Structural Induction (continued)

COMP2600 / COMP6260

Dirk Pattinson
Australian National University

Semester 2, 2015

Induction on Finite Trees

Like natural numbers and lists, trees are also *inductively defined*.

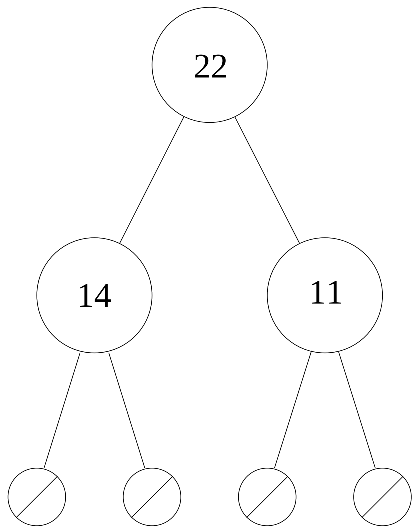
1. $Nul \;::\; Tree\ a$
2. If $x :: a$ and $t_1 :: Tree\ a$ and $t_2 :: Tree\ a$
   then $Node\ x\ t_1\ t_2 :: Tree\ a$

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

To prove a property for all trees (of objects of some type):

- Prove it for Nul
- Prove that, whenever it is true for t_1 and t_2 it is true for $Node\ x\ t_1\ t_2$

Why does it Work?



Suppose we have proved:

- The *base case*: $P(Nul)$
- The *step case*:
  $\forall t_1.\ \forall t_2.\ P(t_1) \wedge P(t_2) \rightarrow P(Node\ x\ t_1\ t_2\ )$

We can use these facts to prove
$P(Node\ 22\ (Node\ 14\ Nul\ Nul)\ (Node\ 11\ Nul\ Nul))$

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

That's Induction on Structure

The rule of *Structural Induction for Trees* is usually written as:

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

or, being fussy with types:

$$\frac{P(Nul :: Tree\ a) \quad \forall (x :: a). \forall (t_1\ t_2 :: Tree\ a).\ P(t_1) \wedge P(t_2) \rightarrow P(Node\ x\ t_1\ t_2)}{\forall (t :: Tree\ a).\ P(t)}$$

Standard functions

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

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

The theorem we will prove about trees is `count (mapT f t) = count t`

It is the tree analog of the list property `length (map f xs) = length xs`

Prove: `count (mapT f t) = count t`

This time we're doing induction over trees.

The smallest possible tree is *Nul*

Base Case: $P(Nul)$

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

This holds by `(M1)`

Step case

Step Case: $\forall x.\ \forall t_1.\ \forall t_2.\ P(t_1) \wedge P(t_2) \rightarrow P(Node\ x\ t_1\ t_2)$

This time the induction hypothesis is $P(u_1) \wedge P(u_2)$, but we will write both parts separately.

Assume $P(u_1) \wedge P(u_2)$:

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

Prove $P(Node\ a\ u_1\ u_2)$, that is,

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

Step case continued

Prove $P(Node\ a\ u_1\ u_2)$, that is,

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

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

- **Recursive Function Definitions**

```
f  Nul           = ...
f (Node x t1 t2) = ... f(t1) ... f(t2) ...
```

- **Structural Induction Principle**

  Prove $P(Nul)$

  Prove $\forall x.\forall t_1.\forall t_2.\ P(t_1) \wedge P(t_2) \rightarrow P(Node\ x\ t_1\ t_2)$

The similarity is that each 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.

Structural Induction *vs* accumulating parameters

Here are two versions of the sum function. The second one uses an accumulating parameter and requires less space at run-time.

```
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)
```

Prove: `sum1 xs = sum2 xs`

We will prove that the two definitions of sum are equivalent:

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)
```

Now we're stuck. We used the IH, but both sides aren't the same...

Proving a Stronger Property

Sometimes we need to prove a stronger property than the one we are given.

How to describe `sum2'` ? `sum2' acc xs` is `acc` + sum of `xs`

So here is a property which involves the current accumulator in `sum2'`

Let $P(xs)$ be $\forall$ `acc. acc + sum1 xs = sum2' acc xs`

Why include the $\forall$ `acc.` ? (We haven't done this before) — see next slide

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

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

Then generalise over `acc` — uses $\forall$-I

Step case

<u>Step Case:</u> $\forall x.\forall xs.P(xs) \rightarrow P(x : xs)$

Assume: $\forall$ `acc. acc + sum1 as = sum2' acc as    -- (IH)`
Prove: $\forall$ `acc. acc + sum1 (a:as) = sum2' acc (a:as)`

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

Then generalise over `acc` — uses $\forall$-I.
Note also the use of associativity of $+$

How is `(*)` a use of the induction hypothesis ?

1. Our induction hypothesis is $\forall$ `acc. ...`
2. We can instantiate it at `(acc + a)` to give the equality we need.
3. Without $\forall$ `acc` in the induction hypothesis, this proof would not work.

Strong(xs) $\rightarrow$ Weak(xs)

We have now proved $\forall xs.\ P(xs)$, that is:

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

Change the order of the quantifiers:

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

Instantiate at `(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
```

Which was our original property required.

When might a stronger property P be necessary ?

The clue here is that

- To evaluate `sum2 xs`, that is `sum2' 0 xs`, there are recursive steps of evaluating `sum2' acc xs`, for `acc` $\neq 0$.
- Likewise, to prove something about `sum2 xs`, that is `sum2' 0 xs`, there are inductive steps where you prove something about `sum2' acc xs`, for `acc` $\neq 0$

To put it another way

- in the code we need to define a function (`sum2'`, works for *all* `acc`) which is *more* capable than we want (`sum2`, uses only `acc` $=$ 0)
- in the proof we need to prove a *stronger* result (for *all* `acc`) than we want (for `acc` $=$ 0)

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

We can write out a backwards form of the proof:

`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)`

This "proof attempt" terminates because the list gets smaller each time, even though the accumulator changes (in fact it gets bigger).

In the same way, in the function definition for `sum2'`, evaluation of `sum2' acc [2,3,5]` terminates because the list gets smaller each time, even though the accumulator changes (in fact it gets bigger).

Another example

Given the functions below:

```
flatten :: Tree a -> [a]
flatten Nul            = []                              -- (F1)
flatten (Node a t1 t2) = flatten t1 ++ [a] ++ flatten t2 -- (F2)

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

flatten2' :: Tree a -> [a] -> [a]
flatten2' Nul acc = acc                                  -- (H1)
flatten2' (Node a t1 t2) acc =
    flatten2' t1 (a : flatten2' t2 acc)                  -- (H2)
```

Prove, by induction on the structure of binary trees, that for all `t :: Tree a`, and for all `acc :: [a]`,

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

Proof

The property P we prove by induction will be

`P(t) = ∀acc.  flatten2′ t acc = flatten t ++ acc`

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 y t1 t2`.

Show that if, *for all* `acc`

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

then, *for all* `acc`

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

Proof (continued)

Proof (of Step Case): Let *a* be given (we will generalise *a* to $\forall$acc)

```
flatten2' (Node y t1 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 y t1 t2) ++ a            -- by (F2)
```

`(*)` Note - the *acc* in (IH1) is instantiated to `(y : flatten2' t2 a)`,
the *acc* in (IH2) is instantiated to *a*

Then we can generalise (using the $\forall$-I rule) *a* to *acc*, to get:
$\forall$`acc. flatten2' (Node y t1 t2) acc = flatten (Node y t1 t2) ++acc`
which completes the proof.