Software Verification with LEAN

Bob Rubbens Lukas Armborst Ömer Șakar

FMT, University of Twente

23 Sep. 2021

Table of Contents

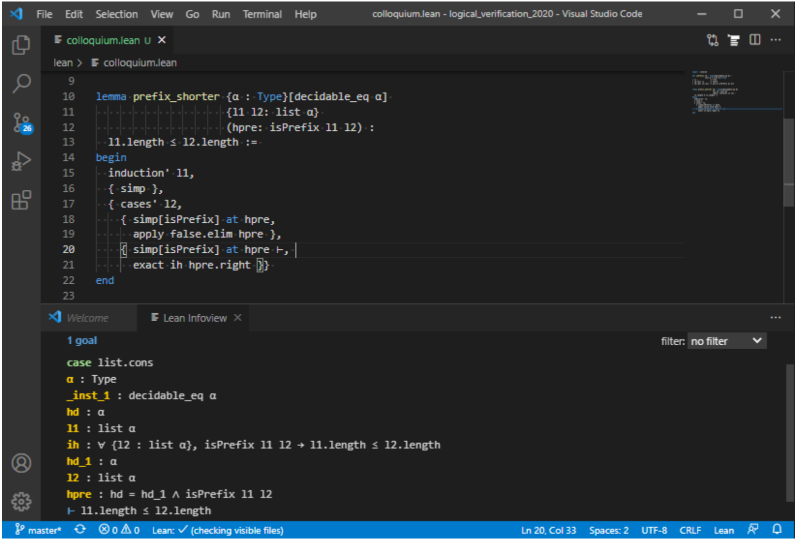
What is Lean?

- Interactive theorem prover
- Functional programming language
- Everything is inductive
- Backward proofs
- Visual Studio Code plugin

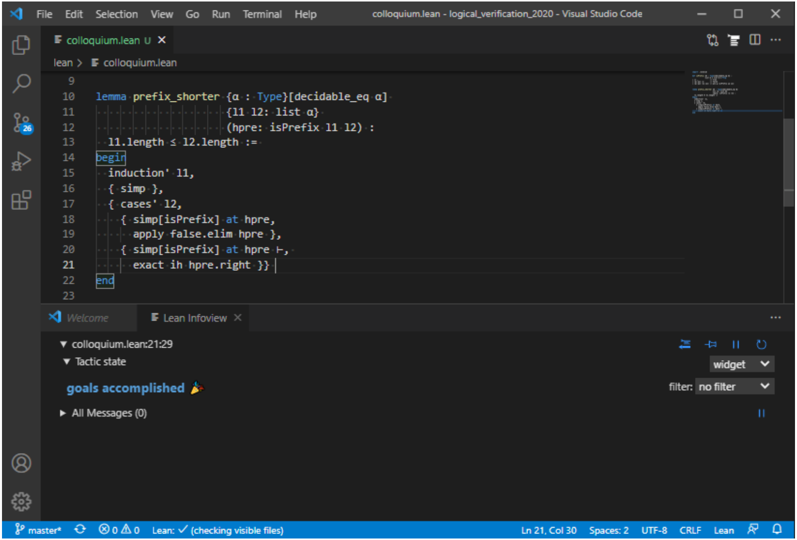
```
def isPrefix {α : Type} :
        list α → list α → Prop
| [] _              := true
| (p::ps) []        := false
| (p::ps) (a::as) := p=a ∧(isPrefix ps as)

lemma prefix_shorter {α : Type}
            (l1 l2: list α)
            (hpre: isPrefix l1 l2) :
  l1.length ≤ l2.length :=
begin
  induction' l1,
  { simp },
  { cases' l2,
    { simp[isPrefix] at hpre,
      apply false.elim hpre },
    { simp[isPrefix] at hpre ⊢,
      apply ih hpre.right }}
end
```

What is Lean?


File Edit Selection View Go Run Terminal Help
colloquium.lean - logical_verification_2020 - Visual Studio Code
colloquium.lean U
lean > colloquium.lean
lemma prefix_shorter {α : Type}[decidable_eq α]
{l1 l2: list α}
(hpre: isPrefix l1 l2) :
l1.length ≤ l2.length :=
begin
induction' l1,
{ simp },
{ cases' l2,
{ simp[isPrefix] at hpre,
apply false.elim hpre },
{ simp[isPrefix] at hpre ⊢,
exact ih hpre.right }}
end
Welcome
Lean Infoview
1 goal
filter: no filter
case list.cons
α : Type
_inst_1 : decidable_eq α
hd : α
l1 : list α
ih : ∀ {l2 : list α}, isPrefix l1 l2 → l1.length ≤ l2.length
hd_1 : α
l2 : list α
hpre : hd = hd_1 ∧ isPrefix l1 l2
⊢ l1.length ≤ l2.length
master*
Lean: ✓ (checking visible files)
Ln 20, Col 33 Spaces: 2 UTF-8 CRLF Lean

What is Lean?


File Edit Selection View Go Run Terminal Help
colloquium.lean - logical_verification_2020 - Visual Studio Code
colloquium.lean U
lean > colloquium.lean
lemma prefix_shorter {α : Type}[decidable_eq α]
{l1 l2: list α}
(hpre: isPrefix l1 l2) :
l1.length ≤ l2.length :=
begin
induction' l1,
{ simp },
{ cases' l2,
{ simp[isPrefix] at hpre,
apply false.elim hpre },
{ simp[isPrefix] at hpre ⊢,
exact ih hpre.right }}
end
Welcome
Lean Infoview
colloquium.lean:21:29
Tactic state
goals accomplished
widget
filter: no filter
All Messages (0)
master*
Lean: ✓ (checking visible files)
Ln 21, Col 30 Spaces: 2 UTF-8 CRLF Lean

Comparison ITP: Lean vs. ...

... Coq:

1. For outsiders: Coq ≈ Lean
   - Syntax and interface ~~cruft~~ differences
   - Automation: Ltac vs. Lean
2. Coq preserves "strong normalization, subject reduction, and canonicity"[1]

... Isabelle:

1. Major difference: HOL vs. CIC
2. Automation: plugins/Eisbach

[1] https://artagnon.com/articles/leancoq

Comparison: matrix multiplication[2]

```
def mult :
  forall (n m p : nat), matrix n m -> matrix m p
                                   -> matrix n p
:= /- ... -/
```

```
fun mult ::  matrix -> matrix -> matrix
where
  (* ... *)
lemma mult_sizes:
  "size (mult a b) = (fst (size a), snd (size b))"
by (* ... *)
```

[2] https://stackoverflow.com/q/30152139, Arthur Azevedo De Amorim

Table of Contents

Axiomatic Data Types (ADTs)

Axiomatic Data Types (ADTs)

- Lists/sequences, sets, bags, tuples, integers, doubles, floats, etc.

Axiomatic Data Types (ADTs)

- Lists/sequences, sets, bags, tuples, integers, doubles, floats, etc.
- Concrete data types (CDTs)

Axiomatic Data Types (ADTs)

- Lists/sequences, sets, bags, tuples, integers, doubles, floats, etc.
- Concrete data types (CDTs)
- Describe *behavior* instead of *implementation*

Map ADT

Map ADT

collection of key/value pairs with unique keys

Map ADT

, immutable collection of key/value pairs with unique keys

Map ADT

, finite, immutable collection of key/value pairs with unique keys

Map ADT

- Unordered, finite, immutable collection of key/value pairs with unique keys

Map constructors

Map constructors

- empty(): Map[K,V]

Map constructors

- empty(): Map[K,V]
- build(m: Map[K,V], k: K, v: V): Map[K,V]

Map constructors

- empty(): Map[K,V]
- build(m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$

Map constructors

- empty(): Map[K,V]
- build(m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$

Map constructors

- empty(): Map[K,V]
- build(m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \to 1$, $2 \to 4$ and $3 \to 9$

empty()

Map constructors

- empty(): Map[K,V]
- build(m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \to 1$, $2 \to 4$ and $3 \to 9$

  build(empty(), 1, 1)

Map constructors

- empty(): Map[K,V]
- build(m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$

  build(build(empty(), 1, 1), 2, 4)

Map constructors

- empty(): Map[K,V]
- build(m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$
- build(build(build(empty(), 1, 1), 2, 4), 3, 9)

Modeling a map

Modeling a map

- keys(m: Map[K,V]): Set[K]

Modeling a map

- keys(m: Map[K,V]): Set[K]
- get(m:Map[K,V], k: K): V

Modeling a map

- keys(m: Map[K,V]): Set[K]
- get(m:Map[K,V], k: K): V
- card(m:Map[K,V]): Int

Axiom on the build function

```
axiom Ax3 {

}
```

Axiom on the build function

```
axiom Ax3 {
     forall k1: K, v1: V , m1: Map[K,V] ::


}
```

Axiom on the build function

```
axiom Ax3 {
    forall k1: K, v1: V, m1: Map[K,V] ::


}
```

Axiom on the build function

```
axiom Ax3 {
    forall k1: K, v1: V, m1: Map[K,V] ::
        k1 in keys(build(m1, k1, v1)) &&

}
```

Axiom on the build function

```
axiom Ax3 {
     forall k1: K, v1: V, m1: Map[K,V] ::
          k1 in keys(build(m1, k1, v1)) &&
          get(build(m1, k1, v1), k1) == v1
}
```

Unsoundness/Inconsistency

```
axiom MyUnsoundAxiom1 {


}
```

Unsoundness/Inconsistency

```
axiom MyUnsoundAxiom1 {
    get(m1, k1) !=
    get(m1, k1)
}
```

Unsoundness/Inconsistency

```
axiom MyUnsoundAxiom1 {
    get(m1, k1) !=
     get(m1, k1)
}

axiom MyUnsoundAxiom2 {
     false
}
```

Maps in Lean

Map constructors

```
inductive map {α β: Type} : Type*
| nil : map
| build (k:α) (v:β) (m: map): map
```

The definition of `map.card`

```
def map.card : @map α β → nat
| map.nil := 0
| (map.build k v m) :=
    (if (k ∈ m.keys) then 0 else 1) + (m.card)
```

The theorems of `map.card`

```
theorem vctMapCardAx1 (m: @map α β):
    m.card >= 0 :=
    begin
      <a three line proof>
    end
```

The theorems of `map.card`

```
theorem vctMapEmptyCardAx1 (m: @map α β):
    m.card = 0 ↔ m = (@map.nil α β) :=
    begin
        <a nineteen line proof>
    end
```

Lessons learned:

Lessons learned:

- Choose your map definition well!

Lessons learned:

- Choose your map definition well!
- Reuse the extensive library of Lean

Table of Contents

Overview

- Based on red-black tree verification project
- Two sub-projects:
  1. prove prefix- and infix-related lemmas not proven in VerCors
  2. re-do verification of some methods

Unproved Lemmas

- 3 lemmas about infixes and sortedness not proven in VerCors
- all 3 proven successfully in Lean
- found one small bug in specification of Java code
  → probable reason for VerCors to fail

⇒ conversion from integers to naturals often annoying

Re-Doing Proofs

- Re-implemented methods of a producer-consumer class in Lean
- Ignored access permissions
- One lemma for each post-condition
- Proved nearly all post-conditions

$\Rightarrow$ often rather lengthy proofs

$\Rightarrow$ termination required $\rightarrow$ progress of waiting loop assumed $\rightarrow$ imperfect representation

Re-Doing Proofs

Java in VerCors
(within the Queue class):

```java
/*@ requires consumer(); @*/
public boolean hasNext() {

  if (reading == null) {
    if (isLastBatch) {
      return false;
    }
    getBatch();
  }



  boolean res = reading!=null;
  return res;
}
```

Lean:

```lean
def hasNext {α: Type}
          (q: Queue α) (hc: q.consumer)
           : bool × Queue α :=
  if hr: q.reading.empty
  then if hl: q.isLastBatch
    then (ff, q)
    else have q': Queue α
                :=  (getBatch q hc hr
                      (heads_equal q hc hr)
                      hl
                      (assumedProgress q)
                    ).snd,
          (¬q'.reading.empty, q')
  else (tt, q)
```

Re-Doing Proofs

- Re-implemented methods of a producer-consumer class in Lean
- Ignored access permissions
- One lemma for each post-condition
- Proved nearly all post-conditions

$\Rightarrow$ often rather lengthy proofs

$\Rightarrow$ termination required → progress of waiting loop assumed → imperfect representation

Re-Doing Proofs

Java in VerCors
(within the `Queue` class):

```
/*@ requires consumer();
    ensures readHead
          == \old(readHead);@*/
public boolean hasNext() {
  if (reading == null) {
    if (isLastBatch) {
      return false;
    }
    getBatch();
  }
  boolean res = reading!=null;
  return res;
}
```

Lean:

```
lemma hasNext_readHead {α: Type}
          (q: Queue α) (hc: q.consumer) :
  (hasNext q hc).snd.readHead
        = q.readHead :=
begin
  simp[hasNext],
  cases' classical.em q.reading.empty
             with hr hr,
  {cases' classical.em q.isLastBatch
              with hl hl,
    {simp[hr, hl]},
    {simp[hr, hl],
     apply eq.symm,
     apply getBatch_readHead
            q hc hr _ hl
            (assumedProgress q)
            (assumedLockInvariant q)
    } },
  { simp[hr] }
end
```

Re-Doing Proofs

- Re-implemented methods of a producer-consumer class in Lean
- Ignored access permissions
- One lemma for each post-condition
- Proved nearly all post-conditions

$\Rightarrow$ often rather lengthy proofs

$\Rightarrow$ termination required → progress of waiting loop assumed → imperfect representation

Table of Contents

- Language from: "Separation Logic: A Logic for Shared Mutable Data Structures"
- Components of Lean formalization
- Main lemmas

Original language

D0 Axiom of Assignment

$\vdash P_0 \{x := f\}\ P$

D1 Rules of Consequence

If $\vdash P\{Q\}R$ and $\vdash R \supset S$ then $\vdash P\{Q\}S$

If $\vdash P\{Q\}R$ and $\vdash S \supset P$ then $\vdash S\{Q\}R$

D2 Rule of Composition

If $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1\,;\,Q_2)\}R$

D3 Rule of Iteration

If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\textbf{while}\ B\ \textbf{do}\ S\}\neg B \wedge P$

$\langle\text{comm}\rangle ::= \cdots$

$\mid \langle\text{var}\rangle := \textbf{cons}(\langle\text{exp}\rangle, \ldots, \langle\text{exp}\rangle)$	allocation
$\mid \langle\text{var}\rangle := [\langle\text{exp}\rangle]$	lookup
$\mid [\langle\text{exp}\rangle] := \langle\text{exp}\rangle$	mutation
$\mid \textbf{dispose}\ \langle\text{exp}\rangle$	deallocation

Lean datatype

```
def store := LoVe.state
def exp   := store → ℕ
def bexp  := store → Prop

inductive cmd : Type
| skip    : cmd
| assign  : string → exp → cmd
| seq     : cmd → cmd → cmd
| ite     : bexp → cmd → cmd → cmd
| while   : bexp → cmd → cmd
| alloc   : string → exp → cmd
| lookup  : string → exp → cmd
| mutate  : exp → exp → cmd
| dispose : exp → cmd
end
```

Semantics: datatype

```
def state := store × heap

inductive cfg : Type
| abort : cfg
| term : state → cfg
| nonterm : cmd × state → cfg
```

Semantics: notation

```
example (s : store) (h: heap) :
  ⟪s, h⟫ = cfg.term (s, h) :=
by simp

example (sh : state) (h: heap) :
  ⟪sh⟫ = cfg.term sh :=
by simp

example :
  ↯ = ↯ :=
by simp

example (sh : state) :
  ⟨cmd.skip, sh⟩ = cfg.nonterm (cmd.skip, sh) :=
by simp
```

Semantics: small step

```
inductive small_step : cfg → cfg → Prop
| skip {sh} :
  small_step ⟨cmd.skip, sh⟩ ⟪sh⟫
| assign {x e s h} :
  small_step ⟨cmd.assign x e, (s, h)⟩ ⟪(s{x ↦ e s}, h)⟫
| seq_step {c c' d sh sh'}
           (hc : small_step ⟨c, sh⟩ ⟨c', sh'⟩) :
  small_step ⟨c ;; d, sh⟩ ⟨c' ;; d, sh'⟩
| seq_abort {c d sh} (hc : small_step ⟨c, sh⟩ ↯) :
  small_step ⟨c ;; d, sh⟩ ↯
/- ... -/
```

Semantics: more notation

```
/- Small step -/
⟨cmd.skip, (s, h)⟩ ⇒ ⟪s', h'⟫

/- Transitive closure of small step -/
⟨cmd.skip, (s, h)⟩ ⇒* ⟪s', h'⟫

/- Big step -/
⟨cmd.skip, (s, h)⟩ ⟹ ⟪s', h'⟫
```

Semantics: more notation

```
inductive big_step : cfg → cfg → Prop
| skip {sh} :
  big_step ⟨cmd.skip, sh⟩ ⟪sh⟫
| assign {x e s h} :
  big_step ⟨cmd.assign x e, (s, h)⟩ ⟪(s{x ↦ e s}, h)⟫
| seq_abort_l {c d sh} (hc : big_step ⟨c, sh⟩ ↯) :
  big_step ⟨c ;; d, sh⟩ ↯
| seq_abort_r {c d sh sh'}
              (hc : big_step ⟨c, sh⟩ ⟪sh'⟫)
              (hd : big_step ⟨d, sh'⟩ ↯) :
  big_step ⟨c ;; d, sh⟩ ↯
```

Programs and smaller heaps still abort ✓

$\langle \mathtt{c},\ (\mathtt{s},\ \mathtt{h}) \rangle \Rightarrow * \lightning$

$\mathtt{h}_0 \subseteq \mathtt{h}$

$\rightarrow$

$\langle \mathtt{c},\ (\mathtt{s},\ \mathtt{h}_0) \rangle \Rightarrow * \lightning$

Programs and smaller heaps still abort: lemma ✓

```
lemma subheap_maintain_abort
  {c : cmd} {s : store}
  (h₀ h : heap)
  (hh₀h : h₀ ⊆ h) :
  ⟨c, (s, h)⟩ ⇒* ↯ → ⟨c, (s, h₀)⟩ ⇒* ↯ :=
begin
  induction' c,
  case skip {
    finish,
  },
  /- ... -/
end
```

Programs and smaller heaps still terminate ✓

```
lemma subheap_maintain_terminate
  {c : cmd} {s s' : store}
  (h₀ h h' : heap) (hh₀h : h₀ ⊆ h) :
  ⟨c, (s, h)⟩ ⟹* ⟪s', h'⟫ →
  (∃h₀', ⟨c, (s, h₀)⟩ ⟹* ⟪s', h₀'⟫ ∧ h₀' ⊆ h') ∨
    ⟨c, (s, h₀)⟩ ⟹* ↯ :=
begin
  induction' c,
  case skip {
    /- ... -/
  },
  /- ... -/
end
```

Technique: via $\Longrightarrow$

```
/- Given concrete `c` -/

/- Assume: -/ ⟨c, (s, h)⟩  ⇒* ↯
/- Goal: -/   ⟨c, (s, h₀)⟩ ⇒* ↯
/- Have: -/   ⟨c, (s, h)⟩ ⟹ ↯
/- Induct on big step, finish proof -/
```

Programs and smaller heaps still diverge ↯

```
lemma subheap_maintain_diverges₀
(c s h h₀) (hh₀h : h₀ ⊆ h) :
diverges₀ ⟨c, (s, h)⟩ →
⟨c, (s, h₀)⟩ ⇒* ↯ ∨ diverges₀ ⟨c, (s, h₀)⟩ :=
begin
induction' c,
case seq : c d ihc ihd {
  /- ... -/
},
/- ... -/
end
```

Summary

- Lean is a cool tool
- None of us continued their project after the course

→ Network effect drives us to Isabelle/HOL or Coq+Iris

- `https://lean-forward.github.io/logical-verification/2020/index.html`