“Beware of bugs in the above code; I have only proved it correct, not tried it.”

— Donald Knuth

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 5 of 10

Proofs & Semantics

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/

Section 1

Week 4 Review

What We Covered Last Week

Dr. Nadim Kobeissi's lectures on proof techniques:

Backward Proofs (Tactic Mode):

- Working from goal to hypotheses
- Basic tactics: intro, apply, exact, assumption
- Reasoning about connectives and equality
- Mathematical induction

Forward Proofs (Structured Proofs):

- Working from hypotheses to goal
- Structured constructs: fix, assume, show, have
- Calculational proofs with calc
- The Propositions-as-Types (PAT) principle

Section 2

Backward Proofs: Tactic Mode

Two Proof Styles in Lean

Backward (Tactic) Proofs:

- Start from the goal
- Work backwards to hypotheses
- Interactive, step-by-step refinement
- Use tactics to transform goals

Forward (Structured) Proofs:

- Start from hypotheses
- Build toward the goal
- More like traditional math proofs
- Explicit intermediate steps

Both styles can be mixed and are equally powerful!

Tactic Mode Syntax

Keyword: by indicates tactical proof

```
theorem example_theorem :
    statement := by
  tactic1
  tactic2
  ...
  tacticN
  done
```

Components:

- by - enters tactic mode
- Each tactic transforms proof state
- done - optional, marks proof completion

Section 2

Backward Proofs: Tactic Mode

Subsection 2.1

Basic Tactics

The intro Tactic

Purpose: Move assumptions into context

```
theorem fst_of_two_props :
    ∀a b : Prop, a → b → a := by
  intro a b    -- Move ∀-quantified variables
  intro ha hb  -- Move implications
  apply ha
  done
```

What intro does:

- Moves $\forall$-quantified variables from goal to context
- Moves implications ($\rightarrow$) into hypotheses
- Changes goal from $A \rightarrow B$ to just B (with A as hypothesis)

The apply Tactic

Purpose: Match goal with theorem's conclusion

```
theorem prop_comp (a b c : Prop)
    (hab : a → b) (hbc : b → c) :
    a → c := by
  intro ha
  apply hbc  -- Goal c matches conclusion of hbc
  apply hab  -- New goal b matches conclusion of hab
  apply ha   -- New goal a matches ha
  done
```

Backward reasoning:

- To prove c, use $b \to c$ (creates goal b)
- To prove b, use $a \to b$ (creates goal a)
- To prove a, we already have it!

The exact Tactic

Purpose: Provide exact proof term for goal

```
theorem fst_of_two_props_exact (a b : Prop)
    (ha : a) (hb : b) :
    a := by
  exact ha  -- ha is exactly what we need
  done
```

Difference from apply:

- `apply`: Can create new subgoals
- `exact`: Must completely solve the goal
- `exact`: Communicates intent more clearly

The assumption Tactic

Purpose: Find matching hypothesis automatically

```
theorem fst_of_two_props_assumption (a b : Prop)
    (ha : a) (hb : b) :
    a := by
  assumption  -- Searches context for matching hypothesis
  done
```

How it works:

- Searches local context for hypothesis matching goal
- Applies it automatically
- Useful when there are many hypotheses

Section 2

Backward Proofs: Tactic Mode

Subsection 2.2

Reasoning About Logical Connectives

Introduction Rules for Connectives

Building proofs of compound propositions:

```
#check True.intro          -- : True
#check And.intro           -- : a → b → a ∧ b
#check Or.inl              -- : a → a ∨ b
#check Or.inr              -- : b → a ∨ b
#check Iff.intro           -- : (a → b) → (b → a) → (a ↔ b)
#check Exists.intro        -- : ∀(a : α), P a → ∃x, P x
```

These are constructors - they build proofs

Elimination Rules for Connectives

Extracting information from compound propositions:

```
#check False.elim          -- : False → a
#check And.left            -- : a ∧ b → a
#check And.right           -- : a ∧ b → b
#check Or.elim             -- : a ∨ b → (a → c) → (b → c) → c
#check Iff.mp              -- : (a ↔ b) → a → b
#check Iff.mpr             -- : (a ↔ b) → b → a
#check Exists.elim         -- : (∃x, P x) → (∀x, P x → c) → c
```

These are destructors - they extract information

Proving Conjunction (And)

```
theorem And_swap (a b : Prop) :
    a ∧ b → b ∧ a := by
  intro hab
  apply And.intro
  { apply And.right
    exact hab }
  { apply And.left
    exact hab }
  done
```

Steps:

1. Assume we have $a \wedge b$
2. To prove $b \wedge a$, use `And.intro`
3. First subgoal: prove b (extract from $a \wedge b$)
4. Second subgoal: prove a (extract from $a \wedge b$)

The Focus Combinator

Use braces to focus on subgoals:

```
theorem And_swap (a b : Prop) :
    a ∧ b → b ∧ a := by
  intro hab
  apply And.intro
  { exact And.right hab }  -- First subgoal
  { exact And.left hab }   -- Second subgoal
  done
```

Benefits of { ... }:

- Structure proofs clearly
- Each subgoal completely solved inside braces
- Error if subgoal not fully proven

Proving Disjunction (Or)

```
theorem Or_swap (a b : Prop) :
    a ∨ b → b ∨ a := by
  intro hab
  apply Or.elim hab
  { intro ha
    exact Or.inr ha }
  { intro hb
    exact Or.inl hb }
  done
```

Case analysis on disjunction:

- `Or.elim`: Split into two cases
- Case 1: If we have a, prove goal
- Case 2: If we have b, prove goal

Negation in Lean

Negation is defined as implication to False:

```
#print Not  -- def Not (a : Prop) : Prop := a → False

theorem Not_Not_intro (a : Prop) :
    a → ¬¬ a := by
  intro ha hna
  apply hna
  exact ha
  done
```

Key insight:

- $\neg a$ means $a \to \texttt{False}$
- To prove $\neg a$, assume a and derive contradiction
- $\neg\neg a$ means $(a \to \texttt{False}) \to \texttt{False}$

Existential Quantification

```
def double (n : ℕ) : ℕ := n + n

theorem Exists_double_iden :
    ∃n : ℕ, double n = n := by
  apply Exists.intro 0
  rfl
  done
```

To prove $\exists x$, P(x):

- Provide witness: `Exists.intro witness`
- Prove P holds for that witness
- Here: $n = 0$ and `double` $0 = 0$

Section 2

Backward Proofs: Tactic Mode

Subsection 2.3

Reasoning About Equality

Three Kinds of Equality

1. **Syntactic equality:** $x = x$, $[2, 1, 3] = [2, 1, 3]$
   - Literally the same expression
2. **Definitional equality:** $2 + 2 = 4$
   - Equal up to computation/reduction
   - Proved by `rfl`
3. **Propositional equality:** $x + y = y + x$
   - Requires proof
   - Not automatically true

The rfl Tactic

rfl: Proves definitional equality

```
-- β-reduction (function application)
theorem β_example {α β : Type} (f : α → β) (a : α) :
    (fun x => f x) a = f a := by
  rfl

-- δ-reduction (definition unfolding)
def double (n : ℕ) : ℕ := n + n

theorem δ_example :
    double 5 = 5 + 5 := by
  rfl
```

rfl works when: Both sides compute to same value

The rw Tactic

rw: Rewrite using equation

```
theorem Eq_trans_symm_rw {α : Type} (a b c : α)
    (hab : a = b) (hcb : c = b) :
    a = c := by
  rw [hab]   -- Replace a with b
  rw [hcb]   -- Replace c with b
  done
```

Features:

- Apply equation left-to-right
- Use ← for right-to-left: `rw [←hab]`
- Can expand definitions

The simp Tactic

simp: Simplify using standard rewrites

```
theorem cong_example {α : Type} (a b c d : α)
    (g : α → α → ℕ → α)
    (hab : a = b) (hcd : c = d) :
    g a c (1 + 1) = g b d 2 := by
  simp [hab, hcd]
```

What simp does:

- Applies standard simplification rules
- Can add custom rules: `simp [theorem1, theorem2]`
- Works exhaustively until no more simplifications

The ac_rfl Tactic

ac_rfl: Associativity and commutativity

```
theorem abc_Eq_cba (a b c : ℕ) :
    a + b + c = c + b + a := by
  ac_rfl
```

What it handles:

- Reordering terms: $a + b = b + a$
- Regrouping: $(a + b) + c = a + (b + c)$
- Works for $+$, $*$, and registered operators

Section 2

Backward Proofs: Tactic Mode

Subsection 2.4

Mathematical Induction

The induction Tactic

Principle: Prove property for all values by cases

```
theorem add_zero (n : ℕ) :
    add 0 n = n := by
  induction n with
  | zero       => rfl
  | succ n' ih => simp [add, ih]
```

Two cases:

- **Base case** (`zero`): Prove for 0
- **Inductive case** (`succ`): Prove for $n' + 1$ assuming true for n'
- `ih` is the induction hypothesis

Induction Example: Commutativity

```
theorem add_comm (m n : ℕ) :
    add m n = add n m := by
  induction n with
  | zero       =>
      simp [add, add_zero]
  | succ n' ih =>
      simp [add, add_succ, ih]
```

Pattern:

1. Base case: Usually simple (often `rfl`)
2. Inductive case: Use `ih` (induction hypothesis)
3. Simplify with `simp` using relevant lemmas

Section 3

Forward Proofs: Structured Constructs

Forward vs Backward Reasoning

Backward (we just saw):

- Start with goal
- Break it down into subgoals
- Work until we reach axioms/hypotheses

Forward (now):

- Start with what we know
- Build up intermediate results
- Eventually reach the goal

Both styles are useful! Choose based on problem.

Structured Proof Keywords

Key constructs:

`fix`: Introduce variables (like `intro`)
`assume`: Introduce hypotheses (like `intro`)
`show ... from`: State and prove goal
`have`: Prove intermediate result

```
theorem fst_of_two_props :
    ∀a b : Prop, a → b → a :=
  fix a b : Prop
  assume ha : a
  assume hb : b
  show a from ha
```

The fix Keyword

fix: Introduce universally quantified variables

```
theorem example_fix :
    ∀a b : Prop, ... :=
  fix a b : Prop
  ...
```

Like intro but forward-style:

- Moves variables into context
- More explicit about types
- Can introduce multiple at once

The assume Keyword

assume: Introduce hypothesis

```
theorem modus_ponens (a b : Prop) :
    (a → b) → a → b :=
  assume hab : a → b
  assume ha : a
  show b from
    hab ha
```

Forward reasoning:

- Start by assuming hypotheses
- Build proof from what we have
- Eventually show the goal

The show Keyword

show ... from: State goal explicitly

```
theorem example_show (a b : Prop) (ha : a) (hb : b) :
    a :=
  show a from ha
```

Benefits:

- Documents what we're proving
- Can rephrase goal (up to computation)
- Makes proof more readable

Optional: Can omit if obvious: just `ha`

The have Keyword

have: Prove intermediate lemma

```
theorem prop_comp (a b c : Prop)
    (hab : a → b) (hbc : b → c) :
    a → c :=
  assume ha : a
  have hb : b := hab ha
  have hc : c := hbc hb
  show c from hc
```

Forward chaining:

- Build intermediate results
- Each have adds to context
- Chain reasoning step by step

Forward vs Backward: Same Proof

Backward style:

```
theorem And_swap_tactical (a b : Prop) :
    a ∧ b → b ∧ a := by
  intro hab
  apply And.intro
  { exact And.right hab }
  { exact And.left hab }
```

Forward style:

```
theorem And_swap (a b : Prop) :
    a ∧ b → b ∧ a :=
  assume hab : a ∧ b
  have ha : a := And.left hab
  have hb : b := And.right hab
  show b ∧ a from And.intro hb ha
```

Calculational Proofs

calc: Chain equalities/inequalities

```
theorem two_mul_example (m n : ℕ) :
    2 * m + n = m + n + m :=
  calc
    2 * m + n = m + m + n := by rw [Nat.two_mul]
    _         = m + n + m := by ac_rfl
```

Benefits:

- Clear chain of reasoning
- Each step explicitly justified
- Like handwritten math proofs

Mixing Styles

Can use backward proofs within forward proofs:

```
theorem example_mixed (a b : Prop) :
    (∀x, x = true → a) → a :=
  assume hall : ∀x, x = true → a
  show a from
    by
      apply hall true
      rfl
```

Best of both worlds:

- High-level structure: forward
- Low-level details: backward tactics
- Choose what's clearest

Section 4

The PAT Principle

Propositions as Types (PAT)

Deep insight: Programs and proofs are the same thing!

Programs	$\leftrightarrow$	**Proofs**
Types	$\leftrightarrow$	**Propositions**
$\rightarrow$	$\leftrightarrow$	$\Longrightarrow$
Execution	$\leftrightarrow$	**Verification**

This is the foundation of proof assistants like Lean!

PAT: Types as Propositions

Every type is a proposition:

- Type $A \to B$ = Proposition "A implies B"
- Type $A \times B$ = Proposition "A and B"
- Type $A + B$ = Proposition "A or B"
- Empty type = False
- Unit type = True

A type is inhabited if and only if the corresponding proposition is provable!

PAT: Terms as Proofs

Every term is a proof:

- Term of type $A \to B$ = Proof that A implies B
- Function application = Modus ponens
- Lambda abstraction = Proof of implication
- Pair = Proof of conjunction
- Sum = Proof of disjunction

Type checking = Proof checking!

PAT Example: Modus Ponens

As a logical rule:

$$\frac{A \quad A \to B}{B}$$

As a function:

```
theorem modus_ponens {A B : Prop}
    (h1 : A → B) (h2 : A) : B :=
  h1 h2  -- Function application!
```

Key insight:

- $h1 : A \to B$ is a function from A to B
- $h2 : A$ is an input
- $h1\ h2 : B$ is function application

PAT: Universal Quantification

Remarkably: $\forall$ is just a dependent function!

$$\forall x : \sigma, P(x) \quad := \quad (x : \sigma) \to P(x)$$

```
-- These are the same!
theorem example1 : ∀n : ℕ, n + 0 = n := ...
theorem example2 : (n : ℕ) → n + 0 = n := ...
```

$\forall$ **is not primitive** - it's defined as dependent functions!

PAT: Proof Terms

Tactics compile to proof terms:

```
-- Tactical proof
theorem And_swap_tactical (a b : Prop) :
    a ∧ b → b ∧ a := by
  intro hab
  exact And.intro (And.right hab) (And.left hab)

-- Raw proof term (what tactics compile to)
theorem And_swap_raw (a b : Prop) :
    a ∧ b → b ∧ a :=
  fun hab => And.intro (And.right hab) (And.left hab)
```

Both are exactly the same to Lean!

Why PAT Matters

Unification of concepts:

- No separate "proof language"
- All of type theory applies to proofs
- Can reason about proofs as data

Practical benefits:

- Type checker = Proof checker
- Programming features work for proofs
- Proofs can compute
- Can extract programs from proofs

This is the foundation of dependent type theory!

Section 5

Inductive Predicates

What are Inductive Predicates?

Inductive predicate: Function of type $\cdots \to \texttt{Prop}$

Like inductive types, but for propositions:

- Define by introduction rules
- "No junk, no confusion"
- Can do induction on proof terms

Think of them as:

- Formal systems (logic)
- Horn clauses (Prolog)
- Inference rules (proof theory)

Example: Even Numbers

Mathematical definition:

The set E of even natural numbers is the smallest set closed under: (1) $0 \in E$ and (2) if $k \in E$ then $k + 2 \in E$.

In Lean:

```
inductive Even : ℕ → Prop where
  | zero    : Even 0
  | add_two : ∀k : ℕ, Even k → Even (k + 2)
```

Two introduction rules construct all even number proofs!

Using Even: Proving 4 is Even

```
theorem Even_4 : Even 4 := by
  apply Even.add_two
  apply Even.add_two
  exact Even.zero
```

Proof tree:

- $4 = 2 + 2$, so use `Even.add_two` with $k = 2$
- Need to prove `Even 2`
- $2 = 0 + 2$, so use `Even.add_two` with $k = 0$
- Need to prove `Even 0`
- Use `Even.zero`

Three Ways to Define "Even"

1. **Inductive predicate:**
   - Abstract, elegant
   - Independent rules
   - Natural for proofs
2. **Recursive function:**
   - Computational
   - Works with `#eval`
   - Need to handle all cases
3. **Direct definition:**
   - Most "mathematical"
   - `def evenNonrec (k : ℕ) : Prop := k % 2 = 0`

Each has its uses! Inductive predicates excel at complex specifications.

Example: Tennis Scores

Model valid tennis scores:

```
inductive Score : Type where
  | vs       : ℕ → ℕ → Score
  | advServ  : Score
  | advRecv  : Score
  | gameServ : Score
  | gameRecv : Score

infixr:50 " – " => Score.vs
```

Transition system: Which scores can follow from 0-0?

Tennis Score Transitions

```
inductive Step : Score → Score → Prop where
  | serv_0_15    : ∀n, Step (0–n) (15–n)
  | serv_15_30   : ∀n, Step (15–n) (30–n)
  | serv_30_40   : ∀n, Step (30–n) (40–n)
  | serv_40_game : ∀n, n < 40 → Step (40–n) Score.gameServ
  | serv_40_adv  : Step (40–40) Score.advServ
  | recv_0_15    : ∀n, Step (n–0) (n–15)
  ...
```

Binary predicate: Connects "before" and "after" states

Reflexive Transitive Closure

Star: Reflexive transitive closure of any relation

```
inductive Star {α : Type} (R : α → α → Prop) :
    α → α → Prop where
  | base (a b : α)    : R a b → Star R a b
  | refl (a : α)      : Star R a a
  | trans (a b c : α) :
      Star R a b → Star R b c → Star R a c
```

Three rules:

- `base`: Embed R into `Star` R
- `refl`: Reflexivity
- `trans`: Transitivity

Logical Symbols as Inductive Predicates

Most logical symbols are defined inductively!

- $\wedge$ (And): One constructor taking both props
- $\vee$ (Or): Two constructors (left or right)
- $\leftrightarrow$ (Iff): One constructor taking both directions
- $\exists$ (Exists): One constructor with witness
- `True`: One constructor (trivial proof)
- `False`: Zero constructors (no proof!)
- $=$ (Eq): One constructor (reflexivity)

Only $\forall$ and $\rightarrow$ are built into the logic!

Definition of And

```
inductive And (a b : Prop) : Prop where
  | intro : a → b → And a b
```

To prove $a \wedge b$:

- Must use `And.intro`
- Need proof of a
- Need proof of b

To use hypothesis $h : a \wedge b$:

- Extract left: `And.left h : a`
- Extract right: `And.right h : b`

Definition of Or

```
inductive Or (a b : Prop) : Prop where
  | inl : a → Or a b
  | inr : b → Or a b
```

To prove $a \lor b$:

- Either use `Or.inl` (prove a)
- Or use `Or.inr` (prove b)

To use hypothesis $h : a \lor b$:

- Case analysis with `Or.elim`
- Handle both possibilities

Definition of Exists

```
inductive Exists {α : Type} (P : α → Prop) : Prop where
  | intro : ∀a : α, P a → Exists P
```

Syntactic sugar: $\exists x : \alpha, P(x) :=$ `Exists (λx : α, P)`

To prove $\exists x, P(x)$:

- Provide witness a
- Prove $P(a)$

To use hypothesis $h : \exists x, P(x)$:

- Eliminate to get witness and proof

Definition of Equality

```
inductive Eq {α : Type} : α → α → Prop where
  | refl : ∀a : α, Eq a a
```

Only one constructor: Reflexivity!

From just refl, we can prove:

- Symmetry: $a = b \to b = a$
- Transitivity: $a = b \to b = c \to a = c$
- Substitution: $a = b \to P(a) \to P(b)$

All equality properties follow from reflexivity!

Rule Induction

Just as we induct on values, we can induct on proofs!

Called rule induction because:

- Induction on introduction rules
- One case per constructor
- Induction hypothesis for recursive constructors

By PAT: This is just structural induction on proof terms!

Rule Induction Example

```
theorem mod_two_Eq_zero_of_Even (n : ℕ) (h : Even n) :
    n % 2 = 0 := by
  induction h with
  | zero =>
      rfl
  | add_two k hk ih =>
      simp [Nat.add_mod, ih]
```

Two cases from Even constructors:

- `zero`: Prove $0 \% 2 = 0$
- `add_two`: Assume $k \% 2 = 0$ (ih), prove $(k + 2) \% 2 = 0$

Example: Sorted Lists

```
inductive Sorted : List ℕ → Prop where
  | nil : Sorted []
  | single (x : ℕ) : Sorted [x]
  | two_or_more (x y : ℕ) {zs : List ℕ}
      (hle : x ≤ y)
      (hsorted : Sorted (y :: zs)) :
    Sorted (x :: y :: zs)
```

Three constructors:

- Empty list is sorted
- Single element is sorted
- Two or more: first $\leq$ second, rest sorted

Proving Sorted Lists

```
theorem sorted_7_9_9_11 :
    Sorted [7, 9, 9, 11] := by
  apply Sorted.two_or_more
  { simp }  -- 7 ≤ 9
  { apply Sorted.two_or_more
    { simp }  -- 9 ≤ 9
    { apply Sorted.two_or_more
      { simp }  -- 9 ≤ 11
      { exact Sorted.single _ } } }
```

Build proof step by step using constructors

Example: Palindromes

```
inductive Palindrome {α : Type} : List α → Prop where
  | nil : Palindrome []
  | single (x : α) : Palindrome [x]
  | sandwich (x : α) (xs : List α)
      (hxs : Palindrome xs) :
    Palindrome ([x] ++ xs ++ [x])
```

Recursive structure:

- Empty and single elements are palindromes
- Sandwich: same element on both ends + palindrome middle

Example: Full Binary Trees

```
inductive IsFull {α : Type} : BTree α → Prop where
  | empty : IsFull BTree.empty
  | node (a : α) (l r : BTree α)
      (hl : IsFull l) (hr : IsFull r)
      (hiff : l = BTree.empty ↔ r = BTree.empty) :
    IsFull (BTree.node a l r)
```

Full tree property:

- Empty tree is full
- Node is full if: both subtrees full, and either both empty or both non-empty

Section 6

Operational Semantics

What is Formal Semantics?

Formal semantics: Mathematical specification of what programs mean

Why formalize?

- Specify programming languages precisely
- Reason about programs mathematically
- Build verified compilers, interpreters, analyzers
- Find bugs in language specifications

Success story: WebAssembly

- Formalization revealed multiple soundness bugs
- Fixed before becoming standards

Why Proof Assistants for Semantics?

Good match for automation:

- Little background machinery needed
- Lots of cases (computers excel at this)
- Track changes when extending language
- Catch subtle bugs

Real world: 30%+ of papers at the Principles of Programming Languages (POPL) conference use proof assistants

In this lecture: Build operational semantics for a simple language

The WHILE Language

Minimalistic imperative language:

$$
\begin{aligned}
S ::= \ & \texttt{skip} && \text{(no-op)} \\
\mid \ & x := a && \text{(assignment)} \\
\mid \ & S; S && \text{(sequence)} \\
\mid \ & \texttt{if } B \texttt{ then } S \texttt{ else } S && \text{(conditional)} \\
\mid \ & \texttt{while } B \texttt{ do } S && \text{(loop)}
\end{aligned}
$$

State: Function from variable names to values

- $\texttt{State} = \texttt{String} \to \mathbb{N}$

WHILE in Lean

```
inductive Stmt : Type where
  | skip       : Stmt
  | assign     : String → (State → ℕ) → Stmt
  | seq        : Stmt → Stmt → Stmt
  | ifThenElse : (State → Prop) → Stmt → Stmt → Stmt
  | whileDo    : (State → Prop) → Stmt → Stmt

infixr:90 "; " => Stmt.seq
```

Design choice: Shallow embedding

- Expressions are Lean functions: `State → ℕ`
- Conditions are predicates: `State → Prop`
- Simpler than deep embedding (ASTs)

Example WHILE Program

```
-- while x > y do
--   skip;
--   x := x - 1

def sillyLoop : Stmt :=
  Stmt.whileDo (fun s => s "x" > s "y")
    (Stmt.skip;
     Stmt.assign "x" (fun s => s "x" - 1))
```

What it does:

- While $x > y$
- Decrement x
- Eventually $x \leq y$

Two Kinds of Operational Semantics

Big-step semantics (natural semantics):

- Judgment: $(S, s) \Rightarrow t$
- "Starting in state s, executing S terminates in state t"
- Direct: one step from start to finish

Small-step semantics (structural operational semantics):

- Judgment: $(S, s) \Rightarrow (S', s')$
- "One step of execution"
- Execution is a chain of steps

Both are useful! Different strengths and weaknesses.

Section 6

Operational Semantics

Subsection 6.1

Big-Step Semantics

Big-Step: The Idea

Judgment form: $(S, s) \Rightarrow t$

Read as:

- Starting in state s
- Executing statement S
- Terminates in state t

Example:

$$(x := x + y;\ y := 0,\ [x \mapsto 3, y \mapsto 5]) \Rightarrow [x \mapsto 8, y \mapsto 0]$$

Single judgment for entire execution!

Big-Step: Skip Rule

$$\frac{}{(\texttt{skip}, s) \Rightarrow s}$$

Skip does nothing:

- No premises (axiom)
- State unchanged
- Simplest rule

Big-Step: Assignment Rule

$$\frac{}{(x := a, s) \Rightarrow s[x \mapsto s(a)]}$$

Assignment updates state:

- Evaluate expression a in state s
- Update state: x maps to new value
- Notation: $s[x \mapsto v]$ means state with x updated to v

Example: $(x := 5, [x \mapsto 3]) \Rightarrow [x \mapsto 5]$

Big-Step: Sequence Rule

$$\frac{(S, s) \Rightarrow t \quad (T, t) \Rightarrow u}{(S;\, T, s) \Rightarrow u}$$

Sequential composition:

- Execute S first: $s \to t$
- Then execute T: $t \to u$
- Final state: u

Intermediate state t threads through!

Big-Step: Conditional Rules

$$\frac{(S, s) \Rightarrow t}{(\texttt{if } B \texttt{ then } S \texttt{ else } T, s) \Rightarrow t} \text{ if } s(B) \text{ is true}$$

$$\frac{(T, s) \Rightarrow t}{(\texttt{if } B \texttt{ then } S \texttt{ else } T, s) \Rightarrow t} \text{ if } s(B) \text{ is false}$$

Two rules depending on condition:

- If B true: execute then-branch
- If B false: execute else-branch

Big-Step: While Rules

$$\frac{(S, s) \Rightarrow t \quad (\texttt{while } B \texttt{ do } S, t) \Rightarrow u}{(\texttt{while } B \texttt{ do } S, s) \Rightarrow u} \text{ if } s(B) \text{ is true}$$

$$\frac{}{(\texttt{while } B \texttt{ do } S, s) \Rightarrow s} \text{ if } s(B) \text{ is false}$$

While loop:

- If condition false: done (state unchanged)
- If condition true: execute body, then loop again
- Recursive rule!

Big-Step in Lean

```
inductive BigStep : Stmt × State → State → Prop where
  | skip (s) :
    BigStep (Stmt.skip, s) s
  | assign (x a s) :
    BigStep (Stmt.assign x a, s) (s[x ↦ a s])
  | seq (S T s t u)
      (hS : BigStep (S, s) t)
      (hT : BigStep (T, t) u) :
    BigStep (S; T, s) u
  ...

infix:110 " => " => BigStep
```

Using Big-Step Semantics

```
theorem sillyLoop_from_1_BigStep :
    (sillyLoop, (fun _ => 0)["x" ↦ 1]) => (fun _ => 0) := by
  rw [sillyLoop]
  apply BigStep.while_true
  { simp }  -- Condition: x > y
  { apply BigStep.seq
    { apply BigStep.skip }
    { apply BigStep.assign } }
  { simp
    apply BigStep.while_false
    simp }
```

Build proof using introduction rules!

Properties of Big-Step Semantics

What can we prove?

1. **Determinism**:
   - If $(S, s) \Rightarrow t$ and $(S, s) \Rightarrow t'$
   - Then $t = t'$
   - Execution is deterministic!

2. **Program equivalence**:
   - Prove two programs equivalent
   - Same semantics in all states

Determinism of Big-Step

```
theorem BigStep_deterministic {Ss l r}
    (hl : Ss → l) (hr : Ss → r) :
    l = r := by
  induction hl generalizing r with
  | skip s =>
    cases hr with | skip => rfl
  | assign x a s =>
    cases hr with | assign => rfl
  | seq S T s l₀ l hS hT ihS ihT =>
    cases hr with
    | seq _ _ _ r₀ _ hS' hT' =>
      cases ihS hS' with | refl =>
        cases ihT hT' with | refl => rfl
  ...
```

What Big-Step Cannot Express

Limitations:

1. **Nontermination**:
   - No judgment for infinite loops
   - Can't prove $\exists t, (S, s) \Longrightarrow t$ for all programs
2. **Intermediate states**:
   - Only see start and end
   - Can't reason about what happens during execution
3. **Interleaving**:
   - Can't model concurrent execution

Solution: Small-step semantics!

Section 6

Operational Semantics

Subsection 6.2

Small-Step Semantics

Small-Step: The Idea

Judgment form: $(S, s) \Rightarrow (S', s')$

Read as:

- Starting in state s
- Executing one step of S
- Leaves us in state s'
- With program S' remaining

Execution: Chain of steps

$$(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow (S_2, s_2) \Rightarrow \cdots$$

Small-Step: Configuration

Configuration: Pair (S, s) of statement and state

Final configuration: No more steps possible

- For WHILE: Only $(\texttt{skip}, s)$ is final
- Execution terminates when we reach final configuration

Example execution:

$$
\begin{aligned}
&(x := x + y;\ y := 0,\ [x \mapsto 3, y \mapsto 5]) \\
\Rightarrow\ &(\texttt{skip};\ y := 0,\ [x \mapsto 8, y \mapsto 5]) \\
\Rightarrow\ &(y := 0,\ [x \mapsto 8, y \mapsto 5]) \\
\Rightarrow\ &(\texttt{skip},\ [x \mapsto 8, y \mapsto 0])
\end{aligned}
$$

Small-Step: Assignment Rule

$$\frac{}{(x := a, s) \Rightarrow (\texttt{skip}, s[x \mapsto s(a)])}$$

Assignment in one step:

- Evaluate and update state
- Leave `skip` to indicate completion
- No further evaluation needed

Small-Step: Sequence Rules

$$\frac{(S, s) \Rightarrow (S', s')}{(S;\ T, s) \Rightarrow (S';\ T, s')}$$

$$\frac{}{(\texttt{skip};\ S, s) \Rightarrow (S, s)}$$

Two rules:

- If first statement can step: step it
- If first statement is `skip`: discard it

Second statement stays unchanged until first completes!

Small-Step: Conditional Rules

$$\frac{}{(\texttt{if } B \texttt{ then } S \texttt{ else } T, s) \Rightarrow (S, s)} \text{ if } s(B) \text{ is true}$$

$$\frac{}{(\texttt{if } B \texttt{ then } S \texttt{ else } T, s) \Rightarrow (T, s)} \text{ if } s(B) \text{ is false}$$

One step to choose branch:

- Evaluate condition
- Replace with chosen branch
- State unchanged

Small-Step: While Rule

$$\frac{}{(\texttt{while } B \texttt{ do } S, s) \Rightarrow (\texttt{if } B \texttt{ then } (S; \texttt{while } B \texttt{ do } S) \texttt{ else skip}, s)}$$

Unfold while loop:

- Replace with conditional
- If true: body then loop again
- If false: `skip` (done)
- No evaluation here - just transformation

Note: There's no rule for `skip`! (It's final)

Small-Step in Lean

```
inductive SmallStep :
    Stmt × State → Stmt × State → Prop where
  | assign (x a s) :
    SmallStep (Stmt.assign x a, s)
              (Stmt.skip, s[x ↦ a s])
  | seq_step (S S' T s s')
      (hS : SmallStep (S, s) (S', s')) :
    SmallStep (S; T, s) (S'; T, s')
  | seq_skip (T s) :
    SmallStep (Stmt.skip; T, s) (T, s)
  ...

infixr:100 " ⇒ " => SmallStep
```

Reflexive Transitive Closure

Need to chain steps together!

Use Star (reflexive transitive closure):

- $(S, s) \Rightarrow^* (S', s')$
- Zero or more steps from (S, s) to (S', s')

Big-step via small-step:

$$(S, s) \Rightarrow t \iff (S, s) \Rightarrow^* (\texttt{skip}, t)$$

This connects the two semantics!

Properties of Small-Step Semantics

What can we prove?

1. **Determinism**:
   - Each configuration steps to at most one next configuration
   - Execution is deterministic

2. **Finality**:
   - Only `skip` is final
   - Ensures we have all necessary rules

3. **Equivalence with big-step**:
   - $(S, s) \Rightarrow t \iff (S, s) \Rightarrow^* (\texttt{skip}, t)$

Finality Theorem

```
theorem SmallStep_final (S s) :
    (¬ ∃T t, (S, s) ⇒ (T, t)) ↔ S = Stmt.skip := by
  induction S with
  | skip =>
      simp
      intros T t hstep
      cases hstep
  | assign x a =>
      simp
      exact (_, _, SmallStep.assign ..)
  | seq S T ihS ihT => ...
  ...
```

Proof by induction on statement structure

Big-Step vs Small-Step

Big-step advantages:

- Simpler (one relation)
- Direct connection to result
- Easier for some proofs

Small-step advantages:

- Can express nontermination
- Can reason about intermediate states
- Can model concurrency/interleaving
- More compositional

Use whichever fits your needs!

Section 7

Summary

What We Learned

Proof techniques:

- Backward proofs: tactics working from goal
- Forward proofs: structured constructs
- PAT principle: propositions as types

Inductive predicates:

- Define propositions inductively
- Introduction and elimination rules
- Rule induction on proofs

Operational semantics:

- Formalize programming language meaning
- Big-step: entire execution
- Small-step: one step at a time
- Prove properties of programs and languages

Section 8

Assignments & Next Steps

This Week's Assignments

Readings

- LoVe Demo 3: Backward Proofs
- LoVe Demo 4: Forward Proofs
- LoVe Demo 6: Inductive Predicates
- LoVe Demo 9: Operational Semantics

Assignments

- Quiz 5: Proofs and Semantics (due next week)
- Programming Assignment 5: Implement and verify semantics

Questions & Discussion

Questions?

Join our community:
Discord: https://discord.gg/ZNGE8Xgd
Website: https://danieldia-dev.github.io/proofs/
Email: dmd13@mail.aub.edu

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

— Donald Knuth

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 5 of 10

Proofs & Semantics

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/