Induction

CS 3234: Logic and Formal Systems

Martin Henz and Aquinas Hobor

August 19, 2010

Generated on Tuesday 24 August, 2010, 22:51

1 Motivating Examples

It is common in the study of formal systems to define a set by a collection of rules that specify the members of the set. Each rule has zero or more premises, or requirements, and one conclusion.

Example 1 (Numerals, first attempt)**.** *We may define the set of unary numerals (i.e., numerals in base 1) for the natural numbers as follows:*

- `Zero` *is a numeral.*
- *If* n *is a numeral, then* $\texttt{Succ}(n)$ *is also a numeral.*

Equivalently, we might say that the set `Num` *of numerals is defined by the following rules:*

- $\texttt{Zero} \in \texttt{Num}$*.*
- *If* $n \in \texttt{Num}$*, then* $\texttt{Succ}(n) \in \texttt{Num}$*.*

Observe that in each formulation the first rule has no premises, whereas the second has one. Examples for elements of `Num` *are:*
`Zero` *and* $\texttt{Succ}(\texttt{Succ}(\texttt{Zero}))$*.*

Example 2 (Binary trees, first attempt)**.** *We may define the set of binary trees by the following rules:*

- *The empty tree,* `Empty`*, is a binary tree.*
- *If* t_l *and* t_r *are binary trees, then* $\texttt{Node}(t_l, t_r)$ *is a binary tree.*

Equivalently, we might say that the set `Tree` *of binary trees is defined by the following rules:*

- $\texttt{Empty} \in \texttt{Tree}$*.*

- *If* $t_l, t_r \in$ *Tree, then* $\mathit{Node}(t_l, t_r) \in$ *Tree.*

The first rule has no premises; the second has two. Examples for elements of T are:
Empty and $\mathit{Node}(\mathit{Empty}, \mathit{Node}(\mathit{Node}(\mathit{Empty}, \mathit{Empty}), \mathit{Empty}))$.

Notice the similarity between these two examples. The empty tree is analogous to the number 0, and the node formation operation is analogous to the successor operation (except that it has two predecessors!).

Excursion: Defining Sets by Rules in Java and Coq

We can directly translate the concept of defining sets by rules into Java. Our examples translated to Java look like this:

```
interface Num {}
class Zero implements Num {}
class Succ implements Num {
   public Num pred;
   Succ(Num p) {pred = p;}
}

interface Tree {}
class Empty implements Tree {}
class Node implements Tree {
   public Tree left, right;
   Node(Tree l,Tree r) {left = l; right = r;}
}
```

These class definitions introduce the types `Num` and `Tree`, respectively, from a given set of constructors. Each constructor defines a rule for membership in that type. The (implicitly defined) constructors `Zero()` and `Empty()` have no arguments; this corresponds to a rule with no premises. The constructor `Succ` has one argument, corresponding to the single premise in the inductive definition; the constructor `Node` has two arguments, corresponding to the two premises in the inductive definition. We can construct the example instances of type `Num` by

```
Num my_num = new Zero();
Num my_other_num = new Succ(new Succ(new Zero()));
```

and the example instances of type `Tree` by

```
Tree my_tree = new Empty();
Tree my_other_tree =
   new Node(new Empty,
```

```
new Node(new Node(new Empty(),
                  new Empty()),
         new Empty()));
```

Exercise 1. *Give a collection of rules defining the set of strings over characters. Give the corresponding Java class definitions (without using Java's* `string` *class) for the same set.*

In Coq, we can define the type `Nat` containing all natural numbers using the keyword `Inductive` as follows.

```
Inductive Nat : Type :=
  | Zero : Nat
  | Succ : Nat -> Nat.
```

Of course, we are free to use Coq variables such as `One`, `Two`, and `Three` to refer to particular natural numbers.

```
Definition One : Nat := Succ Zero.
Definition Two : Nat := Succ One.
Definition Three : Nat := Succ Two.
```

Similarly, we define the type `Tree` containing all binary trees.

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

Definition exampleTree : Tree :=
  Node (Node Empty Empty) Empty.
```

2 The Extremal Clause

When we say that a set is defined by a set of rules, what precisely do we mean? Which set do we consider to be defined by those rules? To see why this is an important question, consider the set

$$\begin{aligned} \mathit{YourNum} \quad = \quad & \{\texttt{Zero}, \texttt{Succ}(\texttt{Zero}), \texttt{Succ}(\texttt{Succ}(\texttt{Zero})), \ldots\} \\ & \cup \{\infty, \texttt{Succ}(\infty), \texttt{Succ}(\texttt{Succ}(\infty)), \ldots\} \end{aligned}$$

where ∞ is an arbitrary new symbol. Observe that $\texttt{Zero} \in \mathit{YourNum}$; and that if $n \in \mathit{YourNum}$, then $\texttt{Succ}(n) \in \mathit{YourNum}$—that is, *YourNum* meets the requirements of the rules we gave to define the set `Num`. This means that the

rules alone are not sufficient to pick out the intended set `Num`, since the strictly bigger set[1] *YourNum* also satisfies these same rules.

To use a set of rules to define a set, we must say something more than just that the set must obey these rules. What more is needed? We need an extremal clause that states that *nothing else is in the set except those elements that are required to be there by the rules.* This may sound like a bit of legalese, but mathematically it is essential to include the extremal clause, for otherwise the rules do not determine a unique set. Thus, the definition of *Num* should really be stated as follows.

Example 3 (Numerals, revised)**.** *The set Num is defined by the following rules:*

- *`Zero` is a numeral.*
- *If n is a numeral, then `Succ`(n) is also a numeral.*
- *nothing else is a numeral.*

Equivalently, we may say that `Num` is the least set that fulfills the first two rules, by which we mean precisely that nothing else is in the set except as is forced by the rules. Here "least" refers to the subset relation over sets; if X is the least set that fulfills some rules, then for any set Y that fulfills the rules, we have $X \subseteq Y$.

To see that YourNum is ruled out by the extremal clause, observe that ∞ has no business being in the specified set because it is not forced to be in there by the rules. Observe that YourNum is not the least set that fulfills the first two rules because YourNum $\supsetneq$ `Num` and `Num` obeys these rules. Thus YourNum is not defined by the rules.

Similarly, we may revise the definition of binary trees by adding an extremal clause as follows.

Example 4 (Binary trees, revised)**.** *We may define the set of binary trees by the following rules:*

- *The empty tree, `Empty` is a binary tree.*
- *If t_l and t_r are binary trees, then `Node`(t_l, t_r) is a binary tree.*
- *Nothing else is a binary tree.*

Equivalently, we might say that the set `Tree` of binary trees is the least set defined by the following two rules

- *`Empty` $\in$ `Tree`.*
- *If $t_l, t_r \in$ `Tree`, then `Node`$(t_l, t_r) \in$ `Tree`.*

[1] bigger in the sense that *YourNum* is a strict superset of `Num`

The extremal clause ensures that a collection of rules of the kind given above determines a unique set. In practice we do not explicitly state the extremal clause, but rather we state that the set in question is inductively defined by a given collection of rules. For example, we may say that the set `Num` is inductively defined by the two rules membership rules given in Example 1. In doing so we are implicitly stating that nothing else is to be a member of that set unless it is *forced* to be there by the rules.

3 Inductive Definition with Inference Rules

It is quite common to give an inductive definition by a set of inference rules. For example, we might say that the set `Num` is inductively defined by the following rules:

$$\frac{}{Zero} \qquad \frac{n}{Succ(n)}$$

Similarly, we might say that the set `Tree` is inductively defined by the following rules:

$$\frac{}{Empty} \qquad \frac{t_l \quad t_r}{Node(t_l, t_r)}$$

The horizontal line plays the role of "if ... then ..." in our earlier presentations of the rules. In general, in an inductive definition of a set X, an inference rule of the form

$$\frac{x_1 \quad \cdots \quad x_n}{x}$$

stands for the rule "if $x_1 \ldots x_n \in X$, then $x \in X$." Why do we say that the least set defined by a collection of rules is *inductively* defined by it? As the terminology suggests, the answer is that there is a close connection with reasoning by mathematical induction. Here's why.

Suppose that we wish to prove that every binary tree t has a size s satisfying the following two requirements:

- The size of `Empty` is 1.
- If t_l and t_r have sizes s_l and s_r, respectively, then the size of `Node`(t_l, t_r) is $1 + s_l + s_r$.

We call these two conditions the specification of the size of a binary tree. The question is this: how do we know that every binary tree in fact has a size? This might seem like an odd question at first, but consider that the *infinite* binary tree

`Node`(`Node`(...,...), `Node`(...,...)) has no size in the sense specified! Luckily, the extremal clause rules out such "infinite trees", which makes it possible to assign a size to each binary tree.

How do we prove that every binary tree has a size? By induction! We have to prove that for every binary tree $t \in$ `Tree`, there exists a number s satisfying the specification of size given above. Given the inductive definition of binary trees, what might t be? By the first rule defining binary trees, t might be `Empty`. In that case, t clearly has a size, namely 1, in accordance with the specification. By the second rule defining binary trees, t might have the form `Node`(t_l, t_r), where t_l and t_r are also binary trees. By induction we may assume that each of them has a size, say s_l and s_r, respectively. But then the size s of t is uniquely determined by the equation $s = 1 + \mathit{size}(t_l) + \mathit{size}(t_r)$ as required by the specification. Since `Tree` contains no other elements than are given by these two rules, we have demonstrated that *every* binary tree $t \in$ `Tree` has a size s.

Size of Trees in Java and Coq

Here is another point of view on the same question. Let look at the definition of the `size` function for the Java type `Tree`.

```
interface Tree {
   public int size();
}
class Empty implements Tree {
   public int size() {return 1;}
}
class Node implements Tree {
   public Tree left, right;
   Node(Tree l,Tree r) {left = l; right = r;}
   public int size() {
      return 1 + size(left) + size(right);
   }
}
```

The question is: why does the call `size(t)` terminate for every `t` of type `Tree`? Once again, the proof is by induction on the structure of `t`. If `t` is an instance of `Empty`, then `size(t)` terminates returning 1, as required. If, on the other hand, `t` is an instance of `Node`, then inductively we may assume that `size(left)` terminates (returning s_l) and that `size(right)` terminates (returning s_r), from which it follows that `size(t)` terminates with `1 + size(left) + size(right)`. This completes the proof.

> In Coq, we would like to define the function `Size` that takes a `Tree` t as argument and returns a `Nat` which represents the size of t. We first need to define as function `Add` which adds two given `Nats`, using the keyword `Fixpoint`.

```
Fixpoint Add a b :=
  match a with
   | Zero => b
   | Succ a' => Succ (Add a' b)
  end.
```

Now we are ready to define `Size`.

```
Fixpoint Size t :=
   match t with
   | Empty => Zero
   | Node t1 t2 => Add One (Add (Size t1) (Size t2))
   end.
```

Note that the Coq environment automatically proves the termination of functions that are defined using `Fixpoint`.

4 Defining Sets in Stages

Here is another perspective that may help guide your intuition. Since the smallest set inductively defined by a set of rules contains only those elements explicitly required by the rules, we might envision elements of this set as being built up in "stages". At the first stage we introduce elements that are forced to be in the set by a rule with no premises. (For example, `Zero` and `Empty` would appear at the first stage of their respective definitions, since the rules introducing them have no premises.) Now suppose that we have completed stage s of the construction. At stage $s+1$ we add in those elements that are introduced by a rule whose premises are satisfied by elements of stage s. (For example, if n is a numeral at stage s, then $\texttt{Succ}(n)$ is a numeral at stage $s+1$, and if both t_l and and t_r are trees at stage s, then $\texttt{Node}(t_l, t_r)$ is a tree at stage $s+1$.)

We may then think of the set inductively defined by these rules as the set of elements that appear at some stage s in the construction. Arguments by induction on the definition of an element of an inductively-defined set may be replaced by arguments by induction on the stage at which the element is introduced. (For example, at stage 0 we have only the empty tree, whose size is 0. At stage $s+1$, we add in those nodes whose children are trees at stage s. By the inductive hypothesis the children have a size, so the node must also have a size. The proof is complete because we have considered all possible stages s.)

It is important to notice the close interplay between recursive function definitions and inductive proofs illustrated here. Proving by induction and programming by recursion are two sides of the same coin. The concept of an invariant specification for a recursive function (describing the input/output behavior of

that function) corresponds exactly to the idea of an inductive hypothesis for an inductive argument. Cases in a recursive function that do not make recursive calls correspond to base cases of inductive proofs. Proving a statement of the form "for all x there exists y such that ..." by induction on x corresponds to writing a recursive function that recurs on x and computes the value y with the required properties.