

## Inductively defined sets, and recursively defined functions

Recall: An inductive definition of a set  $S$  has the following form:

- ▶ *Basis*: Specify one or more “initial” elements of  $S$ .
- ▶ *Induction*: Give one or more rules for constructing “new” elements of  $S$  from “old” elements of  $S$ .

A recursive definition of a function  $f : S \rightarrow A$  sometimes follows the form of an inductive definition of its domain  $S$ :

- ▶ *Basis*: For each “initial” element  $x$  of  $S$ , define  $f(x)$ .
- ▶ *Induction*: For each rule constructing a “new” element  $x$  of  $S$  from “old” elements  $x_1, \dots, x_k$  of  $S$ , define  $f(x)$  in terms of  $f(x_1), \dots, f(x_k)$ .

For instance, given the inductive definition of  $\mathcal{N}$ , it is not surprising that recursively defined functions with domain  $\mathcal{N}$  are often specified as follows:

- ▶ *Basis*: Define  $f(0)$ .
- ▶ *Induction*: Define  $f(n+1)$  in terms of  $f(n)$ , for all  $n \in \mathcal{N}$ .

Example: Take  $f : \mathcal{N} \rightarrow \mathcal{N}$  s.t.

$$\begin{aligned}f(0) &= 0 \\f(n+1) &= f(n) + n + 1\end{aligned}$$

So

$$\begin{aligned}f(4) &= f(3) + 4 \\&= f(2) + 3 + 4 \\&= f(1) + 2 + 3 + 4 \\&= f(0) + 1 + 2 + 3 + 4 \\&= 0 + 1 + 2 + 3 + 4\end{aligned}$$

Other ways to write this recursive definition:

$$f(n) = \begin{cases} 0 & , \text{ if } n = 0 \\ f(n-1) + n & , \text{ otherwise} \end{cases}$$

$$f(n) = \text{if } n = 0 \text{ then } 0 \text{ else } f(n-1) + n$$

A rather different example of a recursively defined function with domain  $\mathcal{N}$ :

$$\begin{aligned}f(0) &= 0 \\f(1) &= 1 \\f(n+2) &= f(n) + f(n+1)\end{aligned}$$

This one doesn't correspond so nicely to the usual inductive definition of  $\mathcal{N}$ .

Instead, this definition says *directly* what  $f$  does on 0 *and* on 1. And then for successive numbers greater than 1, it (recursively) uses the value of  $f$  on the prior *two* numbers.

Roughly speaking, what's crucial here is that we have a place to start — here, the base cases 0 and 1 — and the values for all other numbers are defined recursively in terms of the values for “prior” numbers.

Technically, what's necessary in such a definition is just that there be a *unique* function that satisfies the conditions of the definition.

But such a definition also suggests a recursive algorithm for computing the value of the function for a given input. (In fact, such a definition can be understood as a recursive program.)

As you know, such recursive definitions of functions can be used with inductively defined domains other than  $\mathcal{N}$ ...

Recall:  $\{0, 1\}^*$  can be defined inductively as follows:

- ▶ *Basis:*  $\Lambda \in \{0, 1\}^*$ .
- ▶ *Induction:* If  $s \in \{0, 1\}^*$  and  $x \in \{0, 1\}$ , then  $xs \in \{0, 1\}^*$ .

Take  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  s.t.

$$\begin{aligned}f(\Lambda) &= \Lambda \\f(0s) &= 1f(s) \\f(1s) &= 0f(s)\end{aligned}$$

So, for example,

$$\begin{aligned}f(010) &= 1f(10) \\&= 10f(0) \\&= 101f(\Lambda) \\&= 101\end{aligned}$$

How about defining a function  $g : \{0, 1\}^* \rightarrow \{0, 1\}^*$  that reverses its input?

Here's an inductive definition of  $\text{lists}(A)$ :

- ▶ *Basis*:  $\langle \rangle \in \text{lists}(A)$ .
- ▶ *Induction*: If  $x \in A$  and  $L \in \text{lists}(A)$ , then  $\text{cons}(x, L) \in \text{lists}(A)$ .

We'll also introduce convenient alternative notation: Instead of

$$\text{cons}(x, L)$$

we will sometimes write

$$x :: L.$$

Here's a recursive definition of a function from  $\mathcal{N} \times \text{lists}(\mathcal{N})$  to  $\text{lists}(\mathcal{N})$ :

$$\begin{aligned} \text{insert}(x, \langle \rangle) &= \langle x \rangle \\ \text{insert}(x, y :: L) &= \begin{cases} x :: y :: L & , \text{ if } x \leq y \\ y :: \text{insert}(x, L) & , \text{ otherwise} \end{cases} \end{aligned}$$

Here's a more interesting recursively-defined function, from  $\text{lists}(\mathcal{N})$  to  $\text{lists}(\mathcal{N})$ :

$$\begin{aligned} \text{sort}(\langle \rangle) &= \langle \rangle \\ \text{sort}(x :: L) &= \text{insert}(x, \text{sort}(L)) \end{aligned}$$

What would it be like to prove that these function definitions are “correct”?

Consider:

$$\begin{aligned} \text{insert}(x, \langle \rangle) &= \langle x \rangle \\ \text{insert}(x, y :: L) &= \begin{cases} x :: y :: L & , \text{ if } x \leq y \\ y :: \text{insert}(x, L) & , \text{ otherwise} \end{cases} \end{aligned}$$

Well, what is “correct” behavior here?

The argument would be “inductive”, following the shape of the definition of the function...

First we would argue that the base case is “correct”.

Then we would argue the recursive case is correct for any list  $y :: L$  over  $\mathcal{N}$ , **assuming** that it is correct for the corresponding “prior” list  $L$ .

Now consider the same questions for

$$\begin{aligned}\text{sort}(\langle \rangle) &= \langle \rangle \\ \text{sort}(x :: L) &= \text{insert}(x, \text{sort}(L))\end{aligned}$$

Such functions can be remarkably easy to write *and* prove correct.

Here's the idea for an interesting function. It takes a list of elements from  $A$  and applies to each element a function from  $A$  to  $B$ , returning the resulting list of elements from  $B$ .

$$\text{map} : \text{lists}(A) \times (A \rightarrow B) \rightarrow \text{lists}(B)$$

So what is the base case like? And can we write it?

And what is the recursive case like? And can we write that?

And how hard would it be to “prove” that the function definition is “correct”?

Now let's recursively define a function from  $\text{lists}(A)$  to  $\text{lists}(A)$  that removes duplicate elements from its input.

Here is a useful auxiliary function from  $A \times \text{lists}(A)$  to  $\text{lists}(A)$ :

$$\begin{aligned} \text{removeAll}(x, \langle \rangle) &= \langle \rangle \\ \text{removeAll}(x, y :: L) &= \begin{cases} y :: \text{removeAll}(x, L) & , \text{ if } x \neq y \\ \text{removeAll}(x, L) & , \text{ otherwise} \end{cases} \end{aligned}$$

Now let's write a definition for

$$\text{removeDuplicates} : \text{lists}(A) \rightarrow \text{lists}(A)$$

$$\text{removeDuplicates}(\langle \rangle) =$$

$$\text{removeDuplicates}(x :: L) =$$

Recall: We can define the set  $BT$  of *binary trees* over alphabet  $A$  as follows:

- ▶ *Basis*:  $\langle \rangle \in BT$ .
- ▶ *Induction*: If  $L, R \in BT$  and  $x \in A$ , then  $\langle L, x, R \rangle \in BT$ .

Let's define a function from  $BT$  to  $\mathcal{N}$  that counts the nodes in a binary tree:

$$\text{nodes}(\langle \rangle) =$$

$$\text{nodes}(\langle L, x, R \rangle) =$$

Now let's define a function from  $BT$  to  $\mathcal{Z}$  that returns the "depth" of a tree.

Depth is typically understood to mean the length of the longest path from the root to a leaf. By convention, if the tree is  $\langle \rangle$  (so it has no root and no leaves), its depth is  $-1$ .

$$\text{depth}(\langle \rangle) =$$

$$\text{depth}(\langle L, x, R \rangle) =$$

Now let's consider binary trees over  $\mathcal{Z}$ .

Let's write a function to compute the sum of the nodes of the tree.

To return the maximum node in a nonempty tree.