Programming Languages
3. Definition and Proof by Induction

Shin-Cheng Mu

Oct. 17, 2019

Total Functional Programming

- The next few lectures concerns inductive definitions and proofs of datatypes and programs.
- While Haskell provides allows one to define non-terminating functions, infinite data structures, for now we will only consider its total, finite fragment.
- That is, we temporarily
  - consider only finite data structures,
  - demand that functions terminate for all value in its input type, and
  - provide guidelines to construct such functions.
- Infinite datatypes and non-termination will be discussed later in this course.

1 Induction on Natural Numbers

The So-Called "Mathematical Induction"

- Let P be a predicate on natural numbers.
  - What is a predicate? Such a predicate can be seen as a function of type $Nat \to Bool$.
  - So far, we see Haskell functions as simple mathematical functions too.
  - However, Haskell functions will turn out to be more complex than mere mathematical functions later. To avoid confusion, we do not use the notation $Nat \to Bool$ for predicates.
- We've all learnt this principle of proof by induction: to prove that P holds for all natural numbers, it is sufficient to show that
  - $P\,0$ holds;
  - $P\;(1+n)$ holds provided that $P\,n$ does.

1.1 Proof by Induction

Proof by Induction on Natural Numbers

- We can see the above inductive principle as a result of seeing natural numbers as defined by the datatype [1]

$$\mathbf{data}\ Nat\ =\ 0 \mid \mathbf{1}_{+}\ Nat\ \ .$$

- That is, any natural number is either 0, or $\mathbf{1}_{+}\ n$ where n is a natural number.
- In this lecture, $\mathbf{1}_{+}$ is written in bold font to emphasise that it is a data constructor (as opposed to the function $(+)$, to be defined later, applied to a number 1).

A Proof Generator

Given $P\,0$ and $P\,n \Rightarrow P\ (\mathbf{1}_{+}\ n)$, how does one prove, for example, $P\ 3$?

$$\begin{array}{cl}
 & P\ (\mathbf{1}_{+}\ (\mathbf{1}_{+}\ (\mathbf{1}_{+}\ 0))) \\
\Leftarrow & \quad \{\ P\ (\mathbf{1}_{+}\ n) \Leftarrow P\,n\ \ \} \\
 & P\ (\mathbf{1}_{+}\ (\mathbf{1}_{+}\ 0)) \\
\Leftarrow & \quad \{\ P\ (\mathbf{1}_{+}\ n) \Leftarrow P\,n\ \ \} \\
 & P\ (\mathbf{1}_{+}\ 0) \\
\Leftarrow & \quad \{\ P\ (\mathbf{1}_{+}\ n) \Leftarrow P\,n\ \ \} \\
 & P\,0\ .
\end{array}$$

Having done math. induction can be seen as having designed *a program that generates a proof* — given any $n :: Nat$ we can generate a proof of $P\,n$ in the manner above.

[1] Not a real Haskell definition.

1.2 Inductively Definition of Functions

Inductively Defined Functions

- Since the type *Nat* is defined by two cases, it is natural to define functions on *Nat* following the structure:

$$\begin{array}{ll} exp & :: Nat \to Nat \to Nat \\ exp\ b\ 0 & = 1 \\ exp\ b\ (\mathbf{1}_+\ n) & = b \times exp\ b\ n \quad . \end{array}$$

- Even addition can be defined inductively

$$\begin{array}{ll} (+) & :: Nat \to Nat \to Nat \\ 0 + n & = n \\ (\mathbf{1}_+\ m) + n & = \mathbf{1}_+\ (m + n) \quad . \end{array}$$

- Exercise: define $(\times)$?

A Value Generator

Given the definition of *exp*, how does one compute $exp\ b\ 3$?

$$\begin{array}{ll} & exp\ b\ (\mathbf{1}_+\ (\mathbf{1}_+\ (\mathbf{1}_+\ 0))) \\ = & \quad \{\ \text{definition of } exp\ \} \\ & b \times exp\ b\ (\mathbf{1}_+\ (\mathbf{1}_+\ 0)) \\ = & \quad \{\ \text{definition of } exp\ \} \\ & b \times b \times exp\ b\ (\mathbf{1}_+\ 0) \\ = & \quad \{\ \text{definition of } exp\ \} \\ & b \times b \times b \times exp\ b\ 0 \\ = & \quad \{\ \text{definition of } exp\ \} \\ & b \times b \times b \times 1 \quad . \end{array}$$

It is a program that generates a value, for any $n :: Nat$. Compare with the proof of P above.

Moral: Proving is Programming

An inductive proof is a program that generates a proof for any given natural number.

An inductive program follows the same structure of an inductive proof.

Proving and programming are very similar activities.

Without the $n + k$ Pattern

- Unfortunately, newer versions of Haskell abandoned the "$n + k$ pattern" used in the previous slides:

$$\begin{array}{ll} exp & :: Int \to Int \to Int \\ exp\ b\ 0 & = 1 \\ exp\ b\ n & = b \times exp\ b\ (n - 1) \quad . \end{array}$$

- *Nat* is defined to be *Int* in `MiniPrelude.hs`. Without `MiniPrelude.hs` you should use *Int*.
- For the purpose of this course, the pattern $1 + n$ reveals the correspondence between *Nat* and lists, and matches our proof style. Thus we will use it in the lecture.
- Remember to remove them in your code.

Proof by Induction

- To prove properties about *Nat*, we follow the structure as well.
- E.g. to prove that $exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n$.
- One possibility is to preform induction on m. That is, prove $P\,m$ for all $m :: Nat$, where $P\,m \equiv (\forall n :: exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n)$.

Case $m := 0$. For all n, we reason:

$$\begin{array}{ll} & exp\ b\ (0 + n) \\ = & \quad \{\ \text{defn. of } (+)\ \} \\ & exp\ b\ n \\ = & \quad \{\ \text{defn. of } (\times)\ \} \\ & 1 \times exp\ b\ n \\ = & \quad \{\ \text{defn. of } exp\ \} \\ & exp\ b\ 0 \times exp\ b\ n \quad . \end{array}$$

We have thus proved $P\,0$.

Case $m := \mathbf{1}_+\ m$. For all n, we reason:

$$\begin{array}{ll} & exp\ b\ ((\mathbf{1}_+\ m) + n) \\ = & \quad \{\ \text{defn. of } (+)\ \} \\ & exp\ b\ (\mathbf{1}_+\ (m + n)) \\ = & \quad \{\ \text{defn. of } exp\ \} \\ & b \times exp\ b\ (m + n) \\ = & \quad \{\ \text{induction}\ \} \\ & b \times (exp\ b\ m \times exp\ b\ n) \\ = & \quad \{\ (\times)\ \text{associative}\ \} \\ & (b \times exp\ b\ m) \times exp\ b\ n \\ = & \quad \{\ \text{defn. of } exp\ \} \\ & exp\ b\ (\mathbf{1}_+\ m) \times exp\ b\ n \quad . \end{array}$$

We have thus proved $P\ (\mathbf{1}_+\ m)$, given $P\ m$.

Structure Proofs by Programs

- The inductive proof could be carried out smoothly, because both $(+)$ and *exp* are defined inductively on its lefthand argument (of type *Nat*).

- The structure of the proof follows the structure of the program, which in turns follows the structure of the datatype the program is defined on.

Lists and Natural Numbers

- We have yet to prove that $(\times)$ is associative.
- The proof is quite similar to the proof for associativity of $(+)$, which we will talk about later.
- In fact, *Nat* and lists are closely related in structure.
- Most of us are used to think of numbers as atomic and lists as structured data. Neither is necessarily true.
- For the rest of the course we will demonstrate induction using lists, while taking the properties for *Nat* as given.

1.3 A Set-Theoretic Explanation of Induction

An Inductively Defined Set?

- For a set to be "inductively defined", we usually mean that it is the *smallest* fixed-point of some function.
- What does that maen?

Fixed-Point and Prefixed-Point

- A *fixed-point* of a function f is a value x such that $f\,x = x$.
- **Theorem**. f has fixed-point(s) if f is a *monotonic function* defined on a complete lattice.
  - In general, given f there may be more than one fixed-point.
- A *prefixed-point* of f is a value x such that $f\,x \leq x$.
  - Apparently, all fixed-points are also prefixed-points.
- **Theorem**. the smallest prefixed-point is also the smallest fixed-point.

Example: *Nat*

- Recall the usual definition: *Nat* is defined by the following rules:
  1. 0 is in *Nat*;
  2. if n is in *Nat*, so is $\mathbf{1}_+\ n$;
  3. there is no other *Nat*.
- If we define a function F from sets to sets: $F\,X = \{0\} \cup \{\mathbf{1}_+\ n \mid n \in X\}$, 1. and 2. above means that $F\,\mathit{Nat} \subseteq \mathit{Nat}$. That is, *Nat* is a prefixed-point of F.
- 3. means that we want the *smallest* such prefixed-point.
- Thus *Nat* is also the least (smallest) fixed-point of F.

Least Prefixed-Point

Formally, let $F\,X = \{0\} \cup \{\mathbf{1}_+\ n \mid n \in X\}$, *Nat* is a set such that

$$F\,\mathit{Nat} \subseteq \mathit{Nat} \quad , \tag{1}$$

$$(\forall X : F\,X \subseteq X \;\Rightarrow\; \mathit{Nat} \subseteq X) \quad , \tag{2}$$

where (1) says that *Nat* is a prefixed-point of F, and (2) it is the least among all prefixed-points of F.

Mathematical Induction, Formally

- Given property P, we also denote by P the set of elements that satisfy P.
- That $P\,0$ and $P\,n \Rightarrow P\ (\mathbf{1}_+ n)$ is equivalent to $\{0\} \subseteq P$ and $\{\mathbf{1}_+\ n \mid n \in P\} \subseteq P$,
- which is equivalent to $F\,P \subseteq P$. That is, P is a prefixed-point!
- By (2) we have $\mathit{Nat} \subseteq P$. That is, all *Nat* satisfy P!
- This is "why mathematical induction is correct."

Coinduction?

There is a dual technique called *coinduction* where, instead of least prefixed-points, we talk about *greatest postfixed points*. That is, largest x such that $x \leq f\,x$.

With such construction we can talk about infinite data structures.

2 Induction on Lists

Inductively Defined Lists

- Recall that a (finite) list can be seen as a datatype defined by: [2]

$$\mathbf{data}\ \mathit{List}\ a\ =\ [\,]\mid a : \mathit{List}\ a\ .$$

- Every list is built from the base case $[\,]$, with elements added by $(:)$ one by one: $[1,2,3] = 1 : (2 : (3 : [\,]))$.

All Lists Today are Finite

But what about infinite lists?

- For now let's consider finite lists only, as having infinite lists make the *semantics* much more complicated. [3]

- In fact, all functions we talk about today are total functions. No $\perp$ involved.

Set-Theoretically Speaking...

The type *List a* is the *smallest* set such that

1. $[\,]$ is in $\mathit{List}\ a$;

2. if xs is in $\mathit{List}\ a$ and x is in a, $x : xs$ is in $\mathit{List}\ a$ as well.

Inductively Defined Functions on Lists

- Many functions on lists can be defined according to how a list is defined:

$$\begin{array}{ll}
\mathit{sum} & :: \mathit{List}\ \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{sum}\ [\,] & = 0 \\
\mathit{sum}\ (x : xs) & = x + \mathit{sum}\ xs\ .
\end{array}$$

$$\begin{array}{ll}
\mathit{map} & :: (a \rightarrow b) \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ b \\
\mathit{map}\ f\ [\,] & = [\,] \\
\mathit{map}\ f\ (x : xs) & = F\ X : \mathit{map}\ f\ xs\ .
\end{array}$$

  - $\mathit{sum}\ [1..10] = 55$
  - $\mathit{map}\ (\mathbf{1}_{+})\ [1,2,3,4] = [2,3,4,5]$

[2]Not a real Haskell definition.

[3]What does that mean? We will talk about it later.

2.1 Append, and Some of Its Properties

List Append

- The function $(+\!\!+)$ appends two lists into one

$$\begin{array}{ll}
(+\!\!+) & :: \mathit{List}\ a \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a \\
[\,] +\!\!+ ys & = ys \\
(x : xs) +\!\!+ ys & = x : (xs +\!\!+ ys)\ .
\end{array}$$

- Compare the definition with that of $(+)$!

Proof by Structural Induction on Lists

- Recall that every finite list is built from the base case $[\,]$, with elements added by $(:)$ one by one.

- To prove that some property P holds for all finite lists, we show that

  1. $P\ [\,]$ holds;
  2. forall x and xs, $P\ (x : xs)$ holds provided that $P\ xs$ holds.

For a Particular List...

Given $P\ [\,]$ and $P\ xs \Rightarrow P\ (x : xs)$, for all x and xs, how does one prove, for example, $P\ [1,2,3]$?

$$\begin{array}{ll}
& P\ (1 : 2 : 3 : [\,]) \\
\Leftarrow & \quad \{\ P\ (x : xs) \Leftarrow P\ xs\ \} \\
& P\ (2 : 3 : [\,]) \\
\Leftarrow & \quad \{\ P\ (x : xs) \Leftarrow P\ xs\ \} \\
& P\ (3 : [\,]) \\
\Leftarrow & \quad \{\ P\ (x : xs) \Leftarrow P\ xs\ \} \\
& P\ [\,]\ .
\end{array}$$

Appending is Associative

To prove that $xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$.

Let $P\ xs = (\forall ys, zs :: xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs)$, we prove P by induction on xs.

Case $xs := [\,]$. For all ys and zs, we reason:

$$\begin{array}{ll}
& [\,] +\!\!+ (ys +\!\!+ zs) \\
= & \quad \{\ \text{defn. of } (+\!\!+)\ \} \\
& ys +\!\!+ zs \\
= & \quad \{\ \text{defn. of } (+\!\!+)\ \} \\
& ([\,] +\!\!+ ys) +\!\!+ zs\ .
\end{array}$$

We have thus proved $P\ [\,]$.

Case $xs := x : xs$. For all ys and zs, we reason:

$$
\begin{aligned}
& (x : xs) +\!\!+ (ys +\!\!+ zs) \\
= \quad & \{\ \text{defn. of } (+\!\!+)\ \} \\
& x : (xs +\!\!+ (ys +\!\!+ zs)) \\
= \quad & \{\ \text{induction}\ \} \\
& x : ((xs +\!\!+ ys) +\!\!+ zs) \\
= \quad & \{\ \text{defn. of } (+\!\!+)\ \} \\
& (x : (xs +\!\!+ ys)) +\!\!+ zs \\
= \quad & \{\ \text{defn. of } (+\!\!+)\ \} \\
& ((x : xs) +\!\!+ ys) +\!\!+ zs \quad .
\end{aligned}
$$

We have thus proved $P\ (x : xs)$, given $P\ xs$.

Do We Have To Be So Formal?

- In our style of proof, every step is given a reason. Do we need to be so pedantic?
- Being formal *helps* you to do the proof:
  - In the proof of $exp\ b\ (m + n) = exp\ b\ m \times exp\ b\ n$, we expect that we will use induction to somewhere. Therefore the first part of the proof is to generate $exp\ b\ (m + n)$.
  - In the proof of associativity, we were working toward generating $xs +\!\!+ (ys +\!\!+ zs)$.
- By being formal we can work on the *form*, not the *meaning*. Like how we solved the knight/knave problem
- Being formal actually makes the proof easier!
- *Make the symbols do the work.*

Length

- The function *length* defined inductively:

$$
\begin{aligned}
& length && :: List\ a \to Nat \\
& length\ [\,] && = 0 \\
& length\ (x : xs) && = \mathbf{1}_{+}\ (length\ xs) \quad .
\end{aligned}
$$

- Exercise: prove that *length* distributes into $(+\!\!+)$:

$$length\ (xs +\!\!+ ys) = length\ xs + length\ ys$$

Concatenation

- While $(+\!\!+)$ repeatedly applies $(:)$, the function *concat* repeatedly calls $(+\!\!+)$:

$$
\begin{aligned}
& concat && :: List\ (List\ a) \to List\ a \\
& concat\ [\,] && = [\,] \\
& concat\ (xs : xss) && = xs +\!\!+ concat\ xss \quad .
\end{aligned}
$$

- Compare with *sum*.
- Exercise: prove $sum \cdot concat = sum \cdot map\ sum$.

2.2 More Inductively Defined Functions

Definition by Induction/Recursion

- Rather than giving commands, in functional programming we specify values; instead of performing repeated actions, we define values on inductively defined structures.
- Thus induction (or in general, recursion) is the only "control structure" we have. (We do identify and abstract over plenty of patterns of recursion, though.)
- **Note** Terminology: an inductive definition, as we have seen, define "bigger" things in terms of "smaller" things. Recursion, on the other hand, is a more general term, meaning "to define one entity in terms of itself."
- To inductively define a function f on lists, we specify a value for the base case ($f\ [\,]$) and, assuming that $f\ xs$ has been computed, consider how to construct $f\ (x : xs)$ out of $f\ xs$.

Filter

- *filter p xs* keeps only those elements in *xs* that satisfy *p*.

$$
\begin{aligned}
& filter && :: (a \to Bool) \to List\ a \to List\ a \\
& filter\ p\ [\,] && = [\,] \\
& filter\ p\ (x : xs) && \mid p\ x = x : filter\ p\ xs \\
& && \mid \mathbf{otherwise} = filter\ p\ xs \quad .
\end{aligned}
$$

Take and Drop

- Recall *take* and *drop*, which we used in the previous exercise.

$$
\begin{aligned}
& take && :: Nat \to List\ a \to List\ a \\
& take\ 0\ xs && = [\,] \\
& take\ (\mathbf{1}_{+}\ n)\ [\,] && = [\,] \\
& take\ (\mathbf{1}_{+}\ n)\ (x : xs) && = x : take\ n\ xs \quad .
\end{aligned}
$$

$$
\begin{aligned}
& drop && :: Nat \to List\ a \to List\ a \\
& drop\ 0\ xs && = xs \\
& drop\ (\mathbf{1}_{+}\ n)\ [\,] && = [\,] \\
& drop\ (\mathbf{1}_{+}\ n)\ (x : xs) && = drop\ n\ xs \quad .
\end{aligned}
$$

- Prove: $take\ n\ xs +\!\!+ drop\ n\ xs = xs$, for all n and xs.

TakeWhile and DropWhile

- $takeWhile\ p\ xs$ yields the longest prefix of xs such that p holds for each element.

$$\begin{array}{ll}
takeWhile & :: (a \to Bool) \to List\ a \to List\ a \\
takeWhile\ p\ [\,] & = [\,] \\
takeWhile\ p\ (x : xs) & \mid p\ x = x : takeWhile\ p\ xs \\
 & \mid \mathbf{otherwise} = [\,] \quad .
\end{array}$$

- $dropWhile\ p\ xs$ drops the prefix from xs.

$$\begin{array}{ll}
dropWhile & :: (a \to Bool) \to List\ a \to List\ a \\
dropWhile\ p\ [\,] & = [\,] \\
dropWhile\ p\ (x : xs) & \mid p\ x = dropWhile\ p\ xs \\
 & \mid \mathbf{otherwise} = x : xs \quad .
\end{array}$$

- Prove: $takeWhile\ p\ xs +\!\!+\ dropWhile\ p\ xs = xs$.

List Reversal

- $reverse\ [1,2,3,4] = [4,3,2,1]$.

$$\begin{array}{ll}
reverse & :: List\ a \to List\ a \\
reverse\ [\,] & = [\,] \\
reverse\ (x : xs) & = reverse\ xs +\!\!+ [x] \quad .
\end{array}$$

All Prefixes and Suffixes

- $inits\ [1,2,3] = [[\,],[1],[1,2],[1,2,3]]$

$$\begin{array}{ll}
inits & :: List\ a \to List\ (List\ a) \\
inits\ [\,] & = [[\,]] \\
inits\ (x : xs) & = [\,] : map\ (x :)\ (inits\ xs) \quad .
\end{array}$$

- $tails\ [1,2,3] = [[1,2,3],[2,3],[3],[\,]]$

$$\begin{array}{ll}
tails & :: List\ a \to List\ (List\ a) \\
tails\ [\,] & = [[\,]] \\
tails\ (x : xs) & = (x : xs) : tails\ xs \quad .
\end{array}$$

Totality

- Structure of our definitions so far:

$$\begin{array}{ll}
f\ [\,] & = \ldots \\
f\ (x : xs) & = \ldots f\ xs \ldots
\end{array}$$

  - Both the empty and the non-empty cases are covered, guaranteeing there is a matching clause for all inputs.
  - The recursive call is made on a "smaller" argument, guranteeing termination.

- Together they guarantee that every input is mapped to some output. Thus they define *total* functions on lists.

2.3 Other Patterns of Induction

Variations with the Base Case

- Some functions discriminate between several base cases. E.g.

$$\begin{array}{ll}
fib & :: Nat \to Nat \\
fib\ 0 & = 0 \\
fib\ 1 & = 1 \\
fib\ (2 + n) & = fib\ (\mathbf{1}_{+} n) + fib\ n \quad .
\end{array}$$

- Some functions make more sense when it is defined only on non-empty lists:

$$\begin{array}{ll}
f\ [x] & = \ldots \\
f\ (x : xs) & = \ldots
\end{array}$$

- What about totality?
  - They are in fact functions defined on a different datatype:

$$\mathbf{data}\ List^{+}\ a\ =\ Singleton\ a \mid a : List^{+}\ a \quad .$$

  - We do not want to define *map*, *filter* again for $List^{+}\ a$. Thus we reuse $List\ a$ and pretend that we were talking about $List^{+}\ a$.
  - It's the same with *Nat*. We embedded *Nat* into *Int*.
  - Ideally we'd like to have some form of *subtyping*. But that makes the type system more complex.

Lexicographic Induction

- It also occurs often that we perform *lexicographic induction* on multiple arguments: some arguments decrease in size, while others stay the same.

- E.g. the function *merge* merges two sorted lists into one sorted list:

$$\begin{array}{ll}
merge & :: List\ Int \to List\ Int \to List\ Int \\
merge\ [\,]\ [\,] & = [\,] \\
merge\ [\,]\ (y : ys) & = y : ys \\
merge\ (x : xs)\ [\,] & = x : xs \\
merge\ (x : xs)\ (y : ys) & \mid x \leq y = x : merge\ xs\ (y : ys) \\
 & \mid \mathbf{otherwise} = y : merge\ (x : xs)\ ys \quad .
\end{array}$$

Zip

Another example:

$$
\begin{aligned}
&zip :: List\ a \to List\ b \to List\ (a,b)\\
&zip\ [\,]\ [\,] &&= [\,]\\
&zip\ [\,]\ (y:ys) &&= [\,]\\
&zip\ (x:xs)\ [\,] &&= [\,]\\
&zip\ (x:xs)\ (y:ys) &&= (x,y) : zip\ xs\ ys \quad .
\end{aligned}
$$

Non-Structural Induction

- In most of the programs we've seen so far, the recursive call are made on direct sub-components of the input (e.g. $f\ (x:xs) = ..f\ xs..$). This is called *structural induction.*
  - It is relatively easy for compilers to recognise structural induction and determine that a program terminates.
- In fact, we can be sure that a program terminates if the arguments get "smaller" under some (well-founded) ordering.

Mergesort

- In the implemenation of mergesort below, for example, the arguments always get smaller in size.

$$
\begin{aligned}
&msort &&:: List\ Int \to List\ Int\\
&msort\ [\,] &&= [\,]\\
&msort\ [x] &&= [x]\\
&msort\ xs &&= merge\ (msort\ ys)\ (msort\ zs) \quad ,\\
&\quad \mathbf{where}\ n = length\ xs\ \text{`}div\text{`}\ 2\\
&\qquad\qquad ys = take\ n\ xs\\
&\qquad\qquad zs = drop\ n\ xs \quad .
\end{aligned}
$$

  - What if we omit the case for $[x]$?
- If all cases are covered, and all recursive calls are applied to smaller arguments, the program defines a total function.

A Non-Terminating Definition

- Example of a function, where the argument to the recursive does not reduce in size:

$$
\begin{aligned}
&f &&:: Int \to Int\\
&f\ 0 &&= 0\\
&f\ n &&= f\ n \quad .
\end{aligned}
$$

- Certainly f is not a total function. Do such definitions "mean" something? We will talk about these later.

3 User Defined Inductive Datatypes

Internally Labelled Binary Trees

- This is a possible definition of internally labelled binary trees:

$$\mathbf{data}\ Tree\ a \;=\; \mathsf{Null} \mid \mathsf{Node}\ a\ (Tree\ a)\ (Tree\ a) \quad ,$$

- on which we may inductively define functions:

$$
\begin{aligned}
&sumT &&:: Tree\ Nat \to Nat\\
&sumT\ \mathsf{Null} &&= 0\\
&sumT\ (\mathsf{Node}\ x\ t\ u) &&= x + sumT\ t + sumT\ u \quad .
\end{aligned}
$$

Exercise: given $(\downarrow) :: Nat \to Nat \to Nat$, which yields the smaller one of its arguments, define the following functions

1. $minT :: Tree\ Nat \to Nat$, which computes the minimal element in a tree.
2. $mapT :: (a \to b) \to Tree\ a \to Tree\ b$, which applies the functional argument to each element in a tree.
3. Can you define $(\downarrow)$ inductively on Nat? [4]

Induction Principle for *Tree*

- What is the induction principle for *Tree*?
- To prove that a predicate P on *Tree* holds for every tree, it is sufficient to show that
  1. $P\ \mathsf{Null}$ holds, and;
  2. for every x, t, and u, if $P\ t$ and $P\ u$ holds, $P\ (\mathsf{Node}\ x\ t\ u)$ holds.
- Exercise: prove that for all n and t, $minT\ (mapT\ (n+)\ t) = n + minT\ t$. That is, $minT \cdot mapT\ (n+) = (n+) \cdot minT$.

Induction Principle for Other Types

- Recall that **data** $Bool = False \mid True$. Do we have an induction principle for *Bool*?
- To prove a predicate P on *Bool* holds for all booleans, it is sufficient to show that
  1. $P\ False$ holds, and

[4] In the standard Haskell library, $(\downarrow)$ is called *min*.

2. P *True* holds.

- Well, of course.
- What about $(A \times B)$? How to prove that a predicate P on $(A \times B)$ is always true?
- One may prove some property P_1 on A and some property P_2 on B, which together imply P.
- That does not say much. But the "induction principle" for products allows us to extract, from a proof of P, the proofs P_1 and P_2.
- *Every inductively defined datatype comes with its induction principle.*
- We will come back to this point later.