Lecture 9 – Correctness

Violet Ka I Pun

violet@ifi.uio.no

Reasoning about correctness

- Recursive programming and inductive datatypes
- Securing termination
- Inductive proof of correctness

The approach: “divide and conquer”

1. Which datatype should be used for recursive solution of the problem P?

INDUCTIVE...

Given an instance n of P:

2. What should be done when n is the base case?
3. How to construct the solution $P(n)$ based on the solutions $P(m_i)$ for instances $m_i < n$?

- $\mathrm{P(n)} = 0 + 1 + 2 + ... + n = \sum_{i=0}^{n} i$
- Recursion over n (natural numbers)
- base case, $n = 0$, the value: $P\ 0 = 0$
- induction step, $+1$:
  $P(n) = n + P(n-1)$

Recursion invariant = induction hypotheses

Sum all numbers from 0 to n:

– pre-condition: input $n \geq 0$

– post-condition: output $= \sum_{i=1}^{n} i$

invariant: `sm n` = 0+1+...+n, when $n >= 0$

sum 0 = 0 — sum 0 = 0 is correct

sum x = x + sum(x-1) — assume: sum(x-1) = (0+1+...+x-1)

then, sum x = x + sum(x-1) = (0+1+...+x-1) + x is correct

Recursion invariant = induction hypotheses

Let us guess that, for all n: $sum(n) = \frac{n}{2} * (n+1)$ – call it E(n), i.e..:

– pre-condition: input n ≥ 0

– post-condition: output $= \frac{n}{2} * (n+1)$

invariant: `sum n` – satisfies E(n)

$$
\begin{array}{lll}
\text{sum 0} & = & 0 = \frac{0}{2} * (0+1) \quad \text{sum 0 satisfies E(0)} \\
\text{sum x} & = & \text{x + sum(x-1)} \quad \text{assume: sum(x-1) satisfies E(n-1),} \\
 & & \qquad \text{i.e., } sum(x-1) = \frac{x-1}{2} * x
\end{array}
$$

$$
\begin{array}{llll}
\text{then:} & \text{sum x} & = & \text{x + sum (x-1)} \qquad (IH) \\
 & & = & x + \frac{x-1}{2} * x \\
 & & = & \frac{2}{2} * x + \frac{x-1}{2} * x = \frac{2+x-1}{2} * x \\
 & & = & \frac{x+1}{2} * x = \frac{x}{2} * (x+1)
\end{array}
$$

We can now implement `sum` much more effectively:

sum x = x * (x+1) / 2.

Inductive proof

1. Write clearly the property to be proven: $\forall n : E(n)$.
2. Prove all base cases (n=0, but in general can be more)
3. Imagine that you choose an arbitrary (no long base) element k:
   you can assume IH: for all $j < k, E(j)$, and
   base on that, you show E(k).
4. If you can show that, it is then true that $\forall n : E(n)$.
5. Often, one assumes IH holds for k, and proves E(k+1)
6. The ordering $<$ refers to the underlying ordering of the data type, determined by its inductive definition.

The approach: “divide and conquer”

1. Which datatype should be used for recursive solution of the problem P?

INDUCTIVE...

Given an instance n of P:

2. What should be done when n is the basic case?
3. How to construct the solution $P(n)$ based on the solutions $P(m_i)$ for instances $m_i < n$?

- rev xs = reverse the input list
- Recursion over the length of the list
- basis: rev [] = []
- rev (x:xs) = rev xs ++ [x]

Recursion invariant = induction hypotheses

reverse a list

invariant: `rev xs` – return the reverted xs

$$\begin{array}{rcl} \text{rev } [] & = & [] \\ \text{rev } (x:xs) & = & \text{rev } xs \mathbin{++} [x] \end{array}$$

rev [] = [] is correctly reversed

assume: rev xs reverses xs

that is, if $xs = x_1 : x_2 : \ldots : [x_n]$, then $rev\ xs = x_n : \ldots : x_2 : [x_1]$

$$\begin{array}{rcll} \text{therefore: rev } (x:xs) & = & \text{rev } (x : x_1 : \ldots : [x_n]) & \\ & = & \text{rev } (x_1 : \ldots : [x_n]) \mathbin{++} [x] & \textit{(def. of } \texttt{rev}\textit{)} \\ & = & x_n : \ldots : [x_1] \mathbin{++} [x] & \textit{(IH)} \\ & = & x_n : \ldots : x_1 : [x] & \textit{(def. of } \texttt{(++)}\textit{, } \texttt{(:)}\textit{)} \end{array}$$

– correctly reversed

Recursion invariant = induction hypotheses

for all lists as++bs: rev (as++bs) = (rev bs) ++ (rev as)

Induction on... as:

invariant: `rev (as++bs) = (rev bs) ++ (rev as)`

$$\begin{aligned}
\text{rev } [] &= [] \\
\text{rev } (x{:}xs) &= \text{rev } xs \mathbin{++} [x]
\end{aligned}$$

$$\begin{aligned}
\text{rev } ([]\mathbin{++}bs) &= \text{rev } bs && \textit{(def. of (++))} \\
&= (\text{rev } bs) \mathbin{++} [] && \textit{(def. of (++))} \\
&= (\text{rev } bs) \mathbin{++} (\text{rev } []) && \textit{(def. of rev)}
\end{aligned}$$

assume the invariant for recursive call:

$$\begin{aligned}
\text{then: rev } ((a{:}as)\mathbin{++}bs) &= (\text{rev } a{:}(as\mathbin{++}bs)) && \textit{(def. of (++))} \\
&= (\text{rev } (as\mathbin{++}bs)) \mathbin{++} [a] && \textit{(def. of rev)} \\
&= (\text{rev } bs) \mathbin{++} (\text{rev } as) \mathbin{++} [a] && \textit{(IH)} \\
&= (\text{rev } bs) \mathbin{++} (\text{rev } a{:}as) && \textit{(def. of rev)}
\end{aligned}$$

reverse the input list – tail recursive:

for all lists xs: rev1 xs = rev xs

rev1 xs = revh xs []	
revh [] r = r	rev [] = []
revh (x:xs) r = revh xs (x:r)	rev (x:xs) = rev xs ++ [x]

$$\text{rev1 []} = \text{revh [] []} \quad = \quad \text{[]} = \text{rev []} \qquad \textit{(def. of } \texttt{rev1, revh, rev}\textit{)}$$

$$\text{rev1 (x:xs)} = \text{revh (x:xs) []} \quad = \quad \text{revh xs [x]} \ldots \qquad \textit{(def. of } \texttt{rev1, revh}\textit{)}$$

$$\text{Lemma: revh xs ls} \quad = \quad \text{(rev xs) ++ ls}$$

$$\text{revh [] ls} = \text{ls} \quad = \quad \text{[] ++ ls} = \text{(rev []) ++ ls}$$

(def. of `revh`*, (++),* `rev`*)*

$$\text{revh (y:ys) ls} = \text{(revh ys) (y:ls)} \quad = \quad \text{(rev ys) ++ (y:ls)} \qquad \textit{(def. of } \texttt{revh}\textit{, IH)}$$

$$= \quad \text{(rev ys) ++ [y] ++ ls}$$

(def. of `(:)`*,* `(++)`*)*

$$= \quad \text{(rev (y:ys)) ++ ls} \qquad \textit{(def. of } \texttt{rev}\textit{)}$$

$$\ldots \text{ revh xs [x]} \quad = \quad \text{(rev xs) ++ [x]} = \text{rev (x:xs)}$$

MyBool

- data MB = T | F | And MB MB | Or MB MB | Not MB
  ev :: MB -> MB

base: ev(T) = T   ev(F) = F

ind.: ev(And x y) | (ev x == T) = (ev y) | otherwise = F

ev(Or x y) | (ev x == T) = T | otherwise (ev y)

ev(Not T) = F   ev(Not F) = T

!! For all x::MB, E(x), namely: ev(x)=T or ev(x)=F.

base: ev(T) = T and ev(F) = F

Ind.: ev(And x y) | (ev x == T) = ev y | otherwise = F

– we can assume that E(x) and E(y)

– and since either ev(And x y)=ev y or =F, then E(And x y)

ev(Or x y) | (ev x == T) = T | otherwise = (ev y)

– we can assume that E(x) and E(y)

– and since either ev(Or x y)=T or =ev y, then E(Or x y)

ev(Not x) – we can assume that E(x) and then also E(Not x).

Example – Proof of a false hypothese fails...

- nub [] = []
  nub (x:xs) = x : filter (x/=) (nub xs)

? For all lists xs, ys: nub (xs ++ ys) = (nub xs) ++ (nub ys)

base: nub ([] ++ ys) = nub ys = [] ++ (nub ys) = (nub []) ++ (nub ys)

ind.: nub ((x:xs)++ys) To show: = (nub (x:xs)) ++ (nub ys)

= nub (x : (xs++ys)) = x : filter (x/=) (nub (xs++ys))

*(def. of (++), **nub**)*

= x : filter (x/=) ((nub xs) ++ (nub ys)) *(IH)*

= x : (filter (x/=) (nub xs)) ++ (filter (x/=) (nub ys)) *(distributivity)*

= (nub (x:xs)) ++ (filter (x/=) (nub ys)) *(def. of **nub**)*

- If you can prove something by induction – it is then true!
  proved: nub(x:xs++ys) = (nub(x:xs))++(filter (x/=)(nub ys))
- If you fail to prove something by induction – it can still be true!
  we did not prove: nub (xs ++ ys) /= (nub xs) ++ (nub ys)

but find a counter-example: nub [1,2,1,2] = [1,2] /= [1,2,1,2]

= (nub [1,2]) ++ nub([1,2])

A bit larger example

```
data  BT a = Emp | Node (BT a) a (BT a)

 ins:: Ord a => a -> (BT a) -> (BT a)
       ins i Emp = Node Emp i Emp
       ins i (Node l x r) |      i < x = Node (ins i l) x r
                          | otherwise = Node l x (ins i r)

inord:: BT a -> [a]
       inord Emp = []
       inord (Node l x r) = (inord l) ++ [x] ++ (inord r)

       preord (Node l x r) = [x] ++ (preord l) ++ (preord r)
       postord (Node l x r) = (postord l) ++ (postord r) ++ [x]

 sort:: Ord a => [a] -> [a]
       sort ls = inord (foldr ins Emp ls)
```

We want to prove...

(ins x t): if t is a search tree, then (*ins* x t) is a search tree (for every x) which contains the same elements as t plus x

(inord t): if t is a search tree, then the list (inord t) is sorted

foldr: strictly speaking, we should also show that (foldr ins Emp ls) inserts all the elements from *ls* into a tree which becomes a search tree

but we are content that a call (foldr ins Emp ls), where

$$\text{foldr ins Emp } [x_1, x_2 ... x_n] = \text{ins } x_1 \text{ (ins } x_2 \text{ (...(ins } x_n \text{ Emp)...))},$$

performs a sequence of calls to *ins*, determined by the list *ls*. Thus, the correctness of (*sort ls*) follows from the proof that (*ins* x t) preserves the search tree property while *inord* gives a sorted list.

stree:: Ord a => BT a -> Bool
stree Emp
stree (Node l x r) = $\forall$ z: z $\in$ l $\rightarrow$ z $<$ x &
z $\in$ r $\rightarrow$ z $\geq$ x & stree l & stree r

- stree (Node l x r) = $\forall$ z: z $\in$ l $\rightarrow$ z $<$ x & z $\in$ r $\rightarrow$ z $\geq$ x & stree l & stree r

A. stree (ins i Emp) = stree (Node Emp i Emp) = True

B. stree (Node l x r) and IH: stree (ins i l) & stree (ins i r) $\Rightarrow$

stree (ins i (Node l x r)) =

i<x: stree (Node (ins i l) x r) = (c) stree (ins i l) & stree r & $\forall$ z: (a) z $\in$ (ins i l) $\rightarrow$ z $<$ x &(b) z $\in$ r $\rightarrow$ z $\geq$ x

(a) since $\forall$ z $\in$ (ins i l): z = i or z $\in$ l, and we are given: i $<$ x and stree (Node l x r), which $\Rightarrow$ $\forall$ z $\in$ (ins i l) $\rightarrow$ z $<$ x

(b) stree (Node l x r), then z $\in$ r $\rightarrow$ z $\geq$ x

i$\geq$x: stree (Node l x (ins i r)) = (c) stree l & stree (ins i r) & $\forall$ z: (a) z $\in$ l $\rightarrow$ z $<$ x & (b) z $\in$ (ins i r) $\rightarrow$ z $\geq$ x

(a) stree (Node l x r), then z $\in$ v $\rightarrow$ z $<$ x

(b) since $\forall$ z $\in$ (ins i r): z = i or z $\in$ r, and we are given: i $\geq$ x and stree (Node l x r), which $\Rightarrow$ $\forall$ z $\in$ (ins i r) $\rightarrow$ z $\geq$ x

pre-condition: (stree t) $\Rightarrow$ post-condition: (sorted (inord t))

stree (Node l x r) = stree l & stree r &

$\forall$ z: z $\in$ l $\rightarrow$ z < x & z $\in$ r $\rightarrow$ z $\geq$ x

where sorted $[x_1, x_2, ..., x_n] = \forall 1 \leq i < j \leq n : x_i \leq x_j$

base: stree Emp = True and sorted (inord Emp) = sorted [] = True

IH: stree l $\Rightarrow$ sorted (inord l) and stree r $\Rightarrow$ sorted (inord r)

- stree (Node l x r) $\Rightarrow$ sorted (inord (Node l x r)) =

  sorted((inord l) ++ [x] ++ (inord r))

  sorted($x_1 ... x_{p-1}$ ++ [x] ++ $x_{p+1} ... x_n$)

so if p is the index of x, then the claim holds for indexes $x_1 ... x_{p-1}$ (sublist *inord l*) and $x_{p+1} ... x_n$ (sublist *inord r*)

also all $x_1 ... x_{p-1} \leq x$ and $x \leq$ all $x_{p+1} ... x_n$, and thus,

- for any arbitrary pair of indexes $1 \leq i < j \leq n$:

  $\forall 1 \leq i < j \leq p - 1 : x_i \leq x_j$ and $\forall i < p, \forall j \geq p \leq n : x_i < x_j$

  also $\forall p \leq i < j \leq n : x_i \leq x_j$

Inductive proof

data = Mine a = Base1 | ... | BaseN | Km1 (Mine a) |
Km2 (Mine a) (Mine a) | ... | Ka3 (Mine a) (Other a) | ...

- Set up the claim: for all x::Mine a, E(x).
- Prove all base conditions: E(Base1), E(Base2), ..., E(BaseN)
- For each constructor KmN – from Mine a:

– choose a name for each Mine-parameter m1,...,mK::Mine (which look like variable)

– assume E(m1),...,E(mK)

– and prove E(KmN m1 ... mK).

- For constructor with arguements of other types – like (z::Other a) in (Ka3 m z) – must prove the claim for all possible values z::Other a.