Program verification in Lean

Soundness of verification-condition generation

CMPUT 664 · Formal Verification · October 21, 2026

Verifying a counting loop

Claim: if this program terminates, its final value of x is 10.

```
1 x : Nat
2 x := 0;
3 while x < 10 do
4     x := x + 1
```

Proposed invariant: $x \leq 10$ whenever the loop condition is tested.

$$\underbrace{x \geq 10}_{\text{at exit: } x<10 \text{ is false}} \quad \wedge \quad \underbrace{x \leq 10}_{\text{from the invariant}} \quad \Longrightarrow \quad x = 10$$

To use this argument, we must prove that the invariant holds initially and is preserved by every iteration.

Verification conditions for counting

$$I(x) \equiv x \leq 10 \qquad Q(x) \equiv x = 10$$

Check	Condition	Meaning
Initialization	$0 \leq 10$	Establish I before the loop
Preservation	$(x \leq 10 \wedge x < 10) \Rightarrow x + 1 \leq 10$	One iteration preserves I
Exit	$(x \leq 10 \wedge x \geq 10) \Rightarrow x = 10$	I and the false guard imply Q

These are the conditions we want a VC generator to compute. Verification requires proofs of all three.

Implementing VC generation in Lean

We will write a Lean function that generates conditions for programs in a small imperative language.

$$\text{program} + \text{contract} + \text{invariants} \xrightarrow{\text{Lean function}} \text{proposition to prove}$$

- **Input program:** assignments, sequences, and `while` loops, with a contract and proposed invariants.
- **VC generator:** recursively inspects that program's structure and constructs its proof obligations.

The generator is itself a program to verify. We must prove that its conditions are sufficient to establish the input program's contract.

Soundness of VC generation

Write $G(P, C, A, Q)$ for the conjunction of generated conditions, where:

- C is the input program and A supplies its loop invariants.
- P is the precondition and Q is the postcondition.

The theorem we will prove in Lean:

$$\forall C, P, A, Q, \qquad G(P, C, A, Q) \implies \{P\}\ C\ \{Q\}$$

If the generated conditions hold, every **terminating** execution of C from a state satisfying P ends in a state satisfying Q.

The counting proof checks one input. This theorem justifies the generator for **every input**.

Input to the generator: annotated programs

An **annotated program** combines the commands C with proposed loop invariants A.

```
1 x := 0;
2 while x < 10 invariant x ≤ 10 do
3     x := x + 1
```

- **Program structure:** the generator processes assignments, sequences, and loops by different rules, recursively.
- **Loop invariant:** the loop rule needs a candidate assertion to form the initialization, preservation, and exit checks.
- **Inductive datatype:** representing this syntax as Lean data lets us define the generator by recursion and prove soundness by induction.

`ACmd` will package the commands and their invariants into one input.

Annotated commands

`S` is the state type. `ACmd S` represents the syntax of an annotated program.

```
1  inductive ACmd (S : Type) where
2    | skip : ACmd S
3    | assign : (S → S) → ACmd S
4    | seq : ACmd S → ACmd S → ACmd S
5    | while : (S → Prop) → (S → Bool) → ACmd S → ACmd S
```

- `assign f` updates the state using `f : S → S`.
- `while I b body` stores an invariant `I : S → Prop`, a guard `b : S → Bool`, and a body.

For `S = Nat`, `fun x => x + 1` represents `x := x + 1`.

Commands without annotations

We define `Cmd S` to represent the program syntax used by the execution rules. Its constructors build commands acting on states of type `S`.

```
1  inductive Cmd (S : Type) where
2    | skip : Cmd S
3    | assign : (S → S) → Cmd S
4    | seq : Cmd S → Cmd S → Cmd S
5    | while : (S → Bool) → Cmd S → Cmd S
```

- `assign f` stores a state-update function; `seq c₁ c₂` combines two commands.
- `while b body` takes a guard `b : S → Bool` and a body `body : Cmd S`. It has **no invariant argument**.

For example, `Cmd.assign (fun x : Nat => x + 1)` constructs the syntax for `x := x + 1`.

These constructors define program structure. The execution rules will specify its meaning.

Annotation erasure

An invariant is a claim about execution. Changing that claim must leave the program's behavior unchanged.

- `ACmd` stores commands and proposed invariants for the VC generator.
- `Cmd` stores the same commands without invariant fields for the execution rules.

```
1 def erase {S : Type} : ACmd S → Cmd S
2   | .skip => .skip
3   | .assign f => .assign f
4   | .seq c₁ c₂ => .seq (erase c₁) (erase c₂)
5   | .while _ b body => .while b (erase body)
```

Role in soundness: conditions generated from `c` must establish correctness of the underlying program `erase c`.

For the counting loop, annotations $x \leq 10$ and $x = 0$ produce different obligations, but erase to the same program.

Defining execution

`Cmd` describes program syntax. We define its meaning through **execution rules**.

`Exec c s t` means that command `c` can terminate from initial state `s` with final state `t`. The first three constructors are:

```
1 inductive Exec {S : Type} : Cmd S → S → S → Prop where
2   | skip (s) : Exec .skip s s
3   | assign (f) (s) : Exec (.assign f) s (f s)
4   | seq {c₁ c₂ s u t} :
5       Exec c₁ s u → Exec c₂ u t → Exec (.seq c₁ c₂) s t
```

- **Skip** leaves the state unchanged; **assignment** changes `s` to `f s`.
- **Sequence** connects two executions through an intermediate state `u`.

For `S = Nat` and `f x = x + 1`, the assignment rule gives `Exec (.assign f) 3 4`.

Execution rules for loops

The remaining constructors of `Exec` describe stopping and continuing a loop:

```
1  | whileFalse {b body s} :
2      b s = false → Exec (.while b body) s s
3  | whileTrue {b body s u t} :
4      b s = true → Exec body s u → Exec (.while b body) u t →
5      Exec (.while b body) s t
```

- **Guard false:** the loop stops immediately, leaving the state unchanged.
- **Guard true:** execute the body from `s` to `u`, then the remaining loop from `u` to `t`.

A proof of `Exec` is a **finite derivation** built from these rules. A nonterminating run has no such derivation.

The rules depend on the guard and body; they do not consult an invariant.

Partial correctness

Using these execution rules, the Hoare triple $\{P\}\ c\ \{Q\}$ is defined as:

```
1 def Hoare {S : Type} (P : S → Prop) (c : Cmd S)
2     (Q : S → Prop) : Prop :=
3   ∀ s t, P s → Exec c s t → Q t
```

- `∀ s t` covers every possible initial and final state.
- Given `P s` and an execution `Exec c s t`, the triple requires the postcondition `Q t`.
- Execution is a premise: this definition does **not** prove termination.

While rule

$$\frac{\{I \wedge b\}\ C\ \{I\}}{\{I\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{I \wedge \neg b\}}$$

`hoare_while` is proved by induction on a finite `Exec` derivation.

- **Exit:** the invariant still holds and the guard is false, giving $I \wedge \neg b$.
- **Iteration:** the body re-establishes I; the induction hypothesis applies to the remaining loop execution.

Entry assertions and side conditions

Given a command `c` and a desired postcondition `Q`, compute:

```
1 pre : ACmd S → (S → Prop) → (S → Prop)
2 VC  : ACmd S → (S → Prop) → Prop
```

Result	Role
`pre c Q`	A predicate to establish before executing `c`
`VC c Q`	Logical conditions that make this entry predicate sufficient

For a loop, `pre` uses the supplied invariant. Its sufficiency depends on `VC`; it need not be the weakest precondition.

Entry-assertion calculation

```
1 def pre {S : Type} : ACmd S → (S → Prop) → (S → Prop)
2   | .skip, Q => Q
3   | .assign f, Q => fun s => Q (f s)
4   | .seq c₁ c₂, Q => pre c₁ (pre c₂ Q)
5   | .while I _ _, _ => I
```

- **Assignment:** require `Q (f s)` in the original state `s`.
- **Sequence:** calculate `pre c₂ Q`, then use it as the target for `c₁`.
- **Loop:** use the supplied I; `VC` checks preservation and exit.

Entry assertion for a sequence

```
1 { 2 * (x + 1) ≥ 10 }
2 x := x + 1;
3 { 2 * x ≥ 10 }
4 x := 2 * x
5 { x ≥ 10 }
```

$$\text{pre}(c_1 ; c_2 , Q) = \text{pre}(c_1 , \text{pre}(c_2 , Q))$$

1. Substitute `2*x` into the final assertion `x ≥ 10`.
2. Substitute `x+1` into the resulting assertion `2*x ≥ 10`.

VCs for skip, assignment, and sequence

```
1 | .skip, _ => True
2 | .assign _, _ => True
3 | .seq c₁ c₂, Q => VC c₁ (pre c₂ Q) ∧ VC c₂ Q
```

- **Skip and assignment:** no loop side conditions; their effects are already accounted for in `pre`.
- **Sequence:** check `c₂` against `Q`, and `c₁` against `pre c₂ Q`.
- A separate entry obligation must still establish $P \Rightarrow \mathrm{pre}(c, Q)$.

VCs for loops

```
1 | .while I b body, Q =>
2     VC body I ∧
3     (∀ s, I s ∧ b s = true → pre body I s) ∧
4     (∀ s, I s ∧ b s = false → Q s)
```

Line	Obligation
2	Verify the body's side conditions with target I
3	I and the true guard must establish the body's entry assertion
4	I and the false guard must imply Q

Initialization comes from `pre loop Q = I`: entering the loop requires its invariant.

Verification obligation

```
1 def Verify {S : Type} (P : S → Prop) (c : ACmd S)
2     (Q : S → Prop) : Prop :=
3   (∀ s, P s → pre c Q s) ∧ VC c Q
```

- **Entry:** the supplied precondition `P` must imply the calculated requirement `pre c Q`.
- **Side conditions:** every proposition collected by `VC c Q` must hold.

A proof of `Verify P c Q` supplies both parts to the soundness theorem.

Counting VCs in Lean

`count I`: counting program with invariant `I`.

```
1 theorem count_vcs :
2     Verify (fun _ => True) (count (fun x => x ≤ 10)) (fun x => x = 10) := by
3   simp [Verify, count, pre, VC]
4   constructor <;> intros <;> omega
```

- `simp` unfolds the generator and simplifies the initialization condition.
- `constructor` separates preservation and exit; `intros` introduces hypotheses; `omega` proves the arithmetic.

Soundness converts this proof of generated conditions into a Hoare triple.

Omitting preservation from VC generation

`bugVC` loop clause

```
1 | .while I b body, Q =>
2     bugVC body I ∧ (∀ s, I s ∧ b s = false → Q s)
```

Omitted condition

$$I(s) \wedge b(s) = \text{true} \quad \Rightarrow \quad \text{pre}(\text{body}, I)(s)$$

Without preservation, the generator never requires the body to re-establish I.

`BugVerify` uses `bugVC` for side conditions and keeps the same entry calculation as `Verify`.

Counterexample program

```
1 { True }
2 x := 0;
3 while x < 10 invariant x = 0 do
4     x := x + 1
5 { x = 0 }
```

Final state

$$x = 10$$

Claimed postcondition

$$x = 0$$

The proposed invariant $x = 0$ holds initially, but the first iteration makes it false. The program still executes to $x = 10$.

Conditions generated by bugVC

Check	Formula	Status
Initialization	$0 = 0$	Kept; true
Exit	$(x = 0 \land x \geq 10) \Rightarrow x = 0$	Kept; true
Preservation	$(x = 0 \land x < 10) \Rightarrow x + 1 = 0$	**Omitted; false at $x = 0$**

- The exit implication is vacuously true: $x = 0 \land x \geq 10$ is impossible.
- At $x = 0$, the omitted preservation check would require $1 = 0$.

Proof of BugVerify

```
1 theorem bug_accepts :
2     BugVerify (fun _ => True) (count (fun x => x = 0)) (fun x => x = 0) := by
3   simp [BugVerify, count, pre, bugVC]
```

- `simp` succeeds because every emitted condition is true.
- The theorem proves `BugVerify` for this input. The corresponding semantic contract `Hoare P (erase c) Q` is false.

Checking a proof of the generated formula does not establish that the generator is sound.

Unsoundness of bugVC

$$0 \longrightarrow 1 \longrightarrow \cdots \longrightarrow 10 \neq 0$$

```
1  theorem bug_not_sound :
2      ¬ (∀ (P Q : Nat → Prop) (c : ACmd Nat),
3           BugVerify P c Q → Hoare P (erase c) Q) := by
4    intro h
5    exact bad_contract_false (h _ _ _ bug_accepts)
```

- `bad_contract_false` uses the `Exec` derivation ending at ten to refute the contract.
- A universal soundness claim would turn `bug_accepts` into that false Hoare triple.

Soundness of VC

```
1 vc_sound (c : ACmd S) (Q : S → Prop) (hvc : VC c Q) :
2   Hoare (pre c Q) (erase c) Q
```

- Assuming `VC c Q`, every terminating execution starting in `pre c Q` must end in `Q`.
- **Proof:** structural induction on `c`. Each recursive call in the generator is justified by an induction hypothesis.

Generalizing the postcondition

```
1  induction c generalizing Q with
```

Recursive call	Required postcondition
First command c_1	`pre c₂ Q`
Second command c_2	`Q`
Loop body	`I`

If `Q` is fixed before induction, the hypothesis for c_1 only covers that postcondition.

`generalizing Q` lets the proof instantiate it with `pre c₂ Q`, or with `I` for a loop body.

Assignment and sequence cases

```
1 | assign f => exact hoare_assign Q f
2 | seq c₁ c₂ ih₁ ih₂ =>
3     exact hoare_seq (ih₁ _ hvc.1) (ih₂ _ hvc.2)
```

- **Assignment:** unfolding `pre` gives exactly the triple proved by `hoare_assign`.
- **Sequence:** `ih₁` proves that `c₁` establishes **`pre c₂ Q`**; `ih₂` uses that assertion to establish `Q` after `c₂`.

`hoare_seq` composes the two triples.

Loop case

$C = \text{erase}(\text{body})$

$\{\text{pre}(\text{body}, I)\}\ C\ \{I\}$	body IH + body VCs
$\{I \wedge b\}\ C\ \{I\}$	preservation + consequence
$\{I\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{I \wedge \neg b\}$	while rule
$\{I\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{Q\}$	exit + consequence

Preservation strengthens the body's precondition. Exit weakens the loop's postcondition to Q.

Soundness of Verify

```
theorem verify_sound {S : Type} (P : S → Prop) (c : ACmd S)
    (Q : S → Prop) (h : Verify P c Q) :
    Hoare P (erase c) Q :=
  hoare_consequence h.1 (vc_sound c Q h.2) (fun _ hq => hq)
```

`h.1` establishes `pre c Q` from `P`; `h.2` supplies the conditions required by `vc_sound`.

Application to counting

```
theorem count_correct :
    Hoare (fun _ => True) (erase (count (fun x => x ≤ 10)))
      (fun x => x = 10) :=
  verify_sound _ _ _ count_vcs
```

Division example

```
1 { True }
2 q := 0; r := n;
3 while d ≤ r invariant q*d + r = n do
4     r := r − d;
5     q := q + 1
6 { q*d + r = n ∧ r < d }
```

State: $q, r \in \mathbb{N}$ · Fixed inputs: $n, d \in \mathbb{N}$

Each iteration transfers d from the remainder to qd, preserving $qd + r = n$. The false guard supplies the final bound $r < d$.

Verification conditions for division

Initialization

$$0 \cdot d + n = n$$

Preservation

$$(qd + r = n \wedge d \leq r) \Rightarrow (q + 1)d + (r - d) = n$$

Exit

$$(qd + r = n \wedge \neg(d \leq r)) \Rightarrow (qd + r = n \wedge r < d)$$

The guard $d \leq r$ permits cancellation in natural-number subtraction. Expand $(q + 1)d$, then discharge the remaining arithmetic with `omega`.

Division correctness

$$\mathrm{Post}(n, d)(q, r) \equiv qd + r = n \wedge r < d$$

```
1 theorem division_correct (n d : Nat) :
2     Hoare (fun _ => True) (erase (division n d)) (Post n d) :=
3   verify_sound _ _ _ (division_vcs n d)
```

- `division_vcs n d` proves the generated arithmetic obligations.
- `verify_sound` converts that proof into the quotient/remainder contract.

This is partial correctness: for $d = 0$, the loop never terminates.

Insufficient invariants

Counting program, postcondition $x = 10$:

Invariant	Exit condition	Result
`True`	$x \geq 10 \Rightarrow x = 10$	False at $x = 11$
$x \leq 10$	$(x \leq 10 \wedge x \geq 10) \Rightarrow x = 10$	True

$x = 11$ is unreachable, but invariant `True` does not exclude it.

A failed obligation can indicate an insufficient invariant even when the program satisfies its contract.

Formalization and results

	Lean definitions and theorems
Syntax and semantics	`ACmd`, `erase`, `Exec`
VC generation	`pre`, `VC`, `Verify`
Soundness	`vc_sound`, `verify_sound`
Applications	`count_correct`, `division_correct`

- The soundness theorem connects a recursive Lean function to the program's execution semantics.
- Each application proves its generated conditions, then obtains correctness by applying that theorem.

The guarantee is **partial correctness under the defined semantics**; termination requires a separate proof.

Optional: loop case of vc_sound

```
1  | «while» I b body ih =>
2      have hbody := hoare_consequence hvc.2.1 (ih I hvc.1)
3        (fun _ h => h)
4      exact hoare_consequence (fun _ h => h)
5        (hoare_while I b (erase body) hbody) hvc.2.2
```

`hvc.1`	`hvc.2.1`	`hvc.2.2`
Body VCs	Preservation	Exit

- `hbody` combines the body induction hypothesis with preservation to prove $\{I \wedge b\}\ C\ \{I\}$.
- `hoare_while` handles repetition; the final consequence step uses the exit implication.