Structural Induction

COMP1600 / COMP6260

Dirk Pattinson Victor Rivera
Australian National University

Semester 2, 2021

Induction on Lists

Q. How do we *make* all finite lists?

A. All lists (over type A) can be obtained via the following:

- the empty list [] is a list (of elements of type A)
- given a list as and an element a (of type A), then prefixing as with a is a list (written $a : as$)

That is, lists are an *inductively* defined data type.

Q. How do we prove a property $P(l)$ for *all* lists l?

A. We (only) need to prove it for list constructed as above.

- establish that the property holds for [], i.e. $P([])$
- if as is a list for which $P(as)$ holds, and a is arbitrary, show that $P(a : as)$ holds.

Making and Proving in Lockstep

Suppose we want to establish that $P(as)$ holds for all lists as.

Stage 0. $as = []$.

- need to establish $P([])$.

Stage 1. $as = [a]$ has length 1.

- need to establish that $P([a])$
- already know that $P([])$ holds, may use this knowledge!

...

Stage $n+1$. $as = a : as'$ has length $n+1$

- need to establish that $P(a : as')$
- already know that $P(as')$ and may use this knowledge

May use the fact that $P(as)$ holds for lists constructed at previous stage

List Induction, Informally

To prove that $\forall as.P(as)$ it suffices to show

Base Case. $P([])$, i.e. P holds for the empty list

Step Case. $\forall a.\forall as.P(as) \to P(a : as)$

- assuming that $P(as)$ holds for all lists as (considered at previous stage)
- show that also $P(a : as)$ holds for an arbitrary element a

Example.

$$P([1,3,4,7]) \text{ follows from } P([3,4,7]) \text{ by step case}$$
$$P([3,4,7]) \text{ follows from } P([4,7]) \text{ by step case}$$
$$P([4,7]) \text{ follows from } P([7]) \text{ by step case}$$
$$P([7]) \text{ follows from } P([]) \text{ by step case}$$
$$P([]) \text{ holds by base case.}$$

Induction on Structure

List induction as a proof rule:

$$\frac{P([\,]) \qquad \forall x.\ \forall xs.\ P(xs) \rightarrow P(x : xs)}{\forall xs. P(xs)}$$

Annotated with types:

$$\frac{P([\,] :: [a]) \qquad \forall (x :: a).\ \forall (xs :: [a]).\ P(xs) \rightarrow P(x : xs)}{\forall (xs :: [a]).\ P(xs)}$$

Standard functions

Recall the following (standard library) function definitions:


```
length []       = 0                  -- (L1)
length (x:xs)   = 1 + length xs      -- (L2)
```



```
map f []        = []                 -- (M1)
map f (x:xs)    = f x : map f xs     -- (M2)
```



```
[] ++ ys        = ys                 -- (A1)
(x:xs) ++ ys    = x : (xs ++ ys)     -- (A2)
```

We read (and use) each line of the definition as *equation*.

Example. Mapping over Lists Preserves Length

Show. $\forall$`xs.length (map f xs) = length xs`.

Need to establish both premises of induction rule:

- $P([])$, and
- $\forall x.\forall xs.P(xs) \to P(x : xs)$.

Base Case: $P([\,])$


```
length (map f [])     = length []
```

Both sides are equal by `M1`: `map f [] = []`.

Step Case: $\forall x.\ \forall xs.\ P(xs) \to P(x : xs)$

Induction Hypothesis. Assume for an arbitrary list *as* that

```
length (map f as)        = length as       -- (IH)
```


Proof Goal. For arbitrary *a*, now prove that $P(a : as)$, i.e.

```
length (map f (a:as))    = length (a:as)
```



```
length (map f (a:as))
    = length (f a : map f as) -- by (M2)
    = 1 + length (map f as)   -- by (L2)
    = 1 + length as           -- by (IH)
    = length (a:as)           -- by (L2)
```


Formally (using $\to I$ and $\forall I$)

- this gives $P(as) \to P(a : as)$
- as both *a* and *as* were arbitrary, have $\forall x. \forall xs. P(xs) \to P(x : xs)$

In terms of Natural Deduction

Fixing arbitrary a and as and assuming $P(as)$, we show $P(a : as)$. That is, we reason as follows:

1	a	as	$P(as)$	
			$\vdots$	
6			$P(a : as)$	
7			$P(as) \to P(a : as)$	$\to$-I, 1–6
8			$\forall xs.\ P(xs) \to P(a : xs)$	$\forall$-I, 7
9			$\forall x.\ \forall xs.\ P(xs) \to P(x : xs)$	$\forall$-I, 8

Concatenation

Show: `length (xs ++ ys) = length xs + length ys`

- statement contains *two* lists: *xs* and *ys*
- but induction principle only allows for *one*?

Formally. Show that

$$\forall xs. \underbrace{\forall ys. \texttt{length (xs ++ ys) = length xs + length ys}}_{P(xs)}.$$

Equivalent Alternative.

$$\forall ys. \underbrace{\forall xs. \texttt{length (xs ++ ys) = length xs + length ys}}_{P(ys)}.$$

As a slogan.

- list induction allows us to induct on one list *only*.
- the other list is treated as a constant.
- but on which list should we induct?

List Concatenation: Even more Options!

Show: `length (xs ++ ys) = length xs + length ys`

Option 1. Do induction on xs

$$\forall xs.\underbrace{\forall ys.\,\texttt{length (xs ++ ys) = length xs + length ys}}_{P(xs)}$$

Option 2. Reformulate and do induction on ys

$$\forall ys.\underbrace{\forall xs.\,\texttt{length (xs ++ ys) = length xs + length ys}}_{P(ys)}.$$

Option 3. Fix an arbitrary ys and show the below, then use $\forall I$

$$\forall xs.\underbrace{\texttt{length (xs ++ ys) = length xs + length ys}}_{P(xs)}.$$

Option 4. Fix an arbitrary xs and show the below, then use $\forall I$

$$\forall ys.\underbrace{\texttt{length (xs ++ ys) = length xs + length ys}}_{P(ys)}.$$

Choosing the most helpful formulation

Problem. For `length (xs ++ ys) = length xs + length ys`

- induct on *xs* (and treat *ys* as a constant), or
- induct on *ys* (and treat *xs* as a constant)?

Clue. Look at the definition of `xs ++ ys`:

```
[]     ++ ys = ys              -- (A1)
(x:xs) ++ ys = x : (xs ++ ys)  -- (A2)
```


- the list *xs* (i.e. the first argument of ++) *changes*
- the second argument (i.e. `ys`) remains *constant*

Approach. Induction on *xs* and treat *ys* as a constant, i.e.

$$\forall xs. \underbrace{\forall ys.\texttt{length (xs ++ ys) = length xs + length ys}}_{P(xs)}.$$

The Base Case

Given.

```
length []         = 0                    -- (L1)
length (x:xs)     = 1 + length xs        -- (L2)
```



```
map f []          = []                   -- (M1)
map f (x:xs)      = f x : map f xs       -- (M2)
```



```
[]     ++ ys      = ys                   -- (A1)
(x:xs) ++ ys      = x : (xs ++ ys)       -- (A2)
```

Base Case. $P([\,])$ We want to prove


```
length ([] ++ ys) = length [] + length ys
```

```
length ([] ++ ys) = length ys -- by (A1)
                  = 0 + length ys
                  = length [] + length ys -- by (L1)
```

Concatenation preserves length: step case

Step Case. Show that $\forall x.\ \forall xs.\ P(xs) \rightarrow P(x : xs)$

Assume $P(as)$

$\forall ys.$ `length (as ++ ys) = length as + length ys -- (IH)`

Prove $P(a : as)$, that is

$\forall ys.$`length ((a:as) ++ ys) = length (a:as) + length ys`

For arbitrary ys we have:


```
length ((a:as) ++ ys)
     = length (a : (as ++ ys))    -- by (A2)
     = 1 + length (as ++ ys)      -- by (L2)
     = 1 + length as + length ys  -- by (IH)
     = length (a:as) + length ys  -- by (L2)
```

Theorem proved!

A few meta-points:

On the induction hypothesis:

- The *induction hypothesis* ties the recursive knot in the proof.
- If you haven't used it, the proof is likely wrong.
- It's important to know what *precisely* the *induction hypothesis* actually is!

On rules:

- Only use the rules that are given, that is
  - the function definitions
  - the induction hypothesis
  - basic arithmetic

Concatenation Distributes over Map

Show: `map f (xs ++ ys) = map f xs ++ map f ys`

Which list?

- as before: ++ defined by recursion on xs
- treat ys as a constant.

Show.

$$\forall xs\ \forall ys.\ \underbrace{\texttt{map f (xs ++ ys) = map f xs ++ map f ys}}_{P(xs)}$$

So let $P(xs)$ be `map f (xs ++ ys) = map f xs ++ map f ys`

Base Case: $P([\,])$. Show for arbitrary ys, that

```
map f ([] ++ ys)  = map f [] ++ map f ys.
```

```
map f ([] ++ ys)  = map f ys              -- by (A1)
                  = [] ++ map f ys        -- by (A1)
                  = map f [] ++ map f ys  -- by (M1)
```

Concatenation Distributes over Map, Continued

Step Case: $\forall x.\ \forall xs.\ P(xs) \rightarrow P(x : xs)$

Assume $P(as)$

```
map f (as ++ ys)  = map f as ++ map f ys   -- (IH)
```


Prove $P(a : as)$ that is

```
map f ((a:as) ++ ys) = map f (a:as) ++ map f ys
```



```
map f ((a:as) ++ ys)
    = map f (a : (as ++ ys))        -- by (A2)
    = f a : map f (as ++ ys)        -- by (M2)
    = f a : (map f as ++ map f ys)  -- by (IH)
    = (f a : map f as) ++ map f ys  -- by (A2)
    = map f (a:as) ++ map f ys      -- by (M2)
```

Theorem proved!

Observe a Trilogy

- **Inductive Definition** defines *all* lists

  `data [a] = [] | a : [a]`

- **Recursive Function Definitions** give a value for *all* lists

  ```
  f [] = ....
  f (x:xs) = ....   (definition usually involves f xs)
  ```

- **Structural Induction Principle** establishes property for *all* lists

  Prove $P([\,])$

  Prove $\forall x.\ \forall xs.\ P(xs) \rightarrow P(x : xs)$ (proof usually uses $P(xs)$)

- Each version has a base case and a step case.
- The form of the inductive type definition determines the form of recursive function definitions and the structural induction principle.

Induction on Finite Trees

Inductive Definition of finite trees (for an arbitrary type a)

1. `Nul` is of type `Tree a`
2. If `l` and `r` are of type `Tree a` and `x` is of type `a`, then `Node l x r` is of type `Tree a`.

No object is a finite tree of a's unless justified by these clauses.

Tree Induction. To show that $P(t)$ for all t of type `Tree a`:

- Show that $P(\texttt{Nul})$ holds
- Show that $P(\texttt{Node l x r})$ holds whenever both $P(\texttt{l})$ and $P(\texttt{r})$ are true.

Induction for Lists and Trees

Natural Numbers.


```
data Nat =
  O | S Nat
```

$$\frac{P(O) \qquad \forall n. P(n) \rightarrow P(S\,n)}{\forall n. P(n)}$$

Lists.


```
data [a] =
  [] | a : [a]
```

$$\frac{P([]) \qquad \forall \mathtt{x}. \forall \mathtt{xs}. P(\mathtt{xs}) \rightarrow P(\mathtt{x} : \mathtt{xs})}{\forall \mathtt{xs}. P(\mathtt{xs})}$$

Trees.

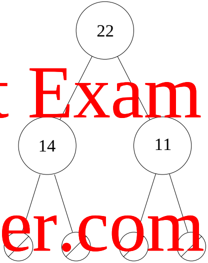

```
data Tree a =
  Nul
| Node (Tree a) a (Tree a)
```

$$\frac{P(\mathtt{Nul}) \qquad \forall \mathtt{l}. \forall \mathtt{x}. \forall \mathtt{r}. P(\mathtt{l}) \wedge P(\mathtt{r}) \rightarrow P(\mathtt{Node}\,\mathtt{l}\,\mathtt{x}\,\mathtt{r})}{\forall t. P(\mathtt{t})}$$

Why does it Work?

Given.

- *Base Case:* $P(\texttt{Nul})$
- *Step Case:*
  $\forall \texttt{l}.\forall \texttt{x}.\forall \texttt{r}.P(\texttt{l}) \wedge P(\texttt{r}) \rightarrow P(\texttt{Node}\,\texttt{l}\,\texttt{x}\,\texttt{r})$



Show. $P(\texttt{Node}\,(\texttt{Node}\,\texttt{Nul}\,14\,\texttt{Nul})\,22\,(\texttt{Node}\,\texttt{Nul}\,11\,\texttt{Nul}))$

1. $P(\texttt{Nul})$ is given
2. $P(\texttt{Node Nul 14 Nul})$ follows from $P(\texttt{Nul})$ and $P(\texttt{Nul})$
3. $P(\texttt{Node Nul 11 Nul})$ follows from $P(\texttt{Nul})$ and $P(\texttt{Nul})$
4. $P(\texttt{Node (Node Nul 14 Nul) 22 (Node Nul 11 Nul)})$
   follows from $P(\texttt{Node Nul 14 Nul})$ and $P(\texttt{Node Nul 11 Nul})$

Induction on Structure

Data Type.


```
data Tree a =
    Nul
  | Node (Tree a) a (Tree a)
```


Tree Induction as a proof rule:

$$\frac{P(Nul) \qquad \forall t_1.\ \forall x.\ \forall t_2.\ P(t_1) \wedge P(t_2) \rightarrow P(Node\ t_1\ x\ t_2)}{\forall t P(t)}$$

with the following types:

- $x :: a$ is of type a
- $t_1 :: Tree\ a$ and $t_2 :: Tree\ a$ are of type $Tree\ a$.

Standard functions

```
mapT f Nul                  = Nul                        -- (M1)
mapT f (Node t1 x t2)
      = Node (mapT f t1) (f x) (mapT f t2)               -- (M2)

count Nul                   = 0                          -- (C1)
count (Node t1 x t2)
    = 1 + count t1 + count t2                            -- (C2)
```


Example. We use tree induction to show that

`count (mapT f t) = count t`

holds for all functions `f` and all trees `t`.

(Analogous to `length (map f xs) = length xs` for lists)

Show `count (mapT f t) = count t`

Base Case: $P(Nul)$


```
count (mapT f Nul) = count Nul
```

This holds by `(M1)`

Step case

Show: $\forall$`l`.$\forall$`x`.$\forall$`r`. P(`l`) $\wedge$ P(`r`) $\to$ P(`Node l x r`)

Induction Hypothesis for arbitrary u_1 and u_2: $P(u_1) \wedge P(u_2)$ written as

```
count (mapT f u1) = count u1          -- (IH1)
count (mapT f u2) = count u2          -- (IH2)
```


Proof Goal. For arbitrary `a`, show that P(`Node u1 a u2`), i.e.


```
count (mapT f (Node u1 a u2)) = count (Node u1 a u2)
```

Step case continued

Proof Goal. P(`Node u1 a u2`), i.e.

```
count (mapT f (Node u1 a u2)) = count (Node u1 a u2)
```


Our Reasoning:

```
  count (mapT f (Node u1 a u2))
= count (Node (mapT f u1) (f x) (mapT f u2)) -- by (M2)
= 1 + count (mapT f u1) + count (mapT f u2)  -- by (C2)
= 1 + count u1 + count u2                    -- by (IH1, IH2)
= count (Node u1 a u2)                       -- by (C2)
```


Theorem proved!

Observe the Trilogy Again

There are three related stories exemplified here, now for trees

- **Inductive Definition**

  ```
  data Tree a = Nul | Node (Tree a) a (Tree a)
  ```

- **Recursive Function Definitions**

  ```
  f Nul          = ...
  f (Node l x r) = ...
  ```


- **Structural Induction Principle**

  Prove $P(\texttt{Nul})$

  Prove $\forall \texttt{l}.\forall \texttt{x}.\forall \texttt{r}.P(\texttt{l}) \wedge P(\texttt{r}) \rightarrow P(\texttt{Node l x r})$

Similarities.

- One definition / proof obligation per Constructor
- Assuming that smaller cases are already defined / proved

Flashback: Accumulating Parameters

Two version of summing a list:


```
sum1 []            = 0                         -- (S1)
sum1 (x:xs)        = x + sum1 xs               -- (S2)

sum2 xs            = sum2' 0 xs                -- (T1)
sum2' acc []       = acc                       -- (T2)
sum2' acc (x:xs) = sum2' (acc + x) xs          -- (T3)
```


Crucial Differences.

- one parameter in `sum1`, two in `sum2`
- *both* parameters change in the recursive call in `sum2`

Show: `sum1 xs = sum2 xs`

```
sum1 []           = 0                        -- (S1)
sum1 (x:xs)       = x + sum1 xs              -- (S2)

sum2 xs           = sum2' 0 xs               -- (T1)
sum2' acc []      = acc                      -- (T2)
sum2' acc (x:xs)  = sum2' (acc + x) xs       -- (T3)
```


Base Case: $P([])$

```
sum2 [] = sum1 []
```

```
sum2 [] = sum2' 0 []   -- by (T1)
        = 0            -- by (T2)
        = sum1 []      -- by (S1)
```

Step case

Step Case: $\forall x. \forall xs. P(xs) \rightarrow P(x : xs)$

Assume:

```
sum2 as = sum1 as        -- (IH)
```


Prove:

```
sum2 (a:as) = sum1 (a:as)
```



```
sum2 (a:as) = sum2' 0 (a:as)      -- by (T1)
            = sum2' (0 + a) as    -- by (T3)
```



```
sum1 (a:as) = a + sum1 as         -- by (S2)
            = a + sum2 as               -- by (IH)
            = a + sum2' 0 as            -- by (T1)
```

Problem.

- can't apply IH: as $0 \neq 0 + a$
- accumulating parameter in `sum2` has *changed*

Proving a Stronger Property

Solution. Prove a property that involved *both* arguments.

```
sum1 []          = 0                      -- (S1)
sum1 (x:xs)      = x + sum1 xs            -- (S2)
```



```
sum2 xs          = sum2' 0 xs             -- (T1)
sum2' acc []     = acc                    -- (T2)
sum2' acc (x:xs) = sum2' (acc + x) xs     -- (T3)
```


Observation. (from looking at the code, or experimenting)

```
sum2' acc xs  = acc + sum1 xs
```

Formally. We show that

$$\forall \texttt{xs}. \underbrace{\forall \texttt{acc}. \texttt{sum2' acc xs = acc + sum1 xs}}_{P(\texttt{xs})}$$

Base Case: Show $P([])$, i.e. $\forall$`acc.acc + sum1 [] = sum2' acc []`.

```
acc + sum1 [] = acc + 0 = acc -- by (S1)
              = sum2' acc []  -- by (T2)
```

Step case

Step Case. $\forall \mathtt{x}.\forall \mathtt{xs}.P(\mathtt{xs}) \rightarrow P(\mathtt{x : xs})$.

Induction Hypothesis


```
∀acc.acc + sum1 as = sum2' acc as      (IH)
```

Show.


```
∀acc.acc + sum1 (a:as) = sum2' acc (a:as)
```

Our Reasoning:


```
acc + sum1 (a:as) = acc + a + sum1 as    -- by (S2)
                  = sum2' (acc + a) as   -- by (IH) (*)
                  = sum2' acc (a:as)     -- by (T3)
```

- Our induction hypothesis is ∀ `acc`. ...
- In (∗) we instantiate ∀`acc` with it `acc + a`
- ∀ `acc` is *absolutely needed* in induction hypothesis

Proving the Original Property

We have. $\forall xs.\ P(\texttt{xs})$, that is:

```
∀ xs. ∀ acc. acc + sum1 xs = sum2' acc xs
```


Equivalent Formulation. (change order of quantifiers)

```
∀ acc. ∀ xs. acc + sum1 xs = sum2' acc xs
```


Instantiation `(acc = 0)`

```
∀ xs. 0 + sum1 xs = sum2' 0 xs          -- by ∀-E
∀ xs. sum1 xs     = sum2' 0 xs          -- by arith
∀ xs. sum1 xs     = sum2 xs             -- by T1
```


That is, we have (finally) proved the original property.

When might a stronger property P be necessary ?

Alarm Bells.

```
sum2' acc (x:xs) = sum2' (acc + x) xs
```


Pattern.

- *both* arguments change in recursive calls

Programming Perspective.

- to evaluate `sum2'`, need evaluation steps where `acc` $\neq 0$

Proving Perspective.

- to prove facts about `sum2'`, need inductive steps where `acc` $\neq 0$

Orthogonal Take.

- `sum2'` is *more capable* than `sum2` (works for *all* values of `acc`)
- when proving, need *stronger statement* that also works for all `acc`

Look at proving it for $xs = [2, 3, 5]$

Backwards Proof for a special case:

```
0 + sum1 [2,3,5] = sum2' 0 [2,3,5]   because
0 + 2 + sum1 [3,5] = sum2' (0+2) [3,5]   because
0 + 2 + 3 + sum1 [5] = sum2' (0+2+3) [5]   because
0 + 2 + 3 + 5 + sum1 [] = sum2' (0+2+3+5) []   because
0 + 2 + 3 + 5 = (0+2+3+5)
```


Termination.

- the list gets shorter with every recursive call
- despite the accumulator getting larger!

Another example

```
flatten :: Tree a -> [a]
flatten Nul              = []                           -- (F1)
flatten (Node l a r) = flatten l ++ [a] ++ flatten r -- (F2)
```



```
flatten2 :: Tree a -> [a]
flatten2 tree = flatten2' tree []                      -- (G)
```



```
flatten2' :: Tree a -> [a] -> [a]
flatten2' Nul acc = acc                                -- (H1)
flatten2' (Node l a r) acc =
   flatten2' l (a:flatten2' r acc)                     -- (H2)
```


Show.

`flatten2' t acc = flatten t ++ acc`

for all `t :: Tree a`, and all `acc :: [a]`.

Proof

Proof Goal.

$$\forall \texttt{t}.\underbrace{\forall \texttt{acc}.\texttt{flatten2' t acc = flatten t ++ acc}}_{P(t)}$$

Base Case `t = Nul`. Show that

```
flatten2' Nul acc = flatten Nul ++ acc
```



```
flatten2' Nul acc = acc                    -- by (H1)
     = [] ++ acc                           -- by (A1)
     = flatten Nul ++ acc                  -- by (F1)
```


Step Case: `t = Node t1 y t2`. Assume that for *all* `acc`,

```
flatten2' t1 acc = flatten t1 ++ acc       -- (IH1)
flatten2' t2 acc = flatten t2 ++ acc       -- (IH2)
```

Required to Show. For *all* `acc`,

```
flatten2' (Node t1 y t2) acc = flatten (Node t1 y t2) ++ acc
```

Proof (continued)

Proof (of Step Case): Let *a* be given (we will generalise *a* to ∀`acc`)


```
flatten2' (Node t1 y t2) a
  = flatten2' t1 (y : flatten2' t2 a)      -- by (H2)
  = flatten t1 ++ (y : flatten2' t2 a)     -- (IH1)(*)
  = flatten t1 ++ (y : flatten t2 ++ a)    -- (IH2)(*)
  = flatten t1 ++ ((y : flatten t2) ++ a)  -- by (A2)
  = (flatten t1 ++ (y : flatten t2)) ++ a  -- (++ assoc)
  = flatten (Node t1 y t2) ++ a            -- by (F2)
```


Notes.

- in `IH1`, `acc` is instantiated with `(y : flatten2' t2 a)`
- in `IH1`, `acc` is instantiated with `a`

As a was arbitrary, this completes the proof.

General Principle

Inductive Definition

- `data Tree a = Nul | Node (Tree a) a (Tree a)`
- Constructors with arguments, may include type being defined!

Structural Induction Principle

- Prove $P(\texttt{Nul})$
  Prove $\forall \texttt{l}.\forall \texttt{x}.\forall \texttt{r}.P(\texttt{l}) \wedge P(\texttt{r}) \rightarrow P(\texttt{Node}\,\texttt{l}\,\texttt{x}\,\texttt{r})$
- One proof obligation for each constructor
- All arguments universally quantified
- May assume property of *same type* arguments

General Principle: Example

Given. Inductive data type definition of type `T`

```
data T =
    C1 Int
  | C2 T T
  | C3 T Int T
```

```
Constructors:
    C1 :: Int -> T
  | C2 :: T -> T -> T
  | C3 :: T -> Int -> T -> T
```

Q. What does the induction principle for `T` look like?

General Principle: Example

Given. Inductive data type definition of type `T`

```
data T =                     Constructors:
    C1 Int                       C1 :: Int -> T
  | C2 T T                     | C2 :: T -> T -> T
  | C3 T Int T                 | C3 :: T -> Int -> T -> T
```


Q. What does the induction principle for `T` look like?

A. To show $\forall t :: T, P(t)$, need to show

- *three* things (*three* constructors)
- all arguments are *universally* quantified
- $P(\mathtt{t})$ may be assumed for arguments of type `t:T`

More Concretely. To show $\forall t :: T, P(t)$, need to show

- $\forall \mathtt{n}. P(\mathtt{C1\ n})$
- $\forall \mathtt{t1}. \forall \mathtt{t2}. P(\mathtt{t1}) \wedge P(\mathtt{t2}) \rightarrow P(\mathtt{C2\ t1\ t2})$
- $\forall \mathtt{t1}. \forall \mathtt{n}. \forall \mathtt{t2}. P(\mathtt{t1}) \wedge P(\mathtt{t2}) \rightarrow P(\mathtt{C3\ t1\ n\ t2})$

Induction on Formulae

Boolean Formulae without negation as Inductive Data Type

```
data NFForm =
    TT
  | Var Int
  | Conj NFForm NFForm
  | Disj NFForm NFForm
  | Impl NFForm NFForm
```


Induction Principle. $\forall \texttt{f} : \texttt{NFForm}. P(\texttt{f})$ follows from

- $P(\texttt{TT})$
- $\forall \texttt{n}. P(\texttt{Var n})$
- $\forall \texttt{f1}. \forall \texttt{f2}. P(\texttt{f1}) \wedge P(\texttt{f2}) \rightarrow P(\texttt{Conj f1 f2})$
- $\forall \texttt{f1}. \forall \texttt{f2}. P(\texttt{f1}) \wedge P(\texttt{f2}) \rightarrow P(\texttt{Disj f1 f2})$
- $\forall \texttt{f1}. \forall \texttt{f2}. P(\texttt{f1}) \wedge P(\texttt{f2}) \rightarrow P(\texttt{Impl f1 f2})$

Recursive Definition

Given.

```
data NFForm =
    TT
  | Var Int
  | Conj NFForm NFForm
  | Disj NFForm NFForm
  | Impl NFForm NFForm
```


Evaluation of a (negation free) formula:

```
eval :: (Int -> Bool) -> NFForm -> Bool
eval theta TT = True
eval theta (Var n) = theta n
eval theta (Conj f1 f2) = (eval theta f1) && (eval theta f2)
eval theta (Disj f1 f2) = (eval theta f1) || (eval theta f2)
eval theta (Impl f1 f2) = (not (eval theta f1)) || (eval theta f2)
```

Example Proof

Theorem. If `f` is a negation free formula, then `f` evaluates to `True` under the valuation `theta _ = True`.

More precise formulation. Let `theta` be defined by `theta _ = True`. Then, for all `f` of type `NFForm`, we have `eval theta f = True`.

Proof using the induction principle *for negation free formulae*.

Base Case 1. Show that `eval theta TT = True`. (immediate).

Base Case 2. Show that `\n. eval theta (Var n) = True`.


```
eval theta (Var n) = theta n = True
```

(by definition of `eval` and definition of `theta`)

Proof of Theorem, Continued

Step Case 1. Assume that

- `eval theta f1 = True (IH1)` and
- `eval theta f2 = True (IH2)`.

Show that

- `eval theta (Conj f1 f2) = True`

Proof (of Step Case 1).

```
  eval theta (Conj f1 f2)
= (eval theta f1) && (eval theta f2)  -- defn eval
= True            && True             -- IH1, IH2
= True                                -- defn &&
```

Wrapping Up

Step Case 2 and Step Case 3. In both cases, we may assume

- `eval theta f1 = True` (IH1) and
- `eval theta f2 = True` (IH2).

and need to show that

- `eval theta (Conj f1 f2) = True` (Step Case 2)
- `eval theta (Impl f1 f2) = True` (Step Case 3)

The reasoning is almost identical to that of Step Case 1, and we use

```
True  || True  = True
False || True  = True
```


Summary. Having gone through all the (base and step) cases, the theorem is proved using induction for the data type `NFForm`.

Inductive Types: Degenerate Examples

Consider the following Haskell type definition:

```
data Roo a b =
  MkRoo a b
```


Q. Given types `a` and `b`, what is the type `Roo a b`?

Inductive Types: Degenerate Examples

Consider the following Haskell type definition:

```
data Roo a b =
  MkRoo a b
```


Q. Given types a and b, what is the type Roo a b?

A. It is the type of *pairs* of elements of a and b

- To make an element of Roo a b, can use constructor MkRoo: a -> b -> Roo a b
- No other way to 'make' elements of Roo a b

Let's give this type its usual name:

```
data Pair a b =
  MkPair a b
```

Recursion and Induction Principle

Data Type.


```
data Pair a b =
  MkPair a b
```


Pair Recursion. To define a function `f::Pair a b -> T`:

```
f (MkPair x y) = ...  x ...  y ...
```


we may use *both* the values of `x` and `y` – same as for pairs

Pair Induction. To prove $\forall \texttt{x::Pair a b}.P(\texttt{x})$

- show that $\forall \texttt{x}.\forall \texttt{y}.P(\texttt{MkPair xy})$

just *one* constructor and *no* occurrences of arguments of pair type

Inductive Types: More Degenerate Examples

Consider the following Haskell type definition:

```
data Wombat a b =
  Left  a
| Right b
```

Q. Given types `a` and `b`, what is the type `Wombat a b`?

Inductive Types: More Degenerate Examples

Consider the following Haskell type definition:

```
data Wombat a b =
  Left  a
| Right b
```


Q. Given types a and b, what is the type `Wombat a b`?

A. It is the type of tagged elements of either a or b:

- use constructor `Left:  a -> Wombat a b`
- use the constructor `Right:  b -> Wombat a b`
- No other way to "make" elements of `Wombat a b`.

Let's give this type its usual name:

```
data CoPair a b =
  Left  a
| Right b
```

Recursion and Induction Principle

Data Type.

```
data CoPair a b =
  Left  a
| Right b
```


Copair Recursion. To define a function `f :: CoPair a b -> T`:


```
 f (Left x)  = ...  x ...  ...
f (Right y)  = ...  y ...  ...
```

we *have* to give equations for both cases: left and right.

Copair Induction. To prove $\forall$`z::CoPair a b`.$P($`z`$)$

- show that $\forall$`x`.$P($`Left x`$)$
- show that $\forall$`y`.$P($`Right y`$)$

here: *two* constructors and *no* occurrences of arguments of copair type

Limitations of Inductive Proof

Termination. Consider the following (legal) definition in Haskell


```
nt :: Int -> Int
nt x = nt x +1
```


Taking *Haskell definitions* as *equations* we can prove 0 = 1:

```
0 = nt 0 - nt 0 = nt 0 + 1 - nt 0 = 1
```


I.e. a statement that is *patently* false.

Limitation 1. The proof principles outlined here only work if *all functions are terminating*.

Limitations, Continued

Finite Data Structures. Consider the following (legal) Haskell definition

```
blink :: [Bool]
blink = True:False:blink
```

and consider for example

```
length blink
```

Clearly, `length blink` is *undefined* and so may introduce *false* statements.

Limitation 2. The proof principles outlined here only work for all *finite* elements of inductive types.

Addressing Termination

Q. How do we *prove* that a function terminates?

Example 1. The argument gets "smaller".

```
length [] = 0
length (x:xs) = 1 + length xs
```


Example 2. Only one argument gets "smaller"?


```
length' [] a = a
length' (x:xs) a = length' xs (a+1)
```

Q. What does "getting smaller" really mean?

Termination Measures

Given. The function `f` defined below as follows

```
f :: T1 -> T2
f x = exp(x)
```


Q. When does the argument of `f` "get smaller"?

A. Need *measure* of smallness: `m: T1 ->` $\mathbb{N}$

Informally.

- in every recursive call, the measure `m` of the argument of the call is smaller
- termination, because natural numbers cannot get smaller indefinitely.

Formally. A function `m: T1 ->` $\mathbb{N}$ is a *termination measure* for `f` if

- for every defining equation `f x = exp`, and
- for every recursive call `f y` in `exp`

we have that `m y < m x`.

Example

List Reversal.

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = (rev xs) + [x]
```


Termination Measure.

```
m :: [a] -> N
m xs = length xs
```


Recursive Calls only in the second line of function definition

- Show that `m xs < m (x:xs)`
- I.e. `length xs < length (x:xs)` – this is obvious.

Termination Measures: General Case

Consider a recursively defined function

```
f : T1 -> T2 -> ... -> Tn -> T0
```


that is defined using multiple equations of the form

```
f x1 .. xn = exp(x1, ..., x_n)
```


taking n arguments of types `T1`, ..., `Tn` and computes a value of type `T`.

Definition. A *termination measure* for `f` is a function of type

$$\mathtt{m: T1 \text{ -> } T2 \text{ -> } \ldots \text{ -> } Tn \text{ -> }} \mathbb{N}$$

such that

- for every defining equation `f x1 ... xn = exp`, and
- for every recursive call `f y1 ... yn` in `exp`

we have that `m y1 .. yn < m x1 ... xn`.

Termination Proofs

Theorem. Let `f: T1 -> ... -> Tn -> T` be a function with termination measure `m: T1 -> T2 -> ...-> Tn -> ` $\mathbb{N}$.
Then the evaluation of `f x1 .. xn` terminates for all `x1`, ..., `xn`.

Proof. We show the following statement by induction on $n \in \mathbb{N}$.

$\forall n.$if `m x1 ... xn` $<$ `n` then `f x1 .. xn` terminates

Base Case. $n = 0$ is trivial(!)

Step Case. Assume that the statement is true for *all* $n_0 < n$ and let `x1`, ..., `xn` be given. Then the recursive call


```
f x1 .. xn = exp(x1, ..., xn)
```

only contains calls of the form `f y1 .. yn` for which `m y1 .. yn < m x1 ... xn` so that these calls terminate by induction hypothesis. Therefore `f x1 ... xn` terminates.

Example

```
rev_a :: [a] -> [a] -> [a]
rev_a [] ys = ys
rev_a (x:xs) = rev_a xs (x:ys)
```


Termination Measure (for any type a)

```
m :: [a] -> [a] -> N
m  xs ys = length xs
```

Recursive Calls only in second line of function definition.

- Show that `m xs (x:ys) < m (x:xs) ys`.
- I.e. `length xs < length (x:xs)` – this is obvious.

Outlook: Induction Principles

More General Type Definitions

```
data Rose a =
   Rose a [Rose a]
```

```
data TTree a b =
   Wr b | Rd (a -> TTree a b)
```


Example using `TTree`


```
eat :: TTree a b -> [a] -> b
eat  (Wr y) _ = y
eat  (Rd f) (x:xs) = eat (f x) xs
```


Induction Principles

- for `Rose`: may assume IH for all list elements
- for `TTree`: mayh assume IH for all values of `f`

Outlook: Termination Proofs

More Complex Function Definitions


```
ack :: Int -> Int -> Int
ack 0 y = y+1
ack x 0 = ack (x-1) 1
ack x y = ack (x-1) (ack x (y-1))
```


Termination Measures

- `m x y = x` doesn't account for last line of function definition
- difficulty: *nested* recursive calls

Digression. Both induction and termination proofs scratch the surface!

Outlook: Formal Proof in a Theorem Prover

The Coq Theorem Prover `https://coq.inria.fr`

- based on Theory *Coq*and's Calculus of Constructions
- *requires* that all functions terminate.

Examples.

- Natural Deduction:

```
Lemma ex_univ {A: Type} (P: A -> Prop) (Q: Prop):
      ((exists x, P x) -> Q) -> forall x,  P x -> Q.
```


- Inductive Proofs:

```
Lemma len_map {A B: Type} (f: A -> B): forall (l: list A),
  length l = length (map f l).
```

(and some other examples)