1.5 Structural induction over Haskell data types

Slide 86

Week 4 – First Challenge

Consider following functions:

```
check :: [a] -> Bool
check []    = true
check x:xs = check1 (x:xs) []

check1 :: [a]  -> [a] -> Bool
check1 [] zs = false
check1 (y:ys) zs | (y:ys)==zs  = true
                 | ys==zs      = true
                 | otherwise   = check1 ys (y:zs)
```

Palindrome ⊆ `[a]` is defined as

$$Palindrome(\texttt{xs}) \equiv \exists \texttt{ys} : \texttt{[a]}, \texttt{y} : \texttt{a}. [\ \texttt{xs} = \texttt{ys++(rev ys)} \lor \texttt{xs} = \texttt{ys++y++(rev ys)}\ ]$$

Show that

$(*)\quad \forall \texttt{xs} : \texttt{[a]}. [\ \texttt{check xs = true} \longleftrightarrow Palindrome(\texttt{xs})\ ]$

The functions `check` and `checkAux` appeared in the first term's logic exercises 8, under the names `guess4` and `guess5`. We will discuss the proof later. For the proof we may use the following properties of lists `xs:[a]`.

(A) $\texttt{xs} = \texttt{rev(rev xs)}$

(B) $Palindrome(\texttt{xs}) \longleftrightarrow Palindrome(\texttt{(rev xs)})$

(C) $\texttt{rev (x:xs)} = \texttt{(rev xs)++x}$

Slide 87

Week 4 – Second Challenge

Remember the cactus/beetle example: A cactus consists of a tree, whose nodes have arbitrary numbers of children.
When the beetle eats a leaf lf, then lf is removed, and if lf is not a child of the root, then, the cactus grows back as follows:
Let T_{lf} be the subtree starting at $parent(lf)$ where lf has been removed.
Add k copies of T_{lf} under $parent(parent(lf))$, where k is an arbitrary natural number.

The cactus is *consumed* when it consists of the root only.

How do we formulate that all cacti can be consumed? How do we prove it?

Slide 88

Structural induction over Haskell data types

Slide 89

Structural induction - motivation

Consider the following Haskell functions

```
elem   :: Eq a => a -> [a] -> Bool
elem x []     =   False
elem x (y:ys)   =   x == y || elem x ys

subList :: Eq a => [a] -> [a] -> [a]
subList [] ys    =   []
subList (x:xs) ys
    | elem x ys      =   subList xs ys
    | otherwise      =   x:(subList xs ys)
```

and the specification

$\forall \texttt{xs}:[\texttt{a}].\forall \texttt{ys}:[\texttt{a}].\forall \texttt{z} : \texttt{a}.\ [\ \texttt{z} \in \texttt{ys}\ \rightarrow\ \texttt{z} \notin \texttt{subList xs ys}\ \ ]$

where $\texttt{z} \in \texttt{ys}$ is short for `elem z ys`, and $\texttt{z} \notin \texttt{ys}$ is short for $\neg$(`elem z ys`).

In other words: `sublist xs ys` removes all elements of `ys` from `xs`.

Slide 90

Structural induction - motivation, continued

We want to prove $\forall \texttt{xs}:[\texttt{a}].Q(\texttt{xs})$, where, Q is defined as:

$$Q(\texttt{xs}) \equiv \forall \texttt{ys}:[\texttt{a}].\forall \texttt{z} : \texttt{a}.\ \ [\texttt{z} \in \texttt{ys}\ \ \rightarrow \texttt{z} \notin \texttt{subLst}\ \ \texttt{xs}\ \ \texttt{ys}\ \ ]$$

Can we use induction? Note that Q is *not* defined over numbers.

1st Approach Map lists to numbers,
express $Q \subseteq [\mathrm{a}]$, through an equivalent $P \subseteq \mathbb{N}$.

2nd Approach Use a new principle: structural induction.

Slide 91

1st Approach - mapping lists onto $\mathbb{N}$

We can map lists to numbers through the `length` function.
Define $P(n)$ as:

$$P(n) \equiv \forall \texttt{xs}:\texttt{[a]}.\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}. \\ [\ \texttt{length xs} = n \ \to\ (\ \texttt{z} \in \texttt{ys} \ \to\ \texttt{z} \notin \texttt{subList xs ys}\ )\ ]$$

Observe that

$$\forall n:\mathbb{N}.P(n) \ \leftrightarrow\ \forall \texttt{xs}:\texttt{[a]}.Q(\texttt{xs})$$

Therefore, proving $\forall n:\mathbb{N}.P(n)$ suffices.

The 1st approach We will discuss the application of the 1st Approach

Application of the Mathematical Induction Principle for $\forall n:\mathbb{N}.\forall \texttt{xs}:\texttt{[a]}.\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\texttt{length xs} = n \ \to\ (\ \texttt{z} \in \texttt{ys} \ \to \texttt{z} \notin \texttt{subList xs ys}\ )]$ gives

$$\begin{array}{l}
\quad \forall \texttt{xs}:\texttt{[a]}.\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\texttt{length xs} = 0 \ \to (\ \texttt{z} \in \texttt{ys} \ \to \texttt{z} \notin \texttt{subList xs ys})] \\
\qquad\qquad \wedge \\
\forall k:\mathbb{N}.[\\
\qquad \forall \texttt{xs}:\texttt{[a]}.\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\texttt{length xs} = k \ \to (\ \texttt{z} \in \texttt{ys} \ \to\ \texttt{z} \notin \texttt{subList xs ys})] \\
\qquad \to \\
\qquad \forall \texttt{xs}:\texttt{[a]}.\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\texttt{length xs} = k+1 \to (\texttt{z} \in \texttt{ys} \to \texttt{z} \notin \texttt{subList xs ys})] \\
\qquad\qquad] \\
\quad \longrightarrow \\
\forall n:\mathbb{N}.\forall \texttt{xs}:\texttt{[a]}.\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\texttt{length xs} = n \ \to (\ \texttt{z} \in \texttt{ys} \ \to\ \texttt{z} \notin \texttt{subList xs ys}\ )]
\end{array}$$

Proof

We can now develop a proof by induction. The proof schema is as follows:

Base Case

To Show:

$\forall \texttt{xs}:\texttt{[a]}.\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\texttt{length xs} = 0 \ \to (\ \texttt{z} \in \texttt{ys} \ \to\ \texttt{z} \notin \texttt{subList xs ys})]$

...

Inductive Step

Take $k : \mathbb{N}$ arbitrary.
Ind. Hyp:
$\forall$xs: [a].$\forall$ys: [a].$\forall$z : a. [length xs $= k$ $\rightarrow$ (z $\in$ ys $\rightarrow$ z $\notin$ subList xs ys)]
To show: $\forall$xs: [a].$\forall$ys: [a].$\forall$z : a. [length xs $= k + 1$ $\rightarrow$ (z $\in$ ys $\rightarrow$ z $\notin$ subList xs ys)]

...

We shall use the following properties of `length`:

L1 $\forall$xs: [a]. (length xs $= 0$ $\rightarrow$ xs $=$ [].)

L2 $\forall$xs: [a].$\forall k$:$\mathbb{N}$. [length xs $= k+1$ $\rightarrow$ $\exists$v: a.$\exists$vs: [a].[length vs $= k$ $\wedge$ xs $=$ v:vs]].

We now proceed with the proofs of the base case and the inductive step.

Base Case

To Show: $\forall$xs: [a].$\forall$ys: [a].$\forall$z : a.[length xs $= 0$ $\rightarrow$ (z $\in$ ys $\rightarrow$ z $\notin$ subList xs ys)]

Take arbitrary `xs: [a]`,`ys: [a]`, and `z:a`.
Assume that
(ass1) length xs $= 0$
(ass2) z $\in$ ys

To Show: z $\notin$ subList xs ys

Then

(3)	`xs = []`	from (ass1) and **L1**.
(4)	subList xs ys $=$ []	from (3) and def of `subList`.
(5)	z $\notin$ subList xs ys	from (4) and def of `elem`.

Note that we did not use (ass2) for this case.

Inductive step

Take $k : \mathbb{N}$ arbitrary.
Ind. Hyp: $\forall$xs: [a].$\forall$ys: [a].$\forall$z : a. [length xs $= k$ $\rightarrow$ (z $\in$ ys $\rightarrow$ z $\notin$ subList xs ys)]
To show: $\forall$xs: [a].$\forall$ys: [a].$\forall$z : a. [length xs $= k+1$ $\rightarrow$ (z $\in$ ys $\rightarrow$ z $\notin$ subList xs ys)]

Take arbitrary `xs: [a]`,`ys: [a]`, and `z:a`.

Assume
(ass1) length xs $= k + 1$
(ass2) z $\in$ ys

To Show: z $\notin$ subList xs ys

From (ass1) and **L2**, we obtain that exist values `v`:`a` and `vs`: `[a]`, so that
(3) `length vs` $= k$
(4) `xs` $=$ `v:vs`

Then (5) `z` $\notin$ `subList vs ys` by (3), (ass2) and **Ind Hyp**

Continue by case analysis over whether `v` $\in$ `ys` - *see next two slides.*

1st Case: `v` $\in$ `ys`

Then, (6) `subList (v:vs) ys` $=$ `subList vs ys` by case & def `subList`

(7) `z` $\notin$ `subList xs ys` by (6), (5) & (4). **2nd Case:** `v` $\notin$ `ys`

Then (6) `subList (v:vs) ys` $=$ `v:(subList vs ys)` by case and definition of `subList`

(7) `z` $\neq$ `v` (ass2) & case.

(8) `z` $\notin$ `v:(subList vs ys)` by (7) , (5) and definition of `elem`.

(9) `z` $\notin$ `subList xs ys` (6), (5) & (4).

Slide 92

1st approach – conclusions

- It is *possible* to reason about lists using math. induction. But this reasoning is *indirect*: In particular, `length` is unrelated to P.
- The math. induct based proof requires lemmas:
  L1: $\forall$`xs`: `[a]`. [`length xs` $= 0 \rightarrow$ `xs` $=$ `[]`]
  L2: $\forall$`xs`: `[a]`.$\forall k$:$\mathbb{N}$.
  [`length xs` $= k+1 \rightarrow \exists$`v`:`a`.$\exists$`vs`: `[a]`.[`length vs` $= k \wedge$ `xs` $=$ `v:vs`]].
  How do we prove **L1** and **L2**?
- Therefore, we need something better.
- In math. induct. step, we argue that a property is "inherited" from "predecessor" to their "successor". E.g., 4 is a "successor" of 3.
- Can we generalize the concept of "predecessor" and "successor"? Can we see `10:44:33:[]` as successor of `44:33:[]`?

Slide 93

Structural Induction Principle over lists

For any type T, and $P \subseteq$ [T]:

$$P(\texttt{[]}) \ \wedge\ \forall \texttt{vs}{:}\texttt{[T]}.\forall \texttt{v}{:}\texttt{T}.[\,P(\texttt{vs}) \rightarrow P(\texttt{v}:\texttt{vs})\,] \ \longrightarrow\ \forall \texttt{xs}{:}\texttt{[T]}.P(\texttt{xs})$$

Slide 94

Structural induction principle for lists applied to the subList property

$$
\begin{array}{l}
\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\texttt{z} \in \texttt{ys} \ \rightarrow \texttt{z} \notin \texttt{subList}\ \texttt{[]}\ \texttt{ys}\,] \\
\quad \wedge \\
\forall \texttt{vs}:\texttt{[a]}.\forall \texttt{v}:\texttt{a}.\ [\\
\qquad \forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\ \texttt{z} \in \texttt{ys} \ \rightarrow \texttt{z} \notin \texttt{subList}\ \texttt{vs}\ \texttt{ys}\ ] \\
\qquad\quad \rightarrow \\
\qquad \forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\texttt{z} \in \texttt{ys} \ \rightarrow \texttt{z} \notin \texttt{subList}\ (\texttt{v:vs})\ \texttt{ys}\ ] \\
\qquad\quad] \\
\longrightarrow \\
\forall \texttt{xs}:\texttt{[a]}.\forall \texttt{ys}:\texttt{[a]}.\forall \texttt{z}:\texttt{a}.[\ \texttt{z} \in \texttt{ys} \ \rightarrow \texttt{z} \notin \texttt{subList}\ \texttt{xs}\ \texttt{ys}\ ]
\end{array}
$$

Slide 95

Proving the `subList` property by str. ind. - schema

Base Case

To Show $\forall ys:[a].\forall z:a.[\ z \in ys\ \rightarrow z \notin \texttt{subList}\ []\ ys\ ]$

...

Inductive Step

Take arbitrary `v:a`, `vs:[a]`

Inductive Hypothesis:

$\forall ys:[a].\forall z:a.[\ y \in ys\ \rightarrow y \notin \texttt{subList}\ \texttt{vs}\ ys\ ]$

To Show: $\forall ys:[a].\forall z:a.[\ z \in ys\ \rightarrow z \notin \texttt{subList}\ (\texttt{v:vs})\ ys\ ]$

...

Compare with the proof schema for proof by mathem. induction on the length of the lists. The one based on structural induction is more succinct.

Slide 96

Base Case

Base Case

To Show: $\forall ys:[a].\forall z:a.[\ z \in ys\ \rightarrow z \notin \texttt{subList}\ []\ ys\ ]$

Take arbitrary `ys: [a]`, and `z: a`.

Assume (ass1) $z \in ys$

To Show: $z \notin \texttt{subList}\ []\ ys$

We have

(1)	$\texttt{subList}\ []\ ys\ = []$	by def. of `subList`
(2)	$z \notin \texttt{subList}\ []\ ys$	by (1) and def. of `elem`

Slide 97

Inductive step

Inductive Step

Take arbitrary `v:a`, `vs:[a]`

Inductive Hypothesis: $\forall$`ys:[a]`.$\forall$`z : a`. [`z` $\in$ `ys` $\rightarrow$ `z` $\notin$ `subList vs ys`]

To Show: $\forall$`ys:[a]`.$\forall$`z : a`. [`z` $\in$ `ys` $\rightarrow$ `z` $\notin$ `subList (v:vs) ys`]

Take arbitrary `ys:[a]`, and `z:a`.

Assume

(ass1) `z` $\in$ `ys`

To Show: `z` $\notin$ `subList (v:vs) ys`

Then (1) `z` $\notin$ `subList vs ys` by (ass1) and Ind Hyp

Continue on next slide

Slide 98

Inductive step - continued

What we have so far:

(ass1) `z` $\in$ `ys`

(1) `z` $\notin$ `subList vs ys`

To Show: `z` $\notin$ `subList (v:vs) ys`.

1st Case: `v` $\in$ `ys`

Then

(2) `subList (v:vs) ys = subList vs ys` by case & def `subList`

(3) `z` $\notin$ `subList (v:vs) ys` by (1) & (2).

2nd Case: `v` $\notin$ `ys`

Then

(2) `subList (v:vs) ys = v:(subList vs ys)` case & def `subList`

(3) `z` $\neq$ `v` (ass1) & case.

(4) `z` $\notin$ `subList (v:vs) ys` by (1), (2), (3) and & def. `elem`.

Conclusion Reasoning about lists through structural induction is natural.

Our plan

Discuss one more example reasoning about lists.

Reason about any user defined Haskell data type.

Slide 99

another example, rev

Consider the following Haskell function:

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

Consider the assertion SPEC_1:

$\forall \texttt{xs} : \texttt{[a]}.\forall \texttt{ys} : \texttt{[a]}.\ \texttt{rev (xs++ys)} = \texttt{(rev ys) ++ (rev xs)}$

We will prove SPEC_1 by structural induction over `xs`.

Lists Struct. induction principle applied to SPEC_1

$$\forall \texttt{ys} : \texttt{[a]}.\ \texttt{rev}(\texttt{[]}\texttt{++}\texttt{ys}) = (\texttt{rev ys})\ \texttt{++}\ (\texttt{rev []})$$

$$\wedge$$

$$\forall \texttt{vs} : \texttt{[a]}.\forall \texttt{v} : \texttt{a}.\ [$$

$$\forall \texttt{ys} : \texttt{[a]}.\ \texttt{rev}\ (\texttt{vs}\texttt{++}\texttt{ys}) = (\texttt{rev ys})\texttt{++}(\texttt{rev vs})$$

$$\rightarrow$$

$$\forall \texttt{ys} : \texttt{[a]}.\ \texttt{rev}\ ((\texttt{v:vs})\texttt{++}\texttt{ys}) = (\texttt{rev ys})\texttt{++}(\texttt{rev (v:vs)})$$

$$]$$

$$\longrightarrow$$

$$\forall \texttt{xs} : \texttt{[a]}.\forall \texttt{ys} : \texttt{[a]}.\ \texttt{rev}\ (\texttt{xs}\texttt{++}\texttt{ys}) = (\texttt{rev ys})\ \texttt{++}\ (\texttt{rev xs})$$

Slide 100

Proving SPEC_1 by str. ind. - schema

Base Case

To Show $\forall \texttt{ys} : \texttt{[a]}.\ \texttt{rev}(\texttt{[]}\texttt{++}\texttt{ys}) = (\texttt{rev ys})\ \texttt{++}\ (\texttt{rev []})$

...

Inductive Step

Take arbitrary $\texttt{z:a}$, $\texttt{zs:[a]}$.

Inductive Hypothesis:

$$\forall \texttt{ys} : \texttt{[a]}.\texttt{rev}(\texttt{zs}\texttt{++}\texttt{ys}) = (\texttt{rev ys})\ \texttt{++}\ (\texttt{rev zs})$$

To Show: $\forall \texttt{ys} : \texttt{[a]}.\texttt{rev}((\texttt{z:zs})\texttt{++}\texttt{ys}) = (\texttt{rev ys})\ \texttt{++}\ (\texttt{rev (z:zs)})$

...

Slide 101

Slide 102

Lists Lemmas

We use the following properties of ++ and : for arbitrary `u:a`, `us,vs,ws:[a]`:

(A) `us ++ []` = `us`
(B) `[] ++ us` = `us`
(C) `(u:us) ++ vs` = `u:(us ++ vs)`
(D) `(us ++ vs) ++ ws` = `us ++ (vs ++ ws)`

Slide 103

Base Case

Base Case

To Show ∀`ys:[a]. rev([] ++ ys)` = `(rev ys) ++ (rev [])`

Take arbitrary `ys:[a]`.

Then, we have

`rev ([] ++ ys)`

= `rev ys` by (B)

= `(rev ys) ++ []` by (A)

= `(rev ys) ++ (rev [])` by definition of `rev`

Slide 104

Inductive Step

Inductive Step

Take arbitrary `z:a`, `zs:[a]`.

Ind. Hypo: $\forall$`ys : [a]. rev (zs++ys) = (rev ys) ++ (rev zs)`

To show:

$\forall$`ys:[a].    rev ((z:zs) ++ ys) = (rev ys) ++ (rev (z:zs))`

Take arbitrary `ys:[a]`.

Then, we have:

```
rev ((z:zs) ++ ys)
 =  rev (z:(zs ++ ys))              by (C)
 =  (rev (zs ++ ys)) ++ [z]         by definition of rev
 =  ((rev ys) ++ (rev zs)) ++ [z]   by induction hypothesis
 =  (rev ys) ++ ((rev zs) ++ [z])   by (D)
 =  (rev ys) ++ (rev (z:zs ))       by definition of rev
```


Slide 105

Why does induction over lists work?

Intuitively, and informally

- Base case: P(`[]`) holds.
- Inductive Step: P(`xs`) $\rightarrow$ P(`x:xs`) for all `x : a`, `xs : [a]`.
  - P(`[]`) $\rightarrow$ P(`x:[]`).
    Therefore, P(`x:[]`) holds for all `x : a`.
  - P(`x:[]`) $\rightarrow$ P(`y:x:[]`).
    Therefore, P(`y:x:[]`) holds for all `y, x : a`.
  - P(`y:x:[]`) $\rightarrow$ P(`z:y:x:[]`).
    Therefore, P(`z:y:x:[]`) holds for all `z, y, x : a`.
  - ...

And so, P(`xs`) holds for all `xs : [a]`.

Note that with induction over lists, every element has an infinite number of successors, (eg `3:[]` and `4:[]` and `55:[]` are all successors of `[]`, while in induction over numbers, eavery element has only one *direct* successor, (eg 3 is the unique successor of 2).

Slide 106

Induction over arbitrary Haskell data structures

```
data Nat = Zero | Succ Nat
```

$P(\texttt{Zero}) \ \wedge\ \forall \texttt{n}:\texttt{Nat}.[\, P(\texttt{n}) \to P(\texttt{Succ n})\,] \ \longrightarrow\ \forall \texttt{n}:\texttt{Nat}.P(\texttt{n})$

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

$P(\texttt{Empty}) \ \wedge\ \forall \texttt{t1,t2}:\texttt{Tree T}.\forall \texttt{x}:\texttt{T}.[\, P(\texttt{t1}) \wedge P(\texttt{t2}) \to P(\texttt{Node t1 x t2})\,]$
$\longrightarrow\ \forall \texttt{t}:\texttt{Tree T}.P(\texttt{t})$

```
data BExp = Tr | Fl | BNt BExp | BAnd BExp BExp
```

$P(\texttt{Tr}) \ \wedge\ P(\texttt{Fl}) \ \wedge\ \forall \texttt{b}:\texttt{BExp}.[\, P(\texttt{b}) \to P(\texttt{BNt b})\,] \ \wedge$
$\forall \texttt{b1,b2}:\texttt{BExp}.[\, P(\texttt{b1}) \wedge P(\texttt{b2}) \to P(\texttt{BAnd b1 b2})\,] \ \longrightarrow\ \forall \texttt{b}:\texttt{BExp}.P(\texttt{b})$

Structural Induction for different kinds of trees

Write the struct. ind. principles for $P \subseteq$ `Tree Int`, and for $Q \subseteq$ `Tree a`, and for $R \subseteq$ `Tree [a]`:

- $P(\texttt{Empty}) \wedge$
  $\forall \texttt{t1,t2:Tree Int}.\forall \texttt{i:Int}.[\, P(\texttt{t1}) \wedge P(\texttt{t2}) \to P(\texttt{Node t1 i t2})]$
  $\longrightarrow \quad \forall \texttt{t:Tree Int}.\ P(\texttt{t})$
- $Q(\texttt{Empty}) \wedge$
  $\forall \texttt{t1,t2:Tree a}.\forall \texttt{x:a}.[\, Q(\texttt{t1}) \wedge Q(\texttt{t2}) \to Q(\texttt{Node t1 x t2})\,]$
  $\longrightarrow \quad \forall \texttt{t : Tree a}.\ Q(\texttt{t})$
- $R(\texttt{Empty}) \wedge$
  $\forall \texttt{t1,t2:Tree [a]}.\forall \texttt{as:[a]}.\ [\ R(\texttt{t1}) \wedge R(\texttt{t2}) \to R(\texttt{Node t1 as t2})\ ]$
  $\longrightarrow \quad \forall \texttt{t : Tree [a]}.\ R(\texttt{t})$

Comment But actually, given

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

Do we have

$$[\ \ P(\texttt{Empty})\ \wedge\ \forall \texttt{t1,t2:Tree T}.\forall \texttt{x:T}.[\,P(\texttt{t1}) \wedge P(\texttt{t2}) \to P(\texttt{Node t1 x t2})]\,] \\ \to \\ \forall \texttt{t:Tree T}.P(\texttt{t})$$

or, should we have, instead

$$[\ \ P(\texttt{Empty})\ \wedge\ \forall \texttt{t1,t2:Tree T}.[\,P(\texttt{t1}) \wedge P(\texttt{t2})\ \to\ \forall \texttt{x:T}.P(\texttt{Node t1 x t2})]\ \ ] \\ \to \\ \forall \texttt{t:Tree T}.P(\texttt{t})$$

Since the two assertions are equivalent, the question actually does *not* arise.

Slide 108

Two Proof strategies

We will conclude this section by discussing two strategies that appear often when writing proofs.

Slide 109

... When the proof does not go through ...

Consider the Haskell function `sum`, and its tail-recursive version, `sum_tr`

```
sum :: [Int] -> Int
sum [] = 0
sum i:is = i + sum is

sum_tr :: [Int] -> Int -> Int
sum_tr [] k = k
sum_tr (i:is) k = sum_tr is (i+k)
```

Prove that

$$\forall \texttt{is}:\texttt{[Int]}.\ \texttt{sum is} = \texttt{sum\_tr is 0}$$

Proving $\forall$`is:[Int]. sum is = sum_tr is 0` **by induction on** `is` **- base case**

Base Case

To Show `sum [] = sum_tr [] 0`

	`sum []`		
We have	=	`0`	by def. of `sum`
	=	`sum_tr [] 0`	by def. of `sum_tr`

The inductive step is in the slides.

Slide 110

Inductive step for $\forall$`is:[Int]. sum is = sum_tr is 0`

Take `i: Int` and `is:[Int]` arbitrary.

Inductive Hypothesis: `sum is = sum_tr is 0`

To Show: `sum (i:is) = sum_tr (i:is) 0`

By applying definitions we obtain

`sum (i:is)`	=	`i + sum is`	by def. of `sum`
	=	`i + sum_tr is 0`	by ind. hypo.
		???	???
	=	`sum_tr is i`	???
	=	`sum_tr (i:is) 0`	by def. of `sum_tr`

When the proof does not go through ... - two approaches

We cannot prove

$$\forall \texttt{is:[Int]. sum is} = \texttt{sum\_tr is 0}$$

directly by induction.
Such situations appear very often in proofs.

There are two approaches to solving such problems:

- 1st Strategy: Invent an Auxiliary Lemma
- 2nd Strategy: Strengthen the original property

1st Strategy: Invent an auxiliary lemma

We revisit the Inductive step from the proof of
$\forall$`is:[Int]. sum is` $=$ `sum_tr is 0`.
Inductive step
Take `i: Int`, and `is:[Int]` arbitrary.
Inductive Hypothesis `sum is` $=$ `sum_tr is 0`
To Show: `sum (i:is)` $=$ `sum_tr (i:is) 0`
We have

`sum (i:is)`	$=$	`i + sum is`	by def. of `sum`
	$=$	`i + (sum_tr is 0)`	by ind. hypo.
	$=$	`sum_tr is (i+0)`	by Lemma ZZ
	$=$	`sum_tr (i:is) 0`	by def. of `sum_tr`

Lemma ZZ

$$\forall \texttt{i:Int}.\forall \texttt{k:Int}.\forall \texttt{is:[Int]}.\ \texttt{i + (sum\_tr is k)} = \texttt{sum\_tr is (i+k)}$$

2nd Strategy: Proving a stronger property

Rather than

$$(*) \qquad \forall \texttt{is:[Int]}.\ \texttt{sum is} = \texttt{sum\_tr is } 0$$

We will prove a stronger property

$$(**) \qquad \forall \texttt{k:Int}.\forall \texttt{is:[Int]}.\ k + (\texttt{sum is}) = \texttt{sum\_tr is k}$$

Proving $\forall \texttt{is:[Int]}.\forall \texttt{k:Int}.\ \texttt{k} + (\texttt{sum is}) = \texttt{sum\_tr is k}$ by induction on is - base case

Base Case

To Show: $\forall \texttt{k:Int}.\ \texttt{k} + (\texttt{sum []}) = \texttt{sum\_tr [] k}$

Take k:Int arbitrary.

To Show: $\texttt{k} + \texttt{sum []} = \texttt{sum\_tr [] k}$

We have

k + (sum [])	=	k	by def. of sum, and arithm
	=	sum_tr [] k	by def. of sum_tr

Proving $\forall$`is:[Int]`.$\forall$`k:Int`. `k` $+$ `(sum is)` $=$ `sum_tr is k` by induction on `is` - ind step

Note: Because use of quantifiers is subtle, we distinguish all variables:

Inductive Step

Take `i:Int` and `is:[Int]` arbitrary.

Inductive Hypothesis: $\forall$`m` : `Int`. `m+(sum is)` $=$ `sum_tr is m`

To Show: $\forall$`n` : `Int`. `n+(sum (i:is))` $=$ `sum_tr (i:is) n`

Take `n:Int` arbitrary.

To Show: `n+sum (i:is)` $=$ `sum_tr (i:is) n`

`n+ (sum (i:is))`	$=$	`(n+i) + (sum is)`	by def. of `sum`, arithm
	$=$	`sum_tr is (n+i)`	by ind. hyp.
	$=$	`sum_tr (i:is) n`	by def. of `sum_tr`

Note: We instantiated the induction hypothesis by replacing `m` by `n+i`.

The limits of induction Fermat stated and did not prove that:

$$x^n + y^n = z^n$$

has no positive integer solutions for $n \geq 3$

The format of the theorem suggests the use of induction.

- **Base case:** it has been proved that $x^3 + y^3 = z^3$ has no solutions.
- **Inductive Step:** Show that if $x^k + y^k = z^k$ has no solutions, then $x^{k+1} + y^{k+1} = z^{k+1}$ has no solutions.
- Nobody has been able to do that.
- A proof of the theorem was developed recently, and is *not* based on induction.

Slide 116

More Induction Principles – mutual recursion

```
data T = C1 [Int] | C2 Int T
```

$$\forall \texttt{is}:\texttt{[Int]}.\, P(\texttt{C1 is}) \quad \wedge \quad \forall \texttt{i}:\texttt{Int}.\forall \texttt{t}:\texttt{T}.[\, P(\texttt{t}) \rightarrow P(\texttt{C2 i t})\,]$$
$$\longrightarrow$$
$$\forall \texttt{t}:\texttt{T}.P(\texttt{t})$$

```
data Reds   =  BaseR | Red Greens
data Greens =  BaseG | Green Reds
```

$$P(\texttt{BaseR}) \quad \wedge \quad \forall \texttt{g}:\texttt{Greens}.[\, Q(\texttt{g}) \rightarrow P(\texttt{Red g})\,] \quad \wedge$$
$$Q(\texttt{BaseG}) \quad \wedge \quad \forall \texttt{r}:\texttt{Reds}.[\, P(\texttt{r}) \rightarrow Q(\texttt{Green r})\,]$$
$$\longrightarrow$$
$$\forall \texttt{r}:\texttt{Reds}.P(\texttt{r}) \quad \wedge \quad \forall \texttt{g}:\texttt{Greens}.Q(\texttt{g})$$

Slide 117

The cactus revisited

```
data Cactus = Root Tree
data Tree   = Leaf | Node Trees
data Trees  = Empty | Cons Tree Trees
```

Express an inductive principle for `Tree`.

$$P(\texttt{Leaf}) \quad \wedge$$
$$\forall \texttt{ts}:\texttt{Trees}.\,[\, Q(\texttt{ts}) \rightarrow P(\texttt{Node ts})\,] \quad \wedge$$
$$Q(\texttt{Empty}) \quad \wedge$$
$$\forall \texttt{t}:\texttt{Tree}.\forall \texttt{ts}:\texttt{Trees}.\ [\, P(\texttt{t}) \wedge Q(\texttt{ts}) \rightarrow Q(\texttt{Cons t ts})\,]$$
$$\longrightarrow$$
$$\forall \texttt{t}:\texttt{Tree}.P(\texttt{t}) \quad \wedge \quad \forall \texttt{ts}:\texttt{Trees}.Q(\texttt{ts})$$

A slightly weaker inductive principle is the following: