $t = \overset{\frown}{t_l \quad t_r}$, this reduces to:

$$\mathsf{leaves}(\overset{\frown}{t_l \quad t_r}) \;>\; \mathsf{nodes}(\overset{\frown}{t_l \quad t_r})$$

   Unfolding the definitions of leaves and nodes leads to the following:

$$\mathsf{leaves}(t_l) + \mathsf{leaves}(t_r) \;>\; 1 + \mathsf{nodes}(t_l) + \mathsf{nodes}(t_r)$$

   Our induction hypotheses are:

$$\mathsf{leaves}(t_l) > \mathsf{nodes}(t_l) \qquad \text{and} \qquad \mathsf{leaves}(t_r) > \mathsf{nodes}(t_r)$$

   This is equivalent to:

$$\mathsf{leaves}(t_l) \geq 1 + \mathsf{nodes}(t_l) \qquad \text{and} \qquad \mathsf{leaves}(t_r) \geq 1 + \mathsf{nodes}(t_r)$$

   Combining the above, we know that

$$\mathsf{leaves}(t_l) + \mathsf{leaves}(t_r) \geq 2 + \mathsf{nodes}(t_l) + \mathsf{nodes}(t_r)$$

   This directly implies that

$$\mathsf{leaves}(t_l) + \mathsf{leaves}(t_r) > 1 + \mathsf{nodes}(t_l) + \mathsf{nodes}(t_r)$$

   And so we are done with case 2.

At the end of the proof, it is important to restate that you have finished all of the cases by saying, "thus, by structural induction we have shown that"

$$\forall t \; \big(\mathsf{leaves}(t) > \mathsf{nodes}(t)\big)$$

Example 6. Let us consider a second example. We define a function called $\mathsf{size} : \mathsf{Tree} \to \mathsf{nat}$ that counts the total number of leaves and nodes in a tree:

$$\mathsf{size}(t) \;\equiv\; \begin{cases} 1 & \text{when t} = \bullet \\ 1 + \mathsf{size}(t_l) + \mathsf{size}(t_r) & \text{when } t = \overset{\frown}{t_l \quad t_r} \end{cases}$$

We want to prove the size of a tree is equal to the number of leaves plus nodes:

$$\forall t \; \big(\mathsf{size}(t) \;=\; \mathsf{leaves}(t) + \mathsf{nodes}(t)\big)$$

Proof. By induction on the structure of t. We use

$$\mathsf{IH}(t) \quad \equiv \quad \mathsf{size}(t) \;=\; \mathsf{leaves}(t) + \mathsf{nodes}(t)$$

1. $(t = \bullet)$. This case is easy by a series of equalities:

$$\begin{array}{rcl} \mathsf{size}(\bullet) & = & 1 \\ & = & \mathsf{leaves}(\bullet) \\ & = & \mathsf{leaves}(\bullet) + 0 \\ & = & \mathsf{leaves}(\bullet) + \mathsf{nodes}(\bullet) \end{array}$$

   By transitivity of equality, we are done with case 1.

2. $(t = \overset{\frown}{t_l \;\; t_r})$. Our induction hypotheses are

   $\mathsf{size}(t_l) = \mathsf{leaves}(t_l) + \mathsf{nodes}(t_l)$ and $\mathsf{size}(t_r) = \mathsf{leaves}(t_r) + \mathsf{nodes}(t_r)$

   Given these hypotheses, we again use a series of equalities to prove $\mathsf{IH}(t)$:

$$\begin{array}{rcl} \mathsf{size}(\overset{\frown}{t_l \;\; t_r}) & = & 1 + \mathsf{size}(t_l) + \mathsf{size}(t_r) \\ & = & 1 + \mathsf{leaves}(t_l) + \mathsf{nodes}(t_l) + \mathsf{leaves}(t_r) + \mathsf{nodes}(t_r) \\ & = & \big(1 + \mathsf{nodes}(t_l) + \mathsf{nodes}(t_r)\big) + \big(\mathsf{leaves}(t_l) + \mathsf{leaves}(t_r)\big) \\ & = & \mathsf{nodes}(\overset{\frown}{t_l \;\; t_r}) + \mathsf{leaves}(\overset{\frown}{t_l \;\; t_r}) \end{array}$$

   We are therefore done with case 2.

So by structural induction we have proved that

$$\forall t \;\; \big(\mathsf{size}(t) \;=\; \mathsf{leaves}(t) + \mathsf{nodes}(t)\big)$$

Exercise 10. Recall the double function you defined for exercise 8. Prove

$$\forall t \;\; \big(\mathsf{leaves}(\mathsf{double}(t)) \;=\; \mathsf{leaves}(t) + \mathsf{leaves}(t)\big)$$

Exercise 11. Recall from example 4 above that we coinductively define stream from the rule Strm. Suppose we try to define stream inductively rather than coinductively. Prove that in that case, stream is empty; that is,

$$\neg\exists s : \mathsf{stream}(\top)$$

Note that this is (deMorgan-) equivalent to:

$$\forall s : \mathsf{stream}(\bot)$$

If you set this up correctly, the proof should be very short.

3.4 Induction in Coq.

After all of the effort required to properly set up an induction proof on paper, you may be quite happy to learn that doing induction in the theorem prover is often quite easy. To illustrate, we will redo our previous two examples in Coq.

Example 5 in Coq. Recall from example 1 our Coq definition for trees:

```
Inductive Tree : Type :=
  | Leaf : Tree
  | Node : Tree -> Tree -> Tree.
```

It is quite straightforward to define the leaves and nodes fixpoints; in fact they are almost a direct translation from the paper definitions:

```
Fixpoint leaves(t : Tree) : nat :=
  match t with
   | Leaf => 1
   | Node tl tr => leaves tl + leaves tr
  end.

Fixpoint nodes(t : Tree) : nat :=
  match t with
   | Leaf => 0
   | Node tl tr => 1 + nodes tl + nodes tr
  end.
```

One point that is worth making is that the result type is `nat`, the built-in type for naturals in Coq. Actually, except for the name of the base generator (it uses `O` for `Z`), it is exactly the same as our `NatC`. However, by using it we have access to the large number of pre-existing definitions, lemmas, and tactics for reasoning about natural numbers. In particular, we will be using a tactic called `omega`, which makes short work of arithmetical goals. To use `omega`, we need to first include the library where it is defined by including this line in our file:

```
Require Import Omega.
```

It is quite simple to state the goal for our lemma; notice we use the symbol "`>`" for greater than; this is predefined on type `nat`.

```
Lemma leaves_gt_nodes: forall t,
  leaves t > nodes t.
Proof.
```

Coq now reports one goal, looking like this:

```
1 subgoal
______________________________________(1/1)
forall t : Tree, leaves t > nodes t
```

We would like to solve this goal by induction on the structure of `t`; the Coq tactic to begin is logically enough just "`induction t.`". We now have two goals:

```
2 subgoals
______________________________________(1/2)
leaves Leaf > nodes Leaf
______________________________________(2/2)
leaves (Node t1 t2) > nodes (Node t1 t2)
```

Each goal corresponds to one of the cases in the induction. The first one is quite easy; we just do "`simpl.`", which transforms the goal into "`1>0`". Now we have a pure arithmetical fact, and can take advantage of "`omega.`" to solve the first goal. Now we must solve the second goal, which looks like this:

```
1 subgoal
t1 : Tree
t2 : Tree
IHt1 : leaves t1 > nodes t1
IHt2 : leaves t2 > nodes t2
______________________________________(1/1)
leaves (Node t1 t2) > nodes (Node t1 t2)
```

We simplify the `leaves` and `nodes` functions with "`simpl.`", and then we have:

```
1 subgoal
t1 : Tree
t2 : Tree
IHt1 : leaves t1 > nodes t1
IHt2 : leaves t2 > nodes t2
______________________________________(1/1)
leaves t1 + leaves t2 > S (nodes t1 + nodes t2)
```

This looks good, but doing all the steps is a pain. Instead, we say, "`omega.`".

```
Proof completed.
```

The proof script is only five lines long. Pretty good, eh?

Example 6 in Coq. Since it is not a lot of work, we also redo example six:

```
Fixpoint size(t : Tree) : nat :=
  match t with
   | Leaf => 1
   | Node tl tr => 1 + size tl + size tr
  end.
```

The size function is very simple to define, and writing our goal is also easy:

```
Lemma size_eq_leavesnodes: forall t,
  size t = leaves t + nodes t.
```

The proof script is only five lines (ok, not counting `Proof.` and `Qed.`).

```
Proof.
  induction t.
  simpl.
  trivial.
  simpl.
  omega.
Qed.
```

Doing inductions in Coq is usually easier than doing them formally on paper. It does take some practice—and you do not get to "fudge it" when the details do not quite work out as you expected—but even so, it's usually a huge win.

Exercise 12. Redo exercise 10 in Coq.

Exercise 13. Redo exercise 11 in Coq. The proof script should be quite short.

3.5 Strengthening the induction hypothesis

Induction is something that seems quite simple, but in fact induction is one of the most complicated and error-prone mathematical techniques to master. Let us take a look at a slightly more complicated example.

Example 7. It is **vital** that you follow all of the steps. Let us consider a problem closely related to example 5. We want to prove that

$$\forall t \ \big(\mathsf{leaves}(t) \geq \mathsf{nodes}(t)\big)$$

Of course, this follows directly from the fact that we have proven example 5 ($\forall x \forall y \ (x > y \Rightarrow x \geq y)$), but let us instead try to prove it directly.

Proof. By induction on the structure of the tree t. We use

$$\mathsf{IH}(t) \quad \equiv \quad \mathsf{leaves}(t) \geq \mathsf{nodes}(t)$$

1. ($t = \bullet$). Since $\bullet$ has no subparts, we do not have any induction hypothesis. Our goal is to prove $\mathsf{IH}(\bullet)$, that is, $\mathsf{leaves}(\bullet) \geq \mathsf{nodes}(\bullet)$. This is direct since $\mathsf{leaves}(\bullet) = 1$ and $\mathsf{nodes}(\bullet) = 0$ and $1 \geq 0$. We are done with case 1.

2. ($t = \overset{\frown}{t_l \ \ t_r}$). We can assume IH on the subparts of t; that is, we can assume $\mathsf{IH}(t_l)$ and $\mathsf{IH}(t_r)$. Our goal is to prove $\mathsf{IH}(t)$; that is, $\mathsf{leaves}(t) \geq \mathsf{nodes}(t)$. Since $t = \overset{\frown}{t_l \ \ t_r}$, this reduces to:

   $$\mathsf{leaves}(\overset{\frown}{t_l \ \ t_r}) \ \geq \ \mathsf{nodes}(\overset{\frown}{t_l \ \ t_r})$$

   Unfolding the definitions of leaves and nodes leads to the following:

   $$\mathsf{leaves}(t_l) + \mathsf{leaves}(t_r) \ \geq \ 1 + \mathsf{nodes}(t_l) + \mathsf{nodes}(t_r)$$

   So far, everything has worked much like before. Unfortunately, now our induction hypotheses are different in a very inconvenient way:

   $$\mathsf{leaves}(t_l) \boxed{\geq} \mathsf{nodes}(t_l) \qquad \text{and} \qquad \mathsf{leaves}(t_r) \boxed{\geq} \mathsf{nodes}(t_r)$$

   Unfortunately, these facts do **not** imply

   $$\mathsf{leaves}(t_l) + \mathsf{leaves}(t_r) \boxed{\geq} 1 + \mathsf{nodes}(t_l) + \mathsf{nodes}(t_r)$$

Because of the "1+" and the difference between "$\geq$" and "$>$". Assuming our induction hypothesis on the subparts of t was not strong enough to prove the induction hypothesis on t itself. Our proof is broken.

When an induction proof fails like this, your best bet is to *strengthen the induction hypothesis*. That is, you are trying to prove something of the form:

$$\forall x\ \mathsf{IH}(x)$$

You find some alternative induction hypothesis IH' such that

$$\forall x\ \left(\ \mathsf{IH}'(x)\ \Rightarrow\ \mathsf{IH}(x)\ \right)$$

Therefore, proving $\forall x\ \mathsf{IH}'(x)$ will directly let you prove the related $\forall x\ \mathsf{IH}(x)$. For example 7, we can use the induction hypothesis from example 5 for IH'.

When you try to prove $\forall x\ \mathsf{IH}'(x)$ by induction on the structure of x, you will discover that some parts of the proof are easier and some are harder. The harder part is that you have to prove a more difficult goal; examining the above two examples, in example 5 (successful), we had to prove a **strict** inequality ($x > y$); in example 7 (unsuccessful), we only had to only prove a **loose** inequality ($x \geq y$). Of course, a strict inequality is **harder** to prove than a loose one, all things equal, since when $x = y$ you can prove $x \geq y$ but not $x > y$.

Fortunately, not everything is equal when we strengthen the induction hypothesis. In particular, on the inductive branches (in examples 5 & 7, the second case) we get to use a more powerful induction hypothesis. Thus, in example 5 (successful), we were given strict inequalities as hypotheses, whereas in example 7 (unsuccessful), we were only given loose inequalities. As a premise, a strict inequality is better than a weak one since from $x > y$ you can rule out the case when $x = y$, which you cannot do from the weaker $x \geq y$.

The real question is, does the trade-off of harder goals for stronger induction hypotheses work out or not? In the case of example 7, it is clear that it does work out. However, in general it can be very hard to tell without doing the proof. In fact, finding a stronger induction hypothesis that strikes the right balance is one of the hardest things to do when trying to prove something.

Coq tactics for weakening. Notice from the above Coq examples how Coq used the goal to determine the induction hypothesis. In fact, the way that Coq chooses induction hypothesis is as follows: First it puts everything directly containing the variable you are inducting over under the bar, much like the `revert` tactic. Second, it uses the goal as the induction hypothesis and sets up the cases. If you do not get the induction hypothesis you were hoping to get (that is, you need a stronger hypothesis), you need to manipulate the goal until it has the right shape before using the `induction` tactic. Useful tactics for this process are `revert`, `generalize`, `clear`, `assert`, and others.

Quantifiers. One of the most common problems when doing an induction is that some quantified variables are not properly accounted for. Suppose you are

trying to prove a goal along the lines of the following:

$$\forall x\ \forall y\ \big(R(x) \Rightarrow P(x,y)\big)$$

Now, if you are doing induction on y, you have two separate choices for what to do induction on. Choice one is:

$$x, R(x)\ \vdash\ \mathsf{IH}(y)\ =\ P(x,y)$$

By "$y, R(y)$" to the left of the "$\vdash$", I mean that x and $R(x)$ have already been been introduced via universal introduction and implication introduction, before setting up the induction. This is in fact **exactly** what you will get if you start a Coq proof for the goal using "`intros. induction y.`", or even just "`induction y.`", which will do the "`intros until y.`" for you. In fact, it will work fine—as long as x does not need to change values during the course of the proof. If it does, though, then you are out of luck with that induction hypothesis, and you need to strengthen it to:

$$\mathsf{IH}(y)\ =\ \forall x\big(R(x) \Rightarrow P(x,y)\big)$$

Of course, this hypothesis is much stronger since it is usable for any x you want, instead of one particular x. When you write your induction hypotheses on paper, be very careful to make sure you are putting the quantifiers in the right places. Incidentally, the way to transform the goal into the form where you get this induction hypothesis is "`intros x y. revert x. induction y.`". Another way, which works quite well and is more general, is to use an `assert`.