Revisiting Predicate Logic

LN chapters 3,4

STV 2024/25

Testing vs verification

Verification: here it means proving that a program will **always** behave correctly, as opposed to testing that only proves that some executions are correct.

- Verification is however undecidable (it cannot be generically automated).
- There are tools to assist us constructing proofs, or to automatically verify special cases (e.g. if the program has finite number of states) → outside our scope.
- More on such automation in Master courses e.g. Program Semantics & Verification.

Our learning goals

- To introduce you to **Hoare logic**: fundamental for imperative languages.
- To learn formulating your correctness arguments **formally**.
- To introduce you to some of the more advanced aspects such as the treatment of loops and program calls. This is useful for your later research, e.g. if you want to prove the correctness of your algorithm, or when your research involves development of a verification tool.

Overall plan

- Revisiting Predicate Logic, also introducing the notation we are going to use.
- Basic Hoare Logic
- More on proving the correctness of loops.
- Reasoning about more advanced language constructs, e.g. program calls, exceptions, OO.

How formal ?

- A proof is (really) *formal* if can be checked by a computer.
- Informal proof → can't be checked by a computer, easier to read by a human, but may be ambiguous.
- In this course we will train with *more* formal proofs: still human-readable, more *precise* than informal proof, but not machine formal.

Elements of our verification approach

- A simple programming language *uPL*
- Specifications are written in formulas of (1st order) predicate logic.
- Hoare logic to reduce program + specification to verification conditions (formulas in predicate logic).
- Use predicate logic to prove the verification conditions.

Overview

- Formulas
- Quantification
- Inference rules
- Proof
- Proofs involving quantification
- Some basic proof techniques:
  - Contradiction
  - Equational
  - Case split
  - Induction

Our Formula-language (LN Ch.2)

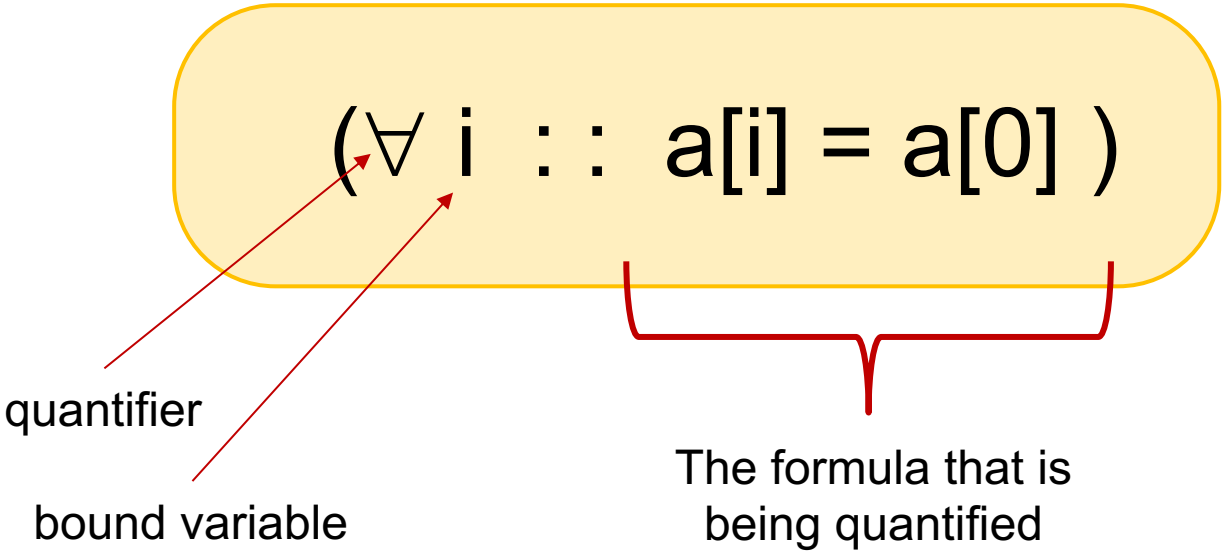
- Language elements:
  - Variables e.g. x,y,z ...
  - Constants e.g. true, false, 0, 1…
  - Arrays e.g a[i]
  - Operators e.g. +,-, <, =, /\ , $\Rightarrow$ ...
  - Quantifiers $\forall$ $\exists$,
- Types: bool, int, arrays of primitives. We usually leave types implicit in the formulas.
- Arrays are assumed to have infinite size.

From now on write your formulas, incl pre/post-condition in the predicate logic notation.

Don’t use in-code notation

(to avoid confusing yourself)

Quantified formula, basic form



It means: “for **all** i, a[i] = a[0] holds.” Implicitly i is of type int, as it is being used as an index of an array.

Quantified formula with domain

$$(\forall\ i\ :\ \mathbf{0 \le i < n}\ :\ a[i] = a[0]\ )$$

domain part : a formula to restrict the domain of the quantification

range part : the formula that is being quantified

It means: “for **all** i such that 0≤i<n, a[i] = a[0] holds.” Again, implicitly i is an int, as it is being used as an index of an array.

Domain part in quantified formula

- Definition of the form with domain part:

  - $(\forall x : \mathbf{P}\ x : Q\ x) \;=\; (\forall x :: \mathbf{P}\ x \Rightarrow Q\ x)$
  - $(\exists x : \mathbf{P}\ x : Q\ x) \;=\; (\exists x :: \mathbf{P}\ x \;/\backslash\; Q\ x)$

- Notice that the def. above implies :
  - $(\forall i : \mathbf{true} : a[i]>0) \;=\; (\forall i :: a[i]>0)$
  - $(\exists i : \mathbf{true} : a[i]>0) \;=\; (\exists i :: a[i]>0)$

Quantifying over “empty domain”

- Quantifying over “false” is also called quantifying over “empty domain” . Their meaning:
- $(\forall x : \text{false} : Q\ x)$

  $= (\forall x : : \text{false} \Rightarrow Q\ x)$

  $= (\forall x : : \text{true})$

  $= \text{true}$
- $(\exists x : \text{false} : Q\ x)$

  $= (\exists x : :\ \text{false}\ /\backslash\ Q\ x)$

  $= (\exists x : : \text{false})$

  $=\ \text{false}$

Scoping and Nesting

- A quantifier has a "scope" :

$$(\exists i : i>0 : b[i]) \quad /\backslash \quad \neg b[i]$$

  - "**Bounded** variable" e.g. i in the quantified formula above.
  - "**Free** variable" e.g. b and the i on the left in the above example.
- Quantification can also be "nested" :

$$(\forall i :: (\exists j :: a[j] > a[i]))$$

How do we prove our claims ?

- In logic we use inference/proof rules. Such a rule is usually shown in this form:

$$\frac{\text{premise}_1 \quad , \quad \text{premise}_2 \quad , \ldots}{\text{conclusion}}$$

- A proof is essentially a series of invocations of inference rules, that produces our claim from known facts and the given assumptions.

Some examples of inference rules

- Modus Ponens

$$\frac{P \quad , \quad P \Rightarrow Q}{Q}$$

- $\forall$ elimination ($\forall$ instantiation)

$$\frac{P\ a \quad , \quad (\forall x : P\ x : Q\ x)}{Q\ a}$$

Proof format (LN Ch.3)

- Stick to the proof format as in the LN (so that we have certainty when evaluating your work).
- To improve training, we deliberately make the format more explicit and also more verbose.
- The format will allow you to mix a deductive style and an equational style of reasoning in one proof.
  - Deductive reasoning: one way (from assumptions to conclusion)
  - Equational reasoning: bi-directional.
- Many of our example proofs will be "quite trivial", but remember that our goal is to train you in formal reasoning (of the correctness of your program). So your "mental process" is just as important.

Basic elements of our proof format

PROOF main

[A1:] $(\forall i : i>0 : a[i]=i)$

[A2:] $x > 10$

[G:] $a[x] > 10$

If successful this will prove the claim: **A1** $\wedge$ **A2** $\Rightarrow$ **G** (note the implication)

BEGIN

1. { follows from A2 } $x>0$
2. { $\forall$**-elim** on A1 using 1 } $a[x]=x$
3. { rewrite A2 with 2 } $a[x]>10$

END

Subproof and proof scope

PROOF main

...

4. …

5. { see subproof below } $0 \leq k \Rightarrow b[k]$

PROOF sub
[A:] $0 \leq k$
[G:] $b[k]$
…
… $k+1 > 0$
...

Suppose in the subproof you manage to derive $k+1>0$. Using this for further inference **within** the same subproof is ok.

7. { because "sub" says $k+1>0$ } $k+2>0$ X

Using facts inferred in a subproof as an argument in a higher level proof is not sound!

Introducing and eliminating quant.

- Eliminating $\forall$:

$$\frac{P\ a \quad , \quad (\forall x : P\ x : Q\ x)}{Q\ a}$$

- Introducing $\exists$:

$$\frac{P\ a \quad , Q\ a}{(\exists x : P\ x : Q\ x\ )}$$

- How about introducing $\forall$ and eliminating $\exists$??

Introducing ∀

PROOF main

[A:] (∀k: k≥0 : b[k])
[G:] (∀k: k>1: b[k])

1. { how ??} (∀k: k>1 : b[k])

> **PROOF** sub
> [A:] k > 1
> [G] b[k]
> 1. { follows from A } k≥0
> 2. {∀-elim on main.A using 1 } b[k]
> Success! (conclusion: k>1 ⇒ b[k])

We then could argue that since k is **unconstrained** in the parent proof, we can generalize the normal conclusion of this subproof to (∀k : k>1 : b[k]). But this is a bit error prone.

∀ Introduction through a subproof

PROOF main

[A:] (∀k: k≥0 : b[k])
[G:] (∀k: k>1: b[k])

1. { see subproof } (∀k: k>1: b[k])

> **PROOF** sub
> **ANY k**
> [A:] k > 1
> [G] b[k]
> 1. { follows from A } k≥0
> 2. {∀-elim on main.A using 1 } b[k]

ANY k marker at the start of a proof introduces a scope for k, namely limited within the proof. Within the proof, occurrences of k refers to this k and any **previous** assumption about this k **cannot** be used.
From such a proof we are allowed to conclude a **generalized** formula, in this case **(∀k: k>1: b[k])**

Proof by Contradiction

- Let's prove this claim:

$$a[i]>i \;\; /\backslash \;\; (\exists i : 0 \le i : a[i] = i)$$
$$\Rightarrow$$
$$\neg(\forall i : 0 \le i : a[i] > i)$$

- We'll prove this *by contradiction*. Based on this rule:

$$\frac{\neg Q \Rightarrow \text{false}}{Q}$$

Top level proof

PROOF main

[A1:] a[i]>i
[A2:] (∃i : 0≤i : a[i] = i)
[G:] ¬(∀i : 0≤i : a[i] > i)

BEGIN
1. { see subproof } (∀i : 0≤i : a[i] > i) ⇒ **false**
2. { rule of contradiction on 1 } ¬ (∀i : 0≤i : a[i] > i)
END

Eliminating ∃

PROOF main
[A1:] a[i]>i
[A2:] (∃i : 0≤i : a[i] = i)
[G:] ¬(∀i : 0≤i : a[i] > i)
BEGIN

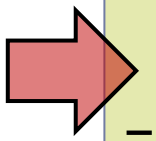
PROOF sub
[A:] (∀i : 0≤i : a[i] > i)
[G:] false
1. {∃ **elimination** on main.A2} **[SOME k]** 0≤k /\ a[k]=k
2. {∀ elimination on A using 1st conjunct of 1 } a[k] > k
3. { contradiction of 1 and 2 } false
END

....

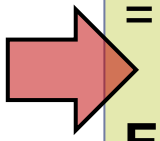
Eliminating ∃ introduces a **[SOME k]** marker which imposes a scope. From this point on, occurrences of k refers to this k and any **previous** assumption about this k **cannot** be used.

Equational proof

EQUATIONAL PROOF
[A:] 0 ≤ n
[D:] found n = (∃i : 0≤i<n : b[i])



found (n + 1)
= { def. found }
(∃i : 0 ≤ i < n+1 : b[i])
= { Domain Merge Thm A.4.16 justified by A }
(∃i : 0 ≤ i < n ∨ i=n : b[i])
= { Domain Split Thm A.4.12 }
(∃i : 0 ≤ i < n : b[i]) ∨ (∃i : i=n : b[i])
= { Quantification over Singleton Thm A.4.10 }
(∃i : 0 ≤ i < n : b[i]) ∨ **b[n]**
= { def. found }



found n ∨ b[n]
END

This equational proof proves
0 ≤n ⇒ **(found (n+1) = found n ∧ b[n])**,
with the given definition of “found n”.

Proof with case split

- Suppose that to prove Q, we identify N-cases, and we want to prove Q separately for each case.
- Based on this inference rule:

$$\frac{P_1 \vee P_2 \quad , \quad P_1 \Rightarrow Q \ , \ P_2 \Rightarrow Q}{Q}$$

- Example, prove this:

$$b[n] \wedge (\forall i : i<n : b[i]) \Rightarrow (\forall i : i<n+1 : b[i])$$

Top level proof of the example

PROOF main
[A1:] b[n]
[A2:] (∀i : i < n : b[i])
[G:] (∀i : i<n+1 : b[i])
BEGIN
1. { see the proof below } G

PROOF sub
ANY i
[A:] i < n+1
[G:] b[i]
BEGIN

1 { follows from A }	i<n ∨ i=n
2 { see subproof sub_1 }	i<n ⇒ b[i]
3 { see subproof sub_2 }	i=n ⇒ b[i]
4 { Case Split on 1,2,3 }	b[i]

END

END

One more example of ∀-intro and ∃-elimination

PROOF
[G:] (∀n: n **mod** 4 = 0 : n **mod** 2 = 0)
BEGIN
1. { see the proof below }

PROOF
ANY n
[A1:] n mod 4 = 0
[G:] n mod 2 = 0
BEGIN
1. {def. of mod on A1} (∃k:: n = 4*k)
2. {∃-elim on 1}. [SOME k] n = 4*k
3. {arith. on 2} n = 2*2k
4. {∃-intro on 3} (∃m:: n = 2*m)
5. {def. mod on 4} n mod 2 = 0

END

END

for a>0:

$p \ \mathbf{mod}\ a = 0 \ \Leftrightarrow (\exists k:: p = a*k)$

Proof with induction

- Induction over "natural numbers" is based on this rule:

$$\frac{P\ 0 \quad , \quad (\forall n: 0 \le n : P\ n \Rightarrow P\ (n+1))}{(\forall n: 0 \le n : P\ n)}$$

(n is implicitly assumed to be of type int)

Example of a proof with induction

$p \bmod 3 = 0 \quad \Leftrightarrow \quad (\exists k :: p = 3*k)$

PROOF
[G:] $(\forall n: 0 \le n: (n^3 + 2n) \bmod 3 = 0)$
BEGIN
1. { trivial } $(0^3 + 2*0) \bmod 3 = 0$
2. { see below } $(\forall n: 0 \le n: (n^3 + 2n) \bmod 3 = 0 \Rightarrow ((n+1)^3 + 2(n+1)) \bmod 3 = 0)$

PROOF Ind
ANY n
[A1:] $0 \le n$
[A2:] $(n^3 + 2n) \bmod 3 = 0$
[G:] $((n+1)^3 + 2(n+1)) \bmod 3 = 0$
BEGIN
1. { def. **mod** on A2 } $(\exists k : n^3 + 2n = 3*k)$
2. { $\exists$-elim } [SOME k] $n^3 + 2n = 3*k$
3. { see subproof}
   $(n+1)^3 + 2(n+1) = 3(k + n^2 + n + 1)$

EQUATIONAL PROOF
$(n+1)^3 + 2(n+1)$
$= \{arithm.\} \ (n^3 + 3n^2 + 3n + 1) + (2n + 2)$
$= \{arithm.\} \ (n^3+2n) + 3n^2 + 3n + 3$
$= \{ Ind.2 \} \ 3*k + 3*(n^2 + n + 1)$
$= \{arithm.\} \ 3(k + n^2 + n + 1)$

4. {2, $\exists$-intro} $(\exists m : ((n+1)^3 + 2(n+1)) = 3*m)$
5. {def. **mod** } $((n+1)^3 + 2(n+1)) \bmod 3 = 0$
END

3. { induction, using 1,2} $(\forall n: 0 \le n: (n^3 + 2n) \bmod 3 = 0)$
END

Lists

(LN Ch.4)

Why lists ?

- Our arrays are infinite. On the other hand, we sometimes need to specify aggregate properties of finite segments of an array → we will use a list.
  - So, we add list to one Formula-language.
  - For convenience, we'll use Haskell like notations to express properties of lists.
- Proving equivalence between different implementations of list functions (to extend what you learned in the course Functional Programming)

Some list notation

- [ ], [1, 2, 3]
- x:s, s ++ t
- Enumeration: (a and be below are integers)
  - [a .. b]
  - [a .. b)
- List comprehension, e.g.: [2*i | i **from** s , isOdd i]

List functions

- On lists we can define functions like SUM, MAX, COUNT etc:

$$\begin{aligned} \textbf{SUM}\ [\,] &= 0 \\ \textbf{SUM}\ (x{:}s) &= x + \textbf{SUM}\ s \end{aligned}$$

- We can map array over finite domain to list, and thus can define notions like SUM, MAX over a finite array-segment by defining them over the corresponding list.

Converting array to list

- Let a be an array (infinite), and s be a list of integers. We define:

```
a s = [ a[i] | i from s ]
```

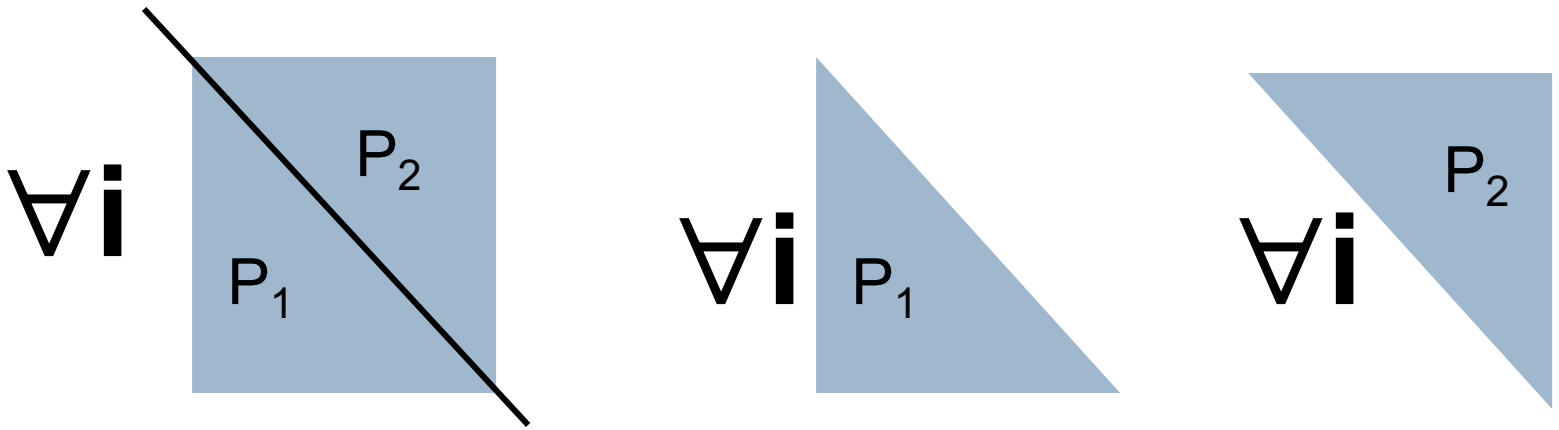
- So now formulas like these are well defined:
  - a[0..n) : the segment of the array a, starting from index 0 up to but not including n.
  - **SUM** (a[0..n)) : the sum of the elements of the segment-array a[0..n).
  - Similarly: **COUNT** (a[0..n)) , **MAX** (a[0..n)) , ...

Domain split

- For $\forall, \exists$ (Thm A.4.12) :

$$(\forall i : P_1\ i \vee P_2\ i : Q\ i) \quad = \quad (\forall i : P_1\ i : Q\ i) \quad \wedge \quad (\forall i : P_2\ i : Q\ i)$$

$$(\exists i : P_1\ i \vee P_2\ i : Q\ i) \quad = \quad (\exists i : P_1\ i : Q\ i) \quad \vee \quad (\exists i : P_2\ i : Q\ i)$$



Inspire solutions for the problem of computing $\forall$ over P_1+P_2 by dividing it into smaller problems.

Domain Split for other "quantifiers"

- Analogous, e.g. :

$$(\Sigma i : P_1\ i \vee P_2\ i : i) \quad = \quad (\Sigma i : P_1\ i : i) \quad + \quad (\exists i : P_2\ i : i)$$

But only if P_1 and P_2 are disjoint!

- On the other hand:

$$\textbf{SUM}\ (s \mathbin{++} t) \quad = \quad \textbf{SUM}\ s \ + \ \textbf{SUM}\ t$$

Now we can do SUM-split on array

EQUATIONAL PROOF

[A:] k≥0

SUM (a[0 .. k+1))
= { enumeration split, Thm A.5.8, justified by A }
SUM (a([0..k) ++ [k]))
= { comprehension split Thm A.5.12 }
SUM (a[0..k) ++ [a[k]])
= {domain split of SUM (prev. slide ... see also Thm A.5.16) }
SUM (a[0..k)) + SUM [a[k]]
= { def. SUM }
SUM (a[0..k)) + a[k]

END

Some standard ‘splitting’ thms

- **SUM** (s ++ t) = **SUM** s + **SUM** t
- **COUNT** (s ++ t) = **COUNT** s + **COUNT** t
- **MAX** (s ++ t) = **MAX** s max **MAX** t

// provided s,t are non-empty

Induction

- Some standard thms in your Appendix for convenience.
- Some properties may require induction to prove. We'll use this list-induction rule:

$$\frac{P\ [] \qquad (\forall x,s::\ P\ s \Rightarrow P(x:s))}{(\forall s::\ P\ s)}$$

- Simple example, prove:

COUNT s $\geq$ 0 , for all s

Top level proof

PROOF main
[G:] (∀s :: COUNT s ≥ 0)
BEGIN
1 { from def. COUNT } COUNT [ ] ≥ 0
2 { subproof } (∀x,s :: COUNT s ≥ 0 ⇒ COUNT (x:s) ≥ 0)

> **PROOF** Pinduct
> **ANY x,s**
> **[A:]** COUNT s ≥ 0
> **[G:]** COUNT (x:s) ≥ 0
> ...
> ...
> **END**

3 { list-induction on 1 and 2 } (∀s :: COUNT s ≥ 0)
END

Reasoning about functional programs

- This is also covered in the course Functional Programming. We will illustrate it with an example.
- Example, two functional programs to do reverse a list :

```
rev [ ]      =  [ ]
rev (x : s)  =  rev s ++ [x]
```

```
rv t [ ]     =  t
rv t (x : s) =  rv (x : t) s
```

The first one is intuitive, the 2nd is much faster. Prove that they are equivalent. More precisely, prove : rev s = rv [ ] s, for all s.

Top level proof

PROOF main
[G:] (∀s :: rev s = rv [ ] s)
BEGIN
1 { def. of rev and rv } rev [ ] = rv [ ] [ ]

2 { subproof } (∀x,s :: (rev s = rv [] s) ⇒ (rev (x:s) = rv [] (x:s)))

3 { list-induction on 1 and 2 } G
END

Attemp-1

PROOF main
[G:] (∀s :: rev s = rv [ ] s)
BEGIN
1 { def. of rev and rv } rev [ ] = rv [ ] [ ]
2 { subproof below } (∀x,s :: (rev s = rv [] s) ⇒ (rev (x:s) = rv [] (x:s)))

Induction hypothesis (IH)

Essentially:
rev (x:s)
= { def. rev } rev s ++ [x]
= { IH } rv [ ] s ++ [x]
= { ?? cannot find a justification for this! } rv [x] s
= { def. rev } rv [ ] (x : s)

We need more information to progress. Basically a way to turn rev s ++ t to rv t s.

3 { list-induction on 1 and 2 } G
END

Attemp-2: let's prove this first

PROOF main
[G:] (∀s :: (∀t :: rev s ++ t = rv t s))
BEGIN
1 { subproof-1 } (∀t :: rev [ ] ++ t = rv t [ ])
2 { subproof-2 below }
(∀x,s :: (∀t :: rev s ++ t = rv t s) ⇒ (∀t :: rev **(x:s)** ++ t = rv t **(x:s)**))

Induction hypothesis (IH)

Essentially:

```
    rev (x:s) ++ t
=  { def. rev }    (rev s ++ [x]) ++ t
=  { associativity of ++ and def. ":" }  rev s ++ (x : t)
=  { IH }   rv (x : t) s
=  { def. rev } rv t (x : s)
```

3 { list-induction on 1 and 2 } G
END

Back to the original problem

- The main claim to prove was (∀s :: rev s = rv [ ] s)
- Along the way, we proved a lemma:

$$(\forall s :: (\forall t :: rev\ s\ ++\ t\ =\ rv\ t\ s)\ )$$

- But notice that this lemma **directly** implies the original claim (without having to do prove the latter through induction)., namely by instantiating t to [ ]. I leave the formal proof of this to you.