CS 565

Programming Languages (graduate)
Spring 2024

Week 14

Course Review

Functional Programming

Algebraic Data Types

- Enumerated types are the simplest data types in Coq
- Type annotations can be inferred here
- Constructors describe how to **introduce** a value of a type

```
Inductive bool :=
| true
| false.

Inductive weekdays :=
  | monday | tuesday | wednesday | thursday
  | friday : weekdays.
```

Pattern Matching

- Pattern matching lets a program use values of a type
- Coq only permits **total** functions
  - A total function is defined on all values in its domain

```
Definition negb (b : bool) : bool :=
  match b with
  | true => false
  | false => true
  end.

Eval compute in (negb true).  (* = false *)
```

Total Maps

Standard operations: higher-order functions:

```
Definition total_map : Type := string -> nat.

Definition lookup (m : map) (x : string) : nat := m x.
Definition empty : map := fun x => 0.

Definition update (m : map) (x : string) (v : nat) : map :=
  fun y => if (eqb_string x y) then v else m y.


Definition example : map := update (update empty “x” 1 ) “y” 2.
```

What is the behavior of m?

```
Definition m : map :=
    update (update (fun y => 42) “x” 7 ) “z” 10.
```

Generic Lists

Coq supports **type abstraction** in data type declarations via **type parameters:**

```
Inductive list (X : Type) : Type :=
  | nil
  | cons (x : X) (l : list X).
```

list is a function from types to types:

```
Check list.  (* : Type -> Type *)
```

Induction

Tree Induction

Works for trees too:

Mathematical Induction for Binary Trees:

For any predicate Q on binary trees, **if:**

1. Q(leaf)
2. $Q(t_1)$ and $Q(t_2)$ implies $Q(\text{node } n\ t_1\ t_2)$

Then:

for all t, Q(t) holds.

Lists


```
Inductive list {X : Type} : Type :=
  | nil
  | cons (x : X) (l : list).
```

Mathematical Induction for Lists:

For any predicate Q on lists, **if:**

1. Q(nil)
2. Q(l) implies Q(cons x l)

Then:

for all l, Q(l) holds.

a list constructed by adding x to the head of l

PROPOSITIONS, DEPENDENT TYPES, AND CONSTRUCTIVE PROOFS

Propositions

A **proposition** is a factual claim.

Have seen a couple of propositions (in Coq) so far:

equalities: 0 + n = n

implications: P -> Q

universally quantified propositions: forall x, P

A **proof** is some evidence for the truth of a proposition

A **proof system** is a formalization of particular kinds of evidence.

Propositions

Propositions are first-class entities in Coq. Can name them:

```
Definition plus_claim : Prop := 2 + 2 = 4.
Theorem ProofExample : plus_claim.
Proof.
… (* unfold plus_claim*)
```

We can also write parameterized propositions (**predicates**)

```
Definition is_three (n : nat) : Prop := n = 3.
Theorem ProofExample2 : is_three 3.
Proof.
… (* unfold is_three *)
```

Inference Rules

Proof systems construct evidence of judgements via inference rules:

Axioms

$$\frac{}{\Gamma \vdash T} \qquad \frac{A \in \Gamma}{\Gamma \vdash A}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \mathbf{I}\!\to \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \mathbf{E}\!\to$$

Inference Rules

Proof

Haven't we already seen a number of proofs?

```
Theorem ProofExample
  : forall n m : nat, n = 0 -> m = 0 -> n + m = 0.
Proof.
 intros n m Hn Hm.
 rewrite Hn. rewrite Hm.
 reflexivity.
```

proofscript

What is a formal proof?

A proof tree in the Calculus of co-Inductive Constructions.

Propositions


```
Inductive ev : nat → Prop :=
|   ev_0 : ev 0
|   ev_SS (n : nat) (H : ev n) : ev (S (S n))
```

Read ":" to mean "proof of"

The type of `ev_SS` is:

```
∀n.  ev n → ev (S (S n))
```

What is an element that inhabits `ev 4`?

It is the proof object (proof tree):

```
ev_SS 2 (ev_SS 0 ev_0)
```

This object is built via the following proof script:

```
apply ev_SS.
apply ev_SS.
apply ev_0.
```

Alternatively


```
Theorem ev_plus4: ∀n, ev n → ev (4 + n).
Proof.
  intros n H.
  simpl.
  apply ev_SS. apply ev_SS. apply H.
Qed.
```

Here is an object that has this type:

```
Definition ev_plus4' : ∀n, ev n → ev (4 + n) :=
  fun (n : nat) => fun (H : ev n) =>
    ev_SS (S (S n)) (ev_SS n H).
```

Also:

```
Definition ev_plus4'' (n : nat) (H : ev n) : ev (4 + n) :=
    ev_SS (S (S n)) (ev_SS n H).
```

Observation

- Quantification allows us to refer to the value of an argument in the type of another:

```
∀n, ev n → ev (4 + n)
```

- Implication is essentially a degenerate form of quantification:

```
∀ (x: nat), nat
∀ (_: nat), nat
nat → nat
```

`∀ (_ : P), Q` is the same as `P → Q`

Induction Principles


```
Inductive nat :
| 0
| S (n : nat).
```

```
Check nat_ind :
  forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n.
```

```
Inductive time :
| day
| night.
```

```
Check time_ind :
  forall P : time -> Prop,
    P day ->
    P night ->
    forall t : time, P t.
```

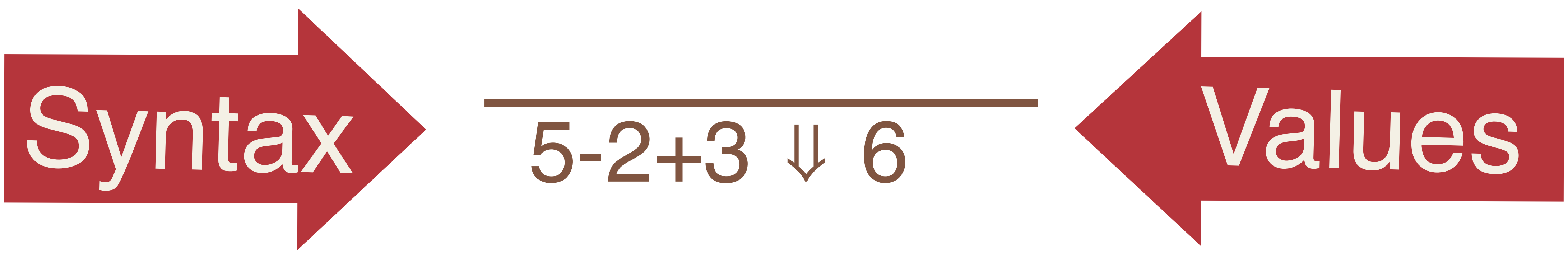
More generally, for a type with n constructors, an induction principle of the following shape is generated:

```
t_ind : forall P : t -> Prop,
          ... case for c1 ... ->
          ... case for c2 ... -> ...
          ... case for cn ... ->
          forall n : t, P n
```

BIGSTEP AND SMALLSTEP SEMANTICS

Big-Step Semantics

- Binary relation on pairs of syntax and values
- Read '⇓' as 'evaluates to'
- Specifies what values program can map to



- Good for whole program reasoning
  - Compiler Correctness; program equivalence;

- Bad for talking about intermediate states
  - Concurrent programs; errors

Small-Step

- Binary relation on pairs of expressions
- Read '$e_1 \longrightarrow e_2$' as 'reduces to'
- Specifies single transition of abstract machine
- Exposes intermediate states

Small-Step Termination

- How to tell when we're 'done' evaluating?
- Define a class of syntactic values:

$$\frac{}{\textbf{value}\ C\ n}$$

Now we can talk about making progress

Theorem [Strong Progress]:

For any term t, either t is a value or there exists a term t' such that t $\longrightarrow$ t'.

Normal Form

A term e that isn’t reducible is in normal form.

$$\neg \exists e'. e \longrightarrow e'$$

How is this different from a value?

Syntactic versus semantic.

Do not need to coincide!

MultiStep Relation

We generically lift single-step to full execution as the *transitive*, *reflexive* closure:

$$\frac{}{e \longrightarrow^* e}\ \textsc{Refl} \qquad \frac{e_1 \longrightarrow^* e_2 \qquad e_2 \longrightarrow e_3}{e_1 \longrightarrow^* e_3}\ \textsc{Trans}$$

So: (C 1)+((C 2) + (C 3))+((C 4)+(C 6))) $\longrightarrow^*$ 16:

1+((2+3)+(4+6)) $\longrightarrow$ 1+(5+(4+6)) $\longrightarrow$ 1+(5+10)
$\longrightarrow$ 6+10 $\longrightarrow$ 16

TYPE SYSTEMS

Typing Imp$^+$

A recipe for type systems:

1. Define bad programs
2. **Define typing rules for classifying programs**
3. Show that the type system is sound, i.e. that it only identifies good programs

Typing Rules

Next, define a classifier for good, well-formed programs:

$$\vdash e : T$$

Goal is to classify good uses of each type of expression:

$$\frac{n \in N}{\vdash \mathbf{n} : \text{nat}}\ \text{TNUM} \qquad \frac{\vdash e_1 : \text{nat} \qquad \vdash e_2 : \text{nat}}{\vdash e_1 + e_2 : \text{nat}}\ \text{TADD}$$

$$\frac{}{\vdash x : \text{nat}}\ \text{TVAR}$$

$$\frac{\vdash e1 : \text{nat} \qquad \vdash e2 : \text{nat}}{\vdash e_1 * e_2 : \text{nat}}\ \text{TMULT}$$

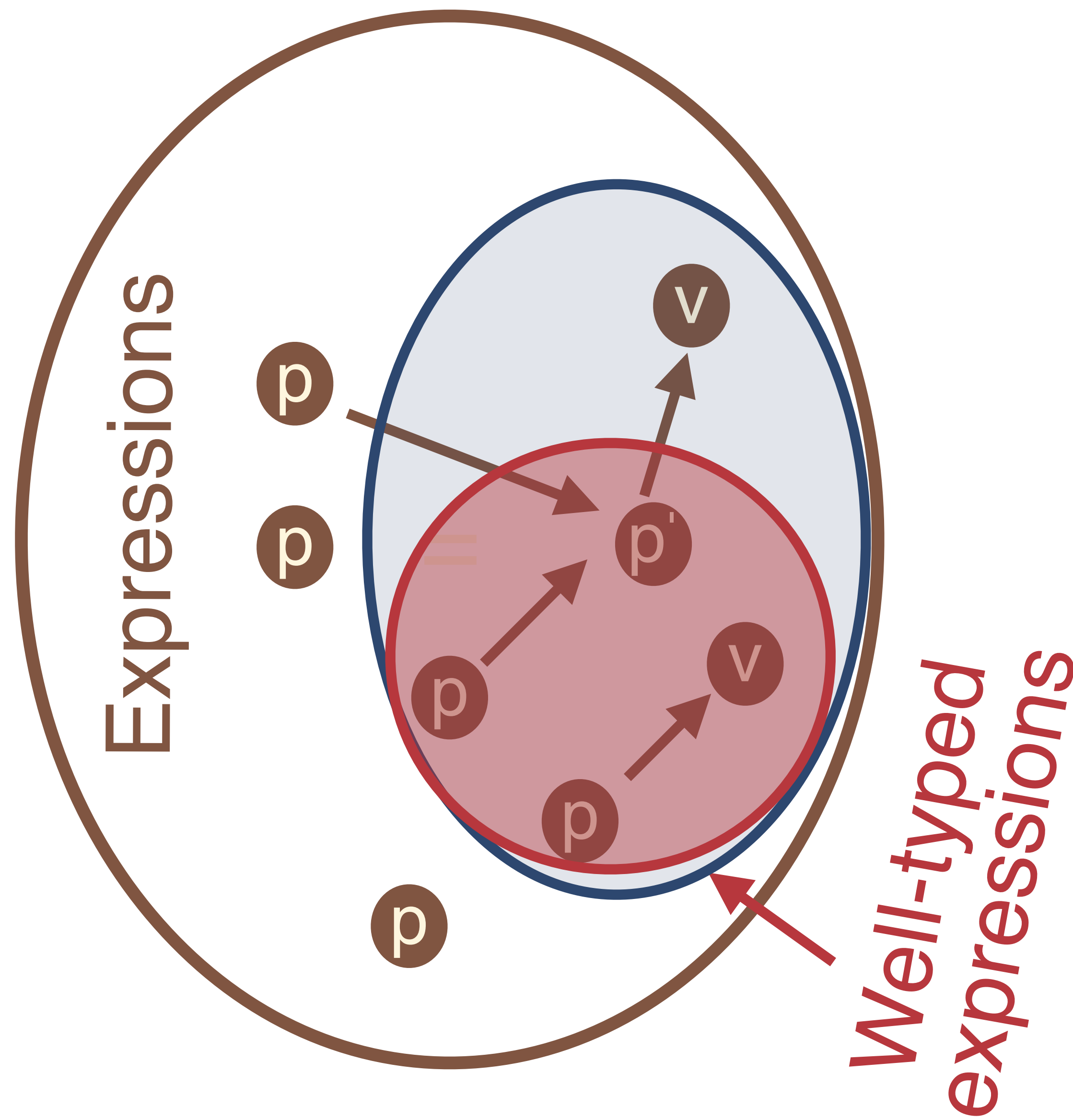
Progress

Theorem [PROGRESS]: Suppose e is a well-typed expression ($\vdash$e:T). Then either e is a value or there exists some e' such that e evaluates to e' (σ, e $\longrightarrow$ e').

Values:

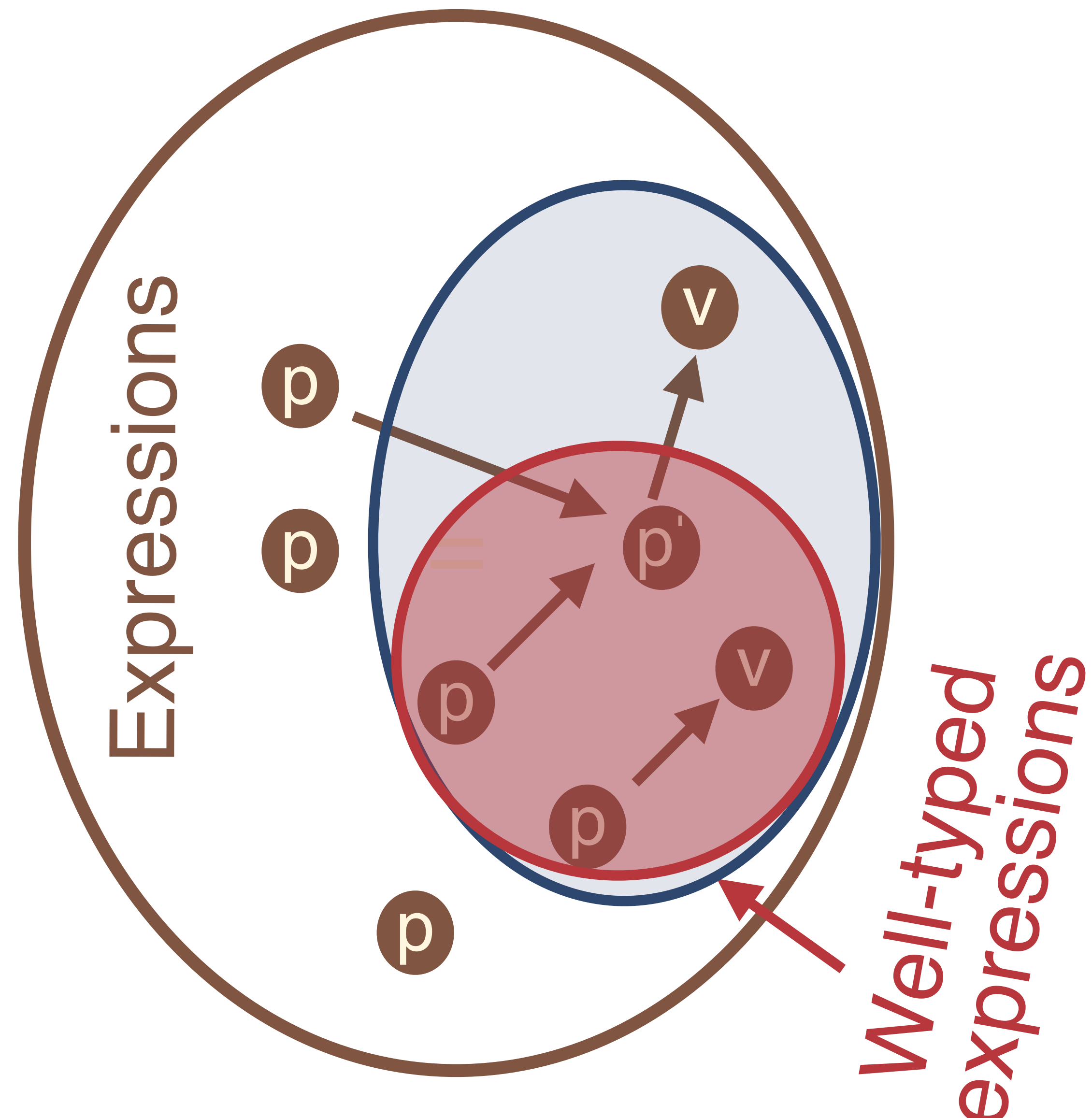
$$\frac{}{\text{value true}}\ \textsc{TValue}$$

$$\frac{n \in N}{\text{value } n}\ \textsc{NumValue}$$



Preservation

- **Theorem** [PRESERVATION]: Suppose e is a well-typed term ($\vdash$ e : T). Then, if e evaluates to e', e' is also a well-typed term under the empty context, with the same type as e ($\vdash$ e' : T).



Type Soundness

Theorem [Type Soundness]: If an expression e has type T, and e reduces to e' in zero or more steps, then e' is not a stuck term.

Proof.

By induction on $\sigma, e \longrightarrow^* e'$…

Qed.

★ Corollary [Normalization]: If an expression e has type T, e reduces to a value in zero or more steps.

Variance

Variance is a property on the arguments of type constructors like function types $(A \rightarrow B)$, tuples $(A \times B)$, and record types

$F(A)$ is **covariant** over A if $A <: A'$ implies that $F(A) <: F(A')$

$F(B)$ is **contravariant** over B if $B' <: B$ implies that $F(B) <: F(B')$

$F(T)$ is **invariant** over T otherwise

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2} \quad \text{SB-TUPLE}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad \text{SB-ARROW}$$

HOARE LOGIC

Hoare Triple

- Step 1B: Define a judgement for claims about programs involving assertions
- Partial Correctness Triple:

$$\{P\}\ c\ \{Q\}$$

Hoare Skip

Use our intuition about what we want to be able to prove to guide definition of rules

$$\{P\}\, c\, \{Q\} \equiv$$
$$\forall \sigma.\ \sigma \models P \rightarrow\ \ \forall \sigma'.\ \sigma, c \Downarrow \sigma' \rightarrow \sigma' \models Q$$

Hoare Assign

$$\{[X := a]Q\}\ X := a\ \{Q\} \equiv$$

$$\forall \sigma.\ \sigma \models [X := a]Q \rightarrow$$

$$\quad \forall \sigma'.\ \sigma,\ X := a \Downarrow \sigma' \rightarrow \sigma' \models Q$$

$$\frac{}{\vdash \{[X := a]Q\}\ X := a\ \{Q\}}\ \textsc{HLAssign}$$

Hoare Seq

$$\frac{\vdash \{P\}\, c_1\, \{R\} \quad \vdash \{R\}\, c_2\, \{Q\}}{\vdash \{P\}\; c_1 ; c_2\; \{Q\}}\ \textsc{HLSeq}$$

Hoare While?

$$\dfrac{\dfrac{\vdash \{X < 4 \land X < 3\}\ X := X + 1\ \{X < 4\}}{\vdash \{X < 4\}\ \textbf{while}\ (X < 3)\ \textbf{do}\ \ X := X + 1\ \textbf{end}\ \{X < 4 \land \neg X < 3\}}}{\vdash \{X < 3\}\ \textbf{while}\ (X < 3)\ \textbf{do}\ \ X := X + 1\ \textbf{end}\ \{X = 3\}}$$

$$\frac{\vdash \{Q \land b\}\ c\ \{Q\}}{\vdash \{Q\}\ \text{while } b \text{ do } c \text{ end}\ \{Q \land \neg b\}}$$

Hoare While

I is a *loop invariant*:

- Holds before loop
- Holds after each loop iteration
- Holds when the loop exits

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \text{while}\ b\ \text{do}\ c\ \text{end}\ \{I \wedge \neg b\}}\ \text{HLWHILE}$$

Loop Invariants

Hoare Logic is a structural model-theoretic proof system

- Rules characterize a set of states consistent with the requirements imposed by the pre- and post-conditions
- Highly mechanical: intermediate states can almost always be automatically constructed
- One major exception:

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end}\ \{I \wedge \neg b\}}\ \text{HLW}\textsc{hile}$$

The invariant must:

- be weak enough to be implied by the precondition
- hold across each iteration
- be strong enough to imply the postcondition

DAFNY

Decreases clause


```
function seqSum (s : seq<int>, lo : int, hi : int) : int
    requires 0 <= lo <= hi <= |s|
{
    if (lo == hi) then 0 else s[lo] + seqSum(s, lo+1, hi)
}
```

Dafny complains that it cannot prove the recursive call terminates - it is unable to identify a termination metric that signals every recursive call gets "smaller"

```
function seqSum (s : seq<int>, lo : int, hi : int) : int
    requires 0 <= lo <= hi <= |s|
     decreases hi - lo
{
    if (lo == hi) then 0 else s[lo] + seqSum(s, lo+1, hi)
}
```

What about using -lo as a decreases clause?

Proof Calculations

Constructive proofs that involve rewrites and simplification

```
calc {
   (x + y) * (x - y);
==
   (x * x) - (x * y) + (y * x) - (y * y);
==
   (x * x) - (x * y) + (x * y) - (y * y);
==
   (x * x) - (y * y);
}
```

Proof Calculations and Induction


```
lemma {:induction false} MirrorMirror<T>(t: Tree)
    ensures mirror(mirror(t)) == t
    {
        match t
            case Leaf(_) =>
            case Node(left,right) =>
                calc
                {
                    mirror(mirror(Node(left,right)));
                    ==
                    mirror(Node(mirror(right),mirror(left)));
                    ==
                    Node(mirror(mirror(left)),mirror(mirror(right)));
                    ==  // IH
              { MirrorMirror(left);    MirrorMirror(right); }
              Node(left, right);
                }
    }
```

Proofs by Contradiction

General shape:

```
!Q -> (R /\ !R)
_______________

       Q
```

```
lemma Lem(args)
   requires P(x)
   ensures Q(x)
 {
    if !Q(x)                // property is false
      {
         assert !P(x)       // contradiction: precondition is
         assert false       // true and false
    }
   assert Q(x)
}
```