AI For Math Seminar, ZJU

Lec 06: Formalization, Verification, and Complexity

Duolei Wang

CUHK-Shenzhen

Learning Goals

- Read Reader, Writer, State, and ExceptT programs from top to bottom.

Learning Goals

- Read Reader, Writer, State, and ExceptT programs from top to bottom.
- Separate monadic program notation from tactic notation such as <;>.

Learning Goals

- Read Reader, Writer, State, and ExceptT programs from top to bottom.
- Separate monadic program notation from tactic notation such as <;>.
- Translate a Scheme expression into runnable definitions and check cases.

Learning Goals

- Read Reader, Writer, State, and ExceptT programs from top to bottom.
- Separate monadic program notation from tactic notation such as <;>.
- Translate a Scheme expression into runnable definitions and check cases.
- Design the Sonoda error-bound proof from its final theorem downward.

Learning Goals

- Read Reader, Writer, State, and ExceptT programs from top to bottom.
- Separate monadic program notation from tactic notation such as <;>.
- Translate a Scheme expression into runnable definitions and check cases.
- Design the Sonoda error-bound proof from its final theorem downward.
- List the semantics and properties needed to verify a Mini-C system.

Learning Goals

- Read Reader, Writer, State, and ExceptT programs from top to bottom.
- Separate monadic program notation from tactic notation such as <;>.
- Translate a Scheme expression into runnable definitions and check cases.
- Design the Sonoda error-bound proof from its final theorem downward.
- List the semantics and properties needed to verify a Mini-C system.
- Read CSLib merge sort as a correctness and cost proof.

Learning Goals

- Read Reader, Writer, State, and ExceptT programs from top to bottom.
- Separate monadic program notation from tactic notation such as <;>.
- Translate a Scheme expression into runnable definitions and check cases.
- Design the Sonoda error-bound proof from its final theorem downward.
- List the semantics and properties needed to verify a Mini-C system.
- Read CSLib merge sort as a correctness and cost proof.
- Design a small language that produces symbolic cost bounds.

Three Blocks of 50 Minutes

0:00–0:50

Monad review and Scheme
APIs, runs, SExpr cases

Short exercises appear beside their examples. Each break gives time to finish one of them.

Three Blocks of 50 Minutes

0:00–0:50

Monad review and Scheme APIs, runs, SExpr cases

1:00–1:50

Limit and cost proofs $\sin x/x$, `TimeM`, merge sort

Short exercises appear beside their examples. Each break gives time to finish one of them.

Three Blocks of 50 Minutes

0:00–0:50

Monad review and Scheme APIs, runs, SExpr cases

1:00–1:50

Limit and cost proofs
$\sin x/x$, `TimeM`, merge sort

2:00–2:50

Finish merge sort; build a project
Big-O and automatic complexity

Short exercises appear beside their examples. Each break gives time to finish one of them.

One Project, Three Questions

1. Which types and rules make the intended example run?

One project may answer all three questions. Each answer needs its own proof and its own list of trusted assumptions.

One Project, Three Questions

1. Which types and rules make the intended example run?
2. What property should be proved about that system or its implementation?

One project may answer all three questions. Each answer needs its own proof and its own list of trusted assumptions.

One Project, Three Questions

1. Which types and rules make the intended example run?
2. What property should be proved about that system or its implementation?
3. What does the chosen cost model count, and what bound follows?

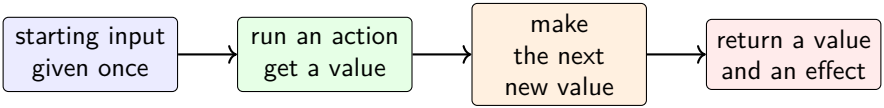
One project may answer all three questions. Each answer needs its own proof and its own list of trusted assumptions.

Section Outline

Subsection Outline

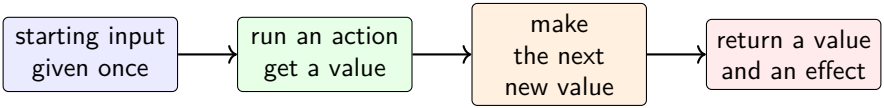
Read a Do Block from Top to Bottom

You can use a Monad before you know how its `bind` is implemented.



Read a Do Block from Top to Bottom

You can use a Monad before you know how its `bind` is implemented.



The Monad instance connects the arrows. The program can still be read as a sequence of small steps.

Four Effects Used in This Lecture

Type	Extra information	Main use	Special API
`ReaderM R A`	read-only input `R`	configuration	`read`, `withReader`
local `Writer A`	list of messages	log a calculation	`tell`, `listen`
`StateM S A`	changing state `S`	counter or store	`get`, `set`, `modify`
`ExceptT E M A`	error or value inside `M`	early exit	`throw`, `tryCatch`

Four Effects Used in This Lecture

Type	Extra information	Main use	Special API
`ReaderM R A`	read-only input `R`	configuration	`read`, `withReader`
local `Writer A`	list of messages	log a calculation	`tell`, `listen`
`StateM S A`	changing state `S`	counter or store	`get`, `set`, `modify`
`ExceptT E M A`	error or value inside `M`	early exit	`throw`, `tryCatch`

`A` is the ordinary returned value in every row.

The Type Shapes

These are the definitions used by Lean's core APIs:

```
def ReaderT (Rho : Type) (M : Type -> Type)
    (Alpha : Type) := Rho -> M Alpha
abbrev ReaderM Rho := ReaderT Rho Id

def StateT (Sigma : Type) (M : Type -> Type)
    (Alpha : Type) := Sigma -> M (Alpha × Sigma)
def StateM Sigma Alpha := StateT Sigma Id Alpha

def ExceptT (Error : Type) (M : Type -> Type)
    (Alpha : Type) := M (Except Error Alpha)
```

Think of a transformer as one more layer around a program: it adds a new effect while the inner monad `M` keeps doing its original job.

Live Lean: `code/D6_01_MonadSupplement.lean`

Do Notation Hides Bind

Read this sequentially

```
do
  let x <- firstAction
  let y := x + 1
  secondAction y
```

Do Notation Hides Bind

Read this sequentially

```
do
  let x <- firstAction
  let y := x + 1
  secondAction y
```

Lean connects it with bind

```
firstAction >>= fun x =>
  let y := x + 1
  secondAction y
```

To use the left program, first ask what `firstAction` returns. The definition of `>>=` can be studied later.

Everyday Do Syntax

Syntax	Read it as
`let x := value`	name a pure value; no monadic action runs
`let x <- action`	run `action`; name its returned value
`action`	run an action and ignore its returned value
`pure value`	return `value` without adding a new effect
`return value`	return from the current `do` block
`if`, `match`	choose the next computation from data

Everyday Do Syntax

Syntax	Read it as
`let x := value`	name a pure value; no monadic action runs
`let x <- action`	run `action`; name its returned value
`action`	run an action and ignore its returned value
`pure value`	return `value` without adding a new effect
`return value`	return from the current `do` block
`if`, `match`	choose the next computation from data

The arrow points from an action to the new name that receives its value.

Similar Symbols Have Different Jobs

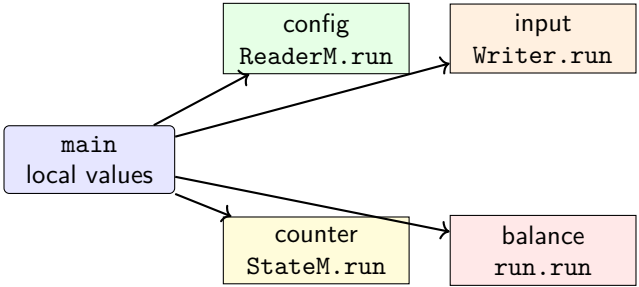
Group	Symbol	Meaning
Monad	`action >>= next`	pass the returned value to `next`
Sequence	`first >> second`	run both; ignore the first result
Functor	`f <$> action`	apply pure `f` to the returned value
Applicative	`mf <*> mx`	apply a wrapped function to a wrapped value
Applicative	`left <* right`	run both; keep the left result
Applicative	`left *> right`	run both; keep the right result
Choice	`left <|> right`	try the right side when the left side fails
Tactic	`tacticOne <;> tacticTwo`	run `tacticTwo` on every new goal

Similar Symbols Have Different Jobs

Group	Symbol	Meaning
Monad	`action >>= next`	pass the returned value to `next`
Sequence	`first >> second`	run both; ignore the first result
Functor	`f <$> action`	apply pure `f` to the returned value
Applicative	`mf <*> mx`	apply a wrapped function to a wrapped value
Applicative	`left <* right`	run both; keep the left result
Applicative	`left *> right`	run both; keep the right result
Choice	`left <|> right`	try the right side when the left side fails
Tactic	`tacticOne <;> tacticTwo`	run `tacticTwo` on every new goal

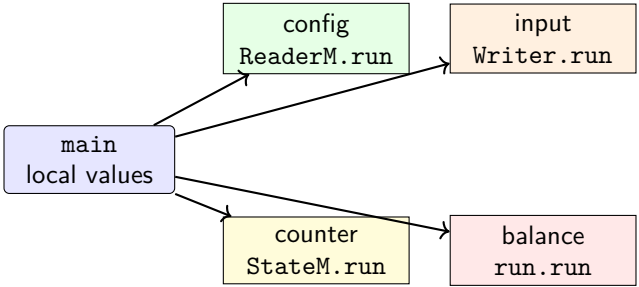
`<;>` is proof syntax. It is not an operator of `Monad`.

One Main Supplies Each Starting Value Once



These are not mutable global variables. `main` creates local programs and passes each initial value at one visible call site.

One Main Supplies Each Starting Value Once



These are not mutable global variables. `main` creates local programs and passes each initial value at one visible call site. `reprStr value` makes a printable string from a value with a `Repr` instance. `IO.println` is the final output action.

ReaderM API Definitions

For `ReaderM`, the environment is simply a function input.

```
def read {R : Type} : ReaderM R R :=
  fun env => env

def withReader {R R2 A : Type}
    (change : R -> R2) (action : ReaderM R2 A) :
    ReaderM R A :=
  fun env => action (change env)

def ReaderM.run {R A : Type}
    (action : ReaderM R A) (env : R) : A :=
  action env
```

Read `withReader change action` as: "run this one action using a temporary view made by `change`." It does not mutate the environment.

ReaderM: Read a Configuration

```
structure AppConfig where
  course : String
  bonus : Nat

def main : IO Unit := do
  let config : AppConfig := { course := "Lean", bonus := 3 }
  let program : ReaderM AppConfig String := do
    let original <- read
    let local <- withReader
      (fun cfg => { cfg with course := cfg.course ++ " workshop" }) do
        let changed <- read
        pure changed.course
    let restored <- read
    pure s!"{original.course}; {local}; {restored.course}"
  IO.println (program.run config)
```

Live Lean: code/D6_01_MonadSupplement.lean

ReaderM Trace and Exercise

`Lean` ⟶ `Lean workshop` ⟶ `Lean`

Think: "run this small part using a temporarily modified configuration." Only the nested action sees `Lean workshop`; the last `read` sees `Lean` again. There is no restore command because the original configuration was never mutated. **Try now:**

```
let readerProgram : ReaderM AppConfig Nat := do
  let current <- read
  pure (sorry)  -- return current.bonus
```

Live Lean: `code/D6_01_MonadSupplement_Exercises.lean`

A Small Local Writer

Lean core has no separate `WriterT`, so this lecture defines one small type.

```
structure Writer (Alpha : Type) where
  value : Alpha
  log : List String

instance : Monad Writer where
  pure value := { value, log := [] }
  bind action next :=
    let result := next action.value
    { value := result.value,
      log := action.log ++ result.log }
```

Live Lean: `code/D6_01_MonadSupplement.lean`

Writer API Definitions

```
def tell (message : String) : Writer PUnit :=
  { value := PUnit.unit, log := [message] }

def listen {A : Type} (action : Writer A) :
    Writer (A × List String) :=
  { value := (action.value, action.log),
    log := action.log }

def Writer.run {A : Type} (action : Writer A) :
    A × List String :=
  (action.value, action.log)
```

Think of a Writer result as an answer with a notebook beside it. `tell` adds one line; `listen` makes one nested action's lines available as a value without removing them from the notebook.

Writer: Calculate and Record Messages

```
let input : Nat := 4
let writerProgram : Writer Nat := do
  tell s!"input = {input}"
  let doubled := input * 2
  let observed <- listen (do
    tell s!"doubled = {doubled}"
    pure doubled)
  tell s!"listen saw {observed.2.length} message"
  pure observed.1

let writerResult := writerProgram.run
```

`tell` writes a note beside the calculation. `listen` lets the program inspect the notes made by one nested calculation. The answer remains 8; listening does not consume the notes.

Live Lean: `code/D6_01_MonadSupplement.lean`

Writer Trace and Exercise

```
(8, ["input = 4", "doubled = 8", "listen saw 1 message"])
```

The log grows from left to right. Earlier values are not changed. **Try now:**

```
let writerProgram : Writer Nat := do
  tell "start"
  let doubled := input * 2
  tell (sorry)  -- record the doubled value
  pure doubled
```

Live Lean: code/D6_01_MonadSupplement_Exercises.lean

StateM API Definitions

A state action is a function from an old state to a value and a new state.

```
def get {S : Type} : StateM S S :=
  fun state => (state, state)

def set {S : Type} (newState : S) : StateM S PUnit :=
  fun _oldState => (PUnit.unit, newState)

def modify {S : Type} (change : S -> S) : StateM S PUnit :=
  fun state => (PUnit.unit, change state)

def StateM.run {S A : Type}
    (action : StateM S A) (initial : S) : A × S :=
  action initial
```

Think of the state as an extra value handed from one line to the next: `get` looks at it, `set` chooses its replacement, and `modify` computes a replacement. No global variable is mutated.

StateM: Pass a New State Forward

```
def main : IO Unit := do
  let initialCounter : Nat := 10
  let stateProgram : StateM Nat (Nat × Nat) := do
    let before <- get
    modify (fun counter => counter + 5)
    let after <- get
    pure (before, after)
  let result := stateProgram.run initialCounter
  IO.println (reprStr result)
```

- The first get takes a snapshot of the carried counter: 10.

Live Lean: code/D6_01_MonadSupplement.lean

StateM: Pass a New State Forward

```
def main : IO Unit := do
  let initialCounter : Nat := 10
  let stateProgram : StateM Nat (Nat × Nat) := do
    let before <- get
    modify (fun counter => counter + 5)
    let after <- get
    pure (before, after)
  let result := stateProgram.run initialCounter
  IO.println (reprStr result)
```

- The first `get` takes a snapshot of the carried counter: 10.
- `modify` passes 15 to later lines; it does not change a global box.

Live Lean: `code/D6_01_MonadSupplement.lean`

StateM: Pass a New State Forward

```
def main : IO Unit := do
  let initialCounter : Nat := 10
  let stateProgram : StateM Nat (Nat × Nat) := do
    let before <- get
    modify (fun counter => counter + 5)
    let after <- get
    pure (before, after)
  let result := stateProgram.run initialCounter
  IO.println (reprStr result)
```

- The first `get` takes a snapshot of the carried counter: `10`.
- `modify` passes `15` to later lines; it does not change a global box.
- `run 10` supplies the start once and returns `((10, 15), 15)`: program value first, final state second.

Live Lean: `code/D6_01_MonadSupplement.lean`

StateM Exercise

The outside caller gives 10 once. Fill only the update function.

```
let initialCounter : Nat := 10
let stateProgram : StateM Nat Nat := do
  modify (fun counter => sorry)
  get

-- wanted result: (15, 15)
stateProgram.run initialCounter
```

Live Lean: code/D6_01_MonadSupplement_Exercises.lean

StateM Exercise

The outside caller gives 10 once. Fill only the update function.

```
let initialCounter : Nat := 10
let stateProgram : StateM Nat Nat := do
  modify (fun counter => sorry)
  get

-- wanted result: (15, 15)
stateProgram.run initialCounter
```

Use `counter + 5`. The first 15 is the returned value; the second is the final state.

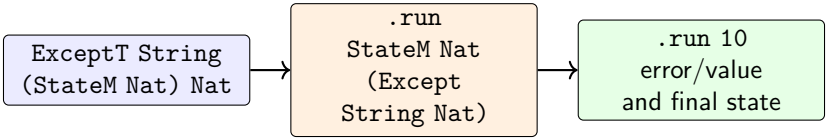
Live Lean: `code/D6_01_MonadSupplement_Exercises.lean`

ExceptT API Definitions

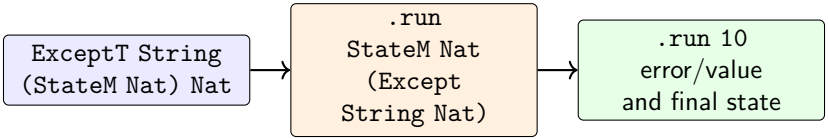
```
def ExceptT.run {E A : Type} {M : Type -> Type}
    (action : ExceptT E M A) : M (Except E A) :=
  action

-- Operational behavior inside the same base monad M:
throw error       = pure (.error error)
tryCatch act h    = do
  match <- act.run with
  | .ok value => pure value
  | .error error => h error
```

Think of `throw` as taking the error exit: later lines on that path are skipped. `tryCatch` provides a local recovery route and runs its handler only when that exit is taken.

ExceptT over StateM



ExceptT over StateM



The two `.run` calls do different jobs. The first peels off the error layer; the second supplies the initial state. Read a transformer stack one layer at a time.

ExceptT: Withdraw or Stop

```
let withdraw : Nat -> ExceptT String (StateM Nat) Nat :=
  fun amount => do
    let balance <- get
    if amount <= balance then
      modify (fun current => current - amount)
      get
    else
      throw s!"need {amount}, but have {balance}"

(withdraw 4).run.run 10
-- (Except.ok 6, 6)
(withdraw 20).run.run 10
-- (Except.error "need 20, but have 10", 10)
```

Success passes the reduced balance forward. Failure takes the error exit before `modify`, so the final balance is still 10.

Live Lean: `code/D6_01_MonadSupplement.lean`

Catch an Error in the Same Stack

```
let recovered : ExceptT String (StateM Nat) Nat :=
  tryCatch (withdraw 20) fun _message => get

recovered.run.run 10
-- (Except.ok 10, 10)
```

- `tryCatch action handler` runs the handler only after an error.

Live Lean: `code/D6_01_MonadSupplement.lean`

Catch an Error in the Same Stack

```
let recovered : ExceptT String (StateM Nat) Nat :=
  tryCatch (withdraw 20) fun _message => get

recovered.run.run 10
-- (Except.ok 10, 10)
```

- `tryCatch action handler` runs the handler only after an error.
- The handler can still use `get` because the base monad is `StateM`.

Live Lean: `code/D6_01_MonadSupplement.lean`

Catch an Error in the Same Stack

```
let recovered : ExceptT String (StateM Nat) Nat :=
  tryCatch (withdraw 20) fun _message => get

recovered.run.run 10
-- (Except.ok 10, 10)
```

- `tryCatch action handler` runs the handler only after an error.
- The handler can still use `get` because the base monad is `StateM`.
- Transformer order matters: this stack does not promise automatic rollback.

Live Lean: `code/D6_01_MonadSupplement.lean`

ExceptT Exercise

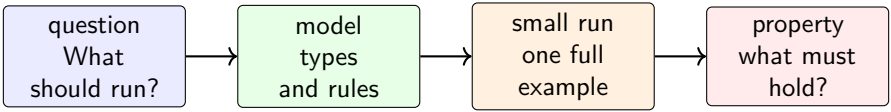
Fill the subtraction and the error message. Then run both calls.

```
let withdraw : Nat -> ExceptT String (StateM Nat) Nat :=
  fun amount => do
    let balance <- get
    if amount <= balance then
      modify (fun current => sorry)
      get
    else
      throw (sorry)

(withdraw 4).run.run 10
(withdraw 20).run.run 10
```

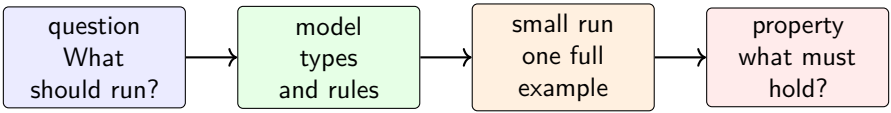
Live Lean: `code/D6_01_MonadSupplement_Exercises.lean`

Subsection Outline

Begin with a Question, Not a Data Type



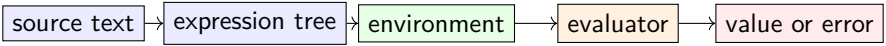
Begin with a Question, Not a Data Type



Formalization builds the model. Verification proves a property of it.

Scheme: Translate the Question Directly

Question: What does a Scheme expression mean?



Each arrow becomes a definition or function. We do not begin with macros, closures, or a full standard.

A Repository Can Follow the Same Plan

`Duolei-Wang/Scheme-Interpreter` separates the jobs:

File	Question answered
`Syntax.hs`	Which expressions and values exist?
`Parser.hs`	How does text become syntax?
`Environment.hs`	What does a name mean?
`Evaluator.hs`	How does an expression run?
`Macro.hs`	How is macro syntax expanded?

Formal Definition: A Small Scheme Syntax

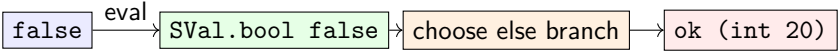
```
inductive SExpr where
  | int (value : Int)
  | bool (value : Bool)
  | symbol (name : String)
  | add (left right : SExpr)
  | ifThenElse
      (condition thenBranch elseBranch : SExpr)

inductive SVal where
  | int (value : Int)
  | bool (value : Bool)
```

The constructors are the grammar of the supported language.

Live Lean: `code/D6_02_FormalizationVerification.lean`

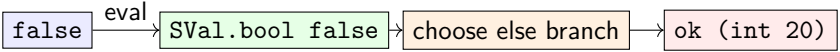
One Run Before More Definitions

Evaluate `if false then 10 else 20`.



One Run Before More Definitions

Evaluate `if false then 10 else 20`.



This run tells us the exact rule that `eval` must implement.

Formal Definition: Evaluation

```
abbrev Env := String -> Option SVal

def eval (env : Env) : SExpr -> Except String SVal
  | .int n => pure (.int n)
  | .symbol name =>
      match env name with
      | some value => pure value
      | none => throw s!"unbound symbol: {name}"
  | .ifThenElse condition yes no => do
      match <- eval env condition with
      | .bool true => eval env yes
      | .bool false => eval env no
      | _ => throw "if expects a Boolean"
  | ...
```

Live Lean: code/D6_02_FormalizationVerification.lean

Exercise: Check the Direct Translation

First predict the result. Then replace `sorry`.

```
theorem eval_if_false (x y : Int) :
    eval emptyEnv
      (.ifThenElse (.bool false) (.int x) (.int y)) =
      .ok (.int y) := by
  sorry
```

Live Lean: `code/D6_02_FormalizationVerification_Exercises.lean`

Exercise: Check the Direct Translation

First predict the result. Then replace `sorry`.

```
theorem eval_if_false (x y : Int) :
    eval emptyEnv
      (.ifThenElse (.bool false) (.int x) (.int y)) =
      .ok (.int y) := by
  sorry
```

The solution is `rfl`: the formal rule computes exactly as the hand trace predicted.

Live Lean: `code/D6_02_FormalizationVerification_Exercises.lean`

First Break: Monad and SExpr Practice

Work locally for ten minutes. Both files are complete by themselves.

1. Open `D6_01_MonadSupplement_Exercises.lean` and fill only the easy Reader, Writer, State, and ExceptT holes.

Start with definitions and concrete runs; the solutions are separate files.

First Break: Monad and SExpr Practice

Work locally for ten minutes. Both files are complete by themselves.

1. Open `D6_01_MonadSupplement_Exercises.lean` and fill only the easy Reader, Writer, State, and ExceptT holes.
2. Open `D6_02_FormalizationVerification_Exercises.lean`.

Start with definitions and concrete runs; the solutions are separate files.

First Break: Monad and SExpr Practice

Work locally for ten minutes. Both files are complete by themselves.

1. Open `D6_01_MonadSupplement_Exercises.lean` and fill only the easy Reader, Writer, State, and ExceptT holes.
2. Open `D6_02_FormalizationVerification_Exercises.lean`.
3. Predict the false-branch result before proving `eval_if_false`.

Start with definitions and concrete runs; the solutions are separate files.

First Break: Monad and SExpr Practice

Work locally for ten minutes. Both files are complete by themselves.

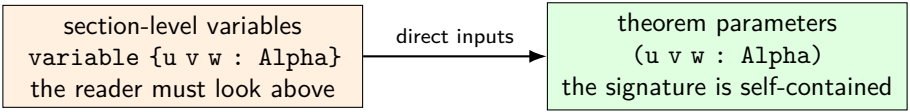
1. Open `D6_01_MonadSupplement_Exercises.lean` and fill only the easy Reader, Writer, State, and ExceptT holes.
2. Open `D6_02_FormalizationVerification_Exercises.lean`.
3. Predict the false-branch result before proving `eval_if_false`.
4. If time remains, prove the literal-addition and unbound-symbol cases.

Start with definitions and concrete runs; the solutions are separate files.

Subsection Outline

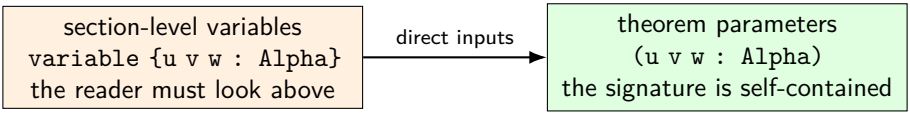
Put the Inputs in the Theorem

Suppose the objects u, v, w are used by only one theorem.



Put the Inputs in the Theorem

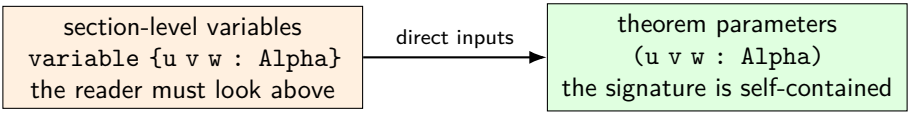
Suppose the objects u, v, w are used by only one theorem.



This is a writing rule, not a new logical rule. Both styles can describe the same proposition, but the second style is easier to review.

Put the Inputs in the Theorem

Suppose the objects u, v, w are used by only one theorem.



This is a writing rule, not a new logical rule. Both styles can describe the same proposition, but the second style is easier to review. Do not confuse these objects with `universe u`; a universe name tells Lean the level of a type.

A Full-Parameter Theorem

```
theorem chain_full
    (Alpha : Type) (R : Alpha -> Alpha -> Prop)
    (u v w : Alpha)
    (huv : R u v) (hvw : R v w)
    (htrans : forall {a b c : Alpha},
      R a b -> R b c -> R a c) :
    R u w := by
  exact htrans huv hvw
```

Every object and every assumption can be checked from this one signature. This style is useful for a paper's main definitions and public theorems.

Local file: `lean-code/D6_03_LHopital_Exercises.lean`

Use @ to Show Hidden Inputs

Lean normally infers type parameters from later arguments.

```
#eval List.map Nat.succ [1, 2, 3]

-- @ makes the hidden input and output types explicit.
#eval @List.map Nat Nat Nat.succ [1, 2, 3]
```

- The first `Nat` is the input element type.

Use @ to Show Hidden Inputs

Lean normally infers type parameters from later arguments.

```
#eval List.map Nat.succ [1, 2, 3]

-- @ makes the hidden input and output types explicit.
#eval @List.map Nat Nat Nat.succ [1, 2, 3]
```

- The first `Nat` is the input element type.
- The second `Nat` is the output element type.

Use @ to Show Hidden Inputs

Lean normally infers type parameters from later arguments.

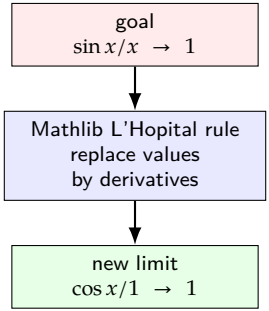
```
#eval List.map Nat.succ [1, 2, 3]

-- @ makes the hidden input and output types explicit.
#eval @List.map Nat Nat Nat.succ [1, 2, 3]
```

- The first `Nat` is the input element type.
- The second `Nat` is the output element type.
- Use `@name` to inspect or debug a call; ordinary code may keep inference.

A Limit Theorem as a Top-Down Plan

The target is the familiar limit

$$\lim_{x \to 0,\ x \neq 0} \frac{\sin x}{x} = 1.$$

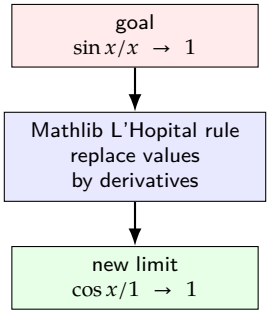


A Limit Theorem as a Top-Down Plan

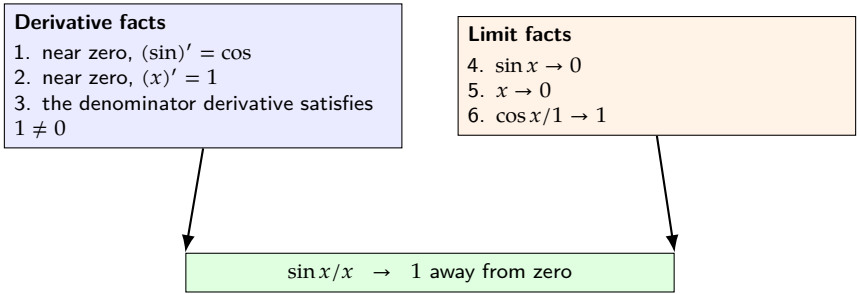
The target is the familiar limit

$$\lim_{x \to 0,\ x \neq 0} \frac{\sin x}{x} = 1.$$

It is a punctured limit: the expression is not defined at $x = 0$.



What L'Hopital Asks Us to Prove



The Final Explicit Call

```
open Filter Set
open scoped Topology

theorem tendsto_sin_div_id :
    Tendsto (fun x : Real => Real.sin x / x)
      (nhdsWithin 0 (Set.compl {0})) (nhds 1) := by
  exact @HasDerivAt.lhopital_zero_nhds
    0 (nhds 1)
    Real.sin Real.cos
    (fun x : Real => x) (fun _x : Real => 1)
    sin_derivative_near_zero
    id_derivative_near_zero
    id_derivative_nonzero_near_zero
    sin_tends_to_zero id_tends_to_zero
    derivative_ratio_tends_to_one
```

The six proof arguments occur in the same order as the six facts.

Local file: `lean-code/D6_03_LHopital_Exercises.lean`

Exercise: Complete the Six L'Hopital Facts

Open `lean-code/D6_03_LHopital_Exercises.lean` locally.

1. Read the final theorem and write down its six needed facts.

This is the first task after the break. The instructor copy is `D6_03_LHopital_Solutions.lean`.

Exercise: Complete the Six L'Hopital Facts

Open `lean-code/D6_03_LHopital_Exercises.lean` locally.

1. Read the final theorem and write down its six needed facts.
2. Fill the two derivative holes with `Real.hasDerivAt_sin` and `hasDerivAt_id'`.

This is the first task after the break. The instructor copy is `D6_03_LHopital_Solutions.lean`.

Exercise: Complete the Six L'Hopital Facts

Open `lean-code/D6_03_LHopital_Exercises.lean` locally.

1. Read the final theorem and write down its six needed facts.
2. Fill the two derivative holes with `Real.hasDerivAt_sin` and `hasDerivAt_id'`.
3. Prove $1 \neq 0$, then use continuity for the three limit holes.

This is the first task after the break. The instructor copy is `D6_03_LHopital_Solutions.lean`.

Exercise: Complete the Six L'Hopital Facts

Open `lean-code/D6_03_LHopital_Exercises.lean` locally.

1. Read the final theorem and write down its six needed facts.
2. Fill the two derivative holes with `Real.hasDerivAt_sin` and `hasDerivAt_id'`.
3. Prove $1 \neq 0$, then use continuity for the three limit holes.
4. Compile the file. Do not edit the final `@...` call first.

This is the first task after the break. The instructor copy is `D6_03_LHopital_Solutions.lean`.

Section Outline

Subsection Outline

What CSLib Tries to Provide

CSLib is the Lean Computer Science Library.

- Common definitions for computer-science formalization.

A library is useful when later proofs can build on small, named lemmas.

What CSLib Tries to Provide

CSLib is the Lean Computer Science Library.

- Common definitions for computer-science formalization.
- APIs that different projects can reuse.

A library is useful when later proofs can build on small, named lemmas.

What CSLib Tries to Provide

CSLib is the Lean Computer Science Library.

- Common definitions for computer-science formalization.
- APIs that different projects can reuse.
- Executable programs together with proof interfaces.

A library is useful when later proofs can build on small, named lemmas.

What CSLib Tries to Provide

CSLib is the Lean Computer Science Library.

- Common definitions for computer-science formalization.
- APIs that different projects can reuse.
- Executable programs together with proof interfaces.
- Clear separation between program results and resource costs.

A library is useful when later proofs can build on small, named lemmas.

A Cost Theorem Needs a Cost Model

Before adding any Lean tick, write four answers.

1. What is the input size?

For CSLib merge sort, input size is list length and one comparison costs one unit. This is a mathematical model, not a wall-clock measurement.

A Cost Theorem Needs a Cost Model

Before adding any Lean tick, write four answers.

1. What is the input size?
2. Which operation costs one unit?

For CSLib merge sort, input size is list length and one comparison costs one unit. This is a mathematical model, not a wall-clock measurement.

A Cost Theorem Needs a Cost Model

Before adding any Lean tick, write four answers.

1. What is the input size?
2. Which operation costs one unit?
3. Which operations are free in this model?

For CSLib merge sort, input size is list length and one comparison costs one unit. This is a mathematical model, not a wall-clock measurement.

A Cost Theorem Needs a Cost Model

Before adding any Lean tick, write four answers.

1. What is the input size?
2. Which operation costs one unit?
3. Which operations are free in this model?
4. Do we want an exact cost, an upper bound, or Big-O?

For CSLib merge sort, input size is list length and one comparison costs one unit. This is a mathematical model, not a wall-clock measurement.

Three Different Claims

For one sorting program, we may prove:

- **Correctness:** the output is sorted and is a permutation of input.

These claims need different definitions. None follows only because the code runs.

Three Different Claims

For one sorting program, we may prove:

- **Correctness:** the output is sorted and is a permutation of input.
- **Exact or pointwise cost:** every input list satisfies a concrete natural-number inequality.

These claims need different definitions. None follows only because the code runs.

Three Different Claims

For one sorting program, we may prove:

- **Correctness:** the output is sorted and is a permutation of input.
- **Exact or pointwise cost:** every input list satisfies a concrete natural-number inequality.
- **Asymptotic cost:** the worst-case cost is $O(n \log n)$.

These claims need different definitions. None follows only because the code runs.

The Library Design Method

A reusable complexity development usually has four layers:

1. an instrumented program with an explicit cost model;

CSLib's merge-sort file follows this order.

The Library Design Method

A reusable complexity development usually has four layers:

1. an instrumented program with an explicit cost model;
2. return-value lemmas for functional correctness;

CSLib's merge-sort file follows this order.

The Library Design Method

A reusable complexity development usually has four layers:

1. an instrumented program with an explicit cost model;
2. return-value lemmas for functional correctness;
3. time lemmas that connect code to a recurrence;

CSLib's merge-sort file follows this order.

The Library Design Method

A reusable complexity development usually has four layers:

1. an instrumented program with an explicit cost model;
2. return-value lemmas for functional correctness;
3. time lemmas that connect code to a recurrence;
4. mathematics that solves the recurrence or proves Big-O.

CSLib's merge-sort file follows this order.

Subsection Outline

A Normal Function Forgets Cost

A normal function has a type such as

```
List Nat -> List Nat.
```

The output contains the result list, but no model cost.

- We could write a second cost function by hand.

A Normal Function Forgets Cost

A normal function has a type such as

```
List Nat -> List Nat.
```

The output contains the result list, but no model cost.

- We could write a second cost function by hand.
- We could return a pair of result and cost.

A Normal Function Forgets Cost

A normal function has a type such as

```
List Nat -> List Nat.
```

The output contains the result list, but no model cost.

- We could write a second cost function by hand.
- We could return a pair of result and cost.
- `TimeM` packages that pair and gives it monad operations.

The Data Stored by TimeM

```
structure TimeM (T : Type) (Alpha : Type) where
  ret : Alpha
  time : T

def tick (cost : T) : TimeM T PUnit :=
  { ret := PUnit.unit, time := cost }
```

- `ret` is the normal program result.

Live Lean: `code/D6_04_ComplexityProof.lean`

The Data Stored by TimeM

```
structure TimeM (T : Type) (Alpha : Type) where
  ret : Alpha
  time : T

def tick (cost : T) : TimeM T PUnit :=
  { ret := PUnit.unit, time := cost }
```

- `ret` is the normal program result.
- `time` is the chosen model cost.

Live Lean: `code/D6_04_ComplexityProof.lean`

The Data Stored by TimeM

```
structure TimeM (T : Type) (Alpha : Type) where
  ret : Alpha
  time : T

def tick (cost : T) : TimeM T PUnit :=
  { ret := PUnit.unit, time := cost }
```

- `ret` is the normal program result.
- `time` is the chosen model cost.
- In this lecture `T` is usually `Nat`.

Live Lean: `code/D6_04_ComplexityProof.lean`

Three TimeM Rules

Operation	Returned value	Cost
`pure x`	`x`	0
`tick c`	unit	c
`m >>= f`	result of `f m.ret`	`m.time + (f m.ret).time`

The bind rule is why do notation can add the costs of sequential steps.

The Proof API Exposes Those Three Rules

CSLib gives simplification lemmas with the same meaning:

```
ret_bind  : (action >>= next).ret = (next action.ret).ret
time_bind : (action >>= next).time =
  action.time + (next action.ret).time
time_tick : (tick cost).time = cost
```

- `simp` unfolds a small computation with these equations.

The Proof API Exposes Those Three Rules

CSLib gives simplification lemmas with the same meaning:

```
ret_bind  : (action >>= next).ret = (next action.ret).ret
time_bind : (action >>= next).time =
  action.time + (next action.ret).time
time_tick : (tick cost).time = cost
```

- `simp` unfolds a small computation with these equations.
- `grw [lemma]` rewrites with a proved equality and keeps arithmetic side conditions for `grind`.

The Proof API Exposes Those Three Rules

CSLib gives simplification lemmas with the same meaning:

```
ret_bind  : (action >>= next).ret = (next action.ret).ret
time_bind : (action >>= next).time =
  action.time + (next action.ret).time
time_tick : (tick cost).time = cost
```

- `simp` unfolds a small computation with these equations.
- `grw [lemma]` rewrites with a proved equality and keeps arithmetic side conditions for `grind`.
- These lemmas expose the cost already written in the program.

One Timed Computation by Hand

Let `constantWork 2 10` return 10 and charge 2.

1. Run `tick 2`: returned value is unit, cost is 2.

Therefore the final object is

$$\{\texttt{ret} := 10,\ \texttt{time} := 2\}.$$

One Timed Computation by Hand

Let `constantWork 2 10` return 10 and charge 2.

1. Run `tick 2`: returned value is unit, cost is 2.
2. Continue with `pure 10`: returned value is 10, cost is 0.

Therefore the final object is

$$\{\texttt{ret} := 10,\ \texttt{time} := 2\}.$$

One Timed Computation by Hand

Let `constantWork 2 10` return 10 and charge 2.

1. Run `tick 2`: returned value is unit, cost is 2.
2. Continue with `pure 10`: returned value is 10, cost is 0.
3. Bind adds the costs: $2 + 0 = 2$.

Therefore the final object is

$$\{\texttt{ret} := 10,\ \texttt{time} := 2\}.$$

TimeM Has a Trust Boundary

CSLib's TimeM documentation says that time annotations are trusted.

- Lean checks arithmetic about the ticks we inserted.

TimeM Has a Trust Boundary

CSLib's TimeM documentation says that time annotations are trusted.

- Lean checks arithmetic about the ticks we inserted.
- Lean does not prove that a tick matches a real CPU instruction.

TimeM Has a Trust Boundary

CSLib's TimeM documentation says that time annotations are trusted.

- Lean checks arithmetic about the ticks we inserted.
- Lean does not prove that a tick matches a real CPU instruction.
- A missing tick can make a false cost model look cheap.

TimeM Has a Trust Boundary

CSLib's TimeM documentation says that time annotations are trusted.

- Lean checks arithmetic about the ticks we inserted.
- Lean does not prove that a tick matches a real CPU instruction.
- A missing tick can make a false cost model look cheap.
- The cost convention must be written and reviewed with the code.

Worked Example: Timed Map

Run a function of cost 2 on `[10, 20, 30]`.

Item	Result	Item cost	Running cost
10	10	2	2
20	20	2	4
30	30	2	6

The final result is the same list, and the final cost is $3 \cdot 2 = 6$.

Timed Map in Lean

```
def constantWork (cost : Nat) (value : Alpha) :
    TimeM Nat Alpha := do
  let _ <- tick cost
  pure value

def timedMap (function : Alpha -> TimeM Nat Beta) :
    List Alpha -> TimeM Nat (List Beta)
  | [] => { ret := [], time := 0 }
  | head :: tail =>
      let first := function head
      let rest := timedMap function tail
      { ret := first.ret :: rest.ret,
        time := first.time + rest.time }
```

Live Lean: code/D6_04_ComplexityProof.lean

Two Theorems About the Same Program

Return-value theorem:

```
(timedMap (constantWork c) xs).ret = xs.
```

Time theorem:

```
(timedMap f xs).time = xs.length * c,
```

assuming every `f x` has time c.

- The first theorem ignores the time field.

Two Theorems About the Same Program

Return-value theorem:

```
(timedMap (constantWork c) xs).ret = xs.
```

Time theorem:

```
(timedMap f xs).time = xs.length * c,
```

assuming every `f x` has time c.

- The first theorem ignores the time field.
- The second theorem ignores most value details.

Two Theorems About the Same Program

Return-value theorem:

```
(timedMap (constantWork c) xs).ret = xs.
```

Time theorem:

```
(timedMap f xs).time = xs.length * c,
```

assuming every `f x` has time c.

- The first theorem ignores the time field.
- The second theorem ignores most value details.
- This separation is a main design principle of CSLib's TimeM.

Why List Induction Proves the Time Formula

Empty list:

$$0 = 0 \cdot c.$$

Why List Induction Proves the Time Formula

Empty list:

$$0 = 0 \cdot c.$$

Head and tail:

Why List Induction Proves the Time Formula

Empty list:

$$0 = 0 \cdot c.$$

Head and tail:

$$\begin{aligned} \mathsf{cost}(x :: xs) &= c + \mathsf{cost}(xs) \\ &= c + |xs|c \\ &= (|xs| + 1)c. \end{aligned}$$

Why List Induction Proves the Time Formula

Empty list:

$$0 = 0 \cdot c.$$

Head and tail:

$$\begin{aligned} \mathsf{cost}(x :: xs) &= c + \mathsf{cost}(xs) \\ &= c + |xs|c \\ &= (|xs| + 1)c. \end{aligned}$$

The second equality is the induction hypothesis.

The Lean Proof Follows the Calculation

```
theorem timedMap_time
    (function : Alpha -> TimeM Nat Beta)
    (cost : Nat)
    (constantCost : forall value,
      (function value).time = cost)
    (input : List Alpha) :
    (timedMap function input).time =
      input.length * cost := by
  induction input with
  | nil => simp [timedMap]
  | cons head tail ih =>
      simp [timedMap, constantCost head, ih,
        Nat.add_mul, Nat.add_comm]
```

Live Lean: code/D6_04_ComplexityProof.lean

Try Now: Read the Two Fields

```
theorem tick_time (cost : Nat) :
    (tick cost : TimeM Nat PUnit).time = cost := by
  sorry

theorem constantWork_ret (cost value : Nat) :
    (constantWork cost value).ret = value := by
  sorry
```

Live Lean: code/D6_04_ComplexityProof_Exercises.lean

Try Now: Read the Two Fields

```
theorem tick_time (cost : Nat) :
    (tick cost : TimeM Nat PUnit).time = cost := by
  sorry

theorem constantWork_ret (cost value : Nat) :
    (constantWork cost value).ret = value := by
  sorry
```

Both proofs only unfold data that was just constructed.

Live Lean: `code/D6_04_ComplexityProof_Exercises.lean`

Check: Read the Two Fields

```
theorem tick_time (cost : Nat) :
    (tick cost : TimeM Nat PUnit).time = cost := by
  rfl

theorem constantWork_ret (cost value : Nat) :
    (constantWork cost value).ret = value := by
  rfl
```

These easy proofs are useful: they teach which field a theorem is about.

Subsection Outline

First Understand Plain Merge

Merge the sorted lists `[1, 4]` and `[2, 3]`.

Heads	Choose	Output so far	Comparisons
1 and 2	1	[1]	1
4 and 2	2	[1,2]	2
4 and 3	3	[1,2,3]	3
right empty	append 4	[1,2,3,4]	3

Only comparison of two nonempty heads costs one unit.

The Tick Is Next to the Comparison

CSLib's merge has this logical shape:

```
def merge : List Alpha -> List Alpha ->
    TimeM Nat (List Alpha)
  | [], right => pure right
  | left, [] => pure left
  | x :: xs, y :: ys => do
      tick 1
      if x <= y then
        pure (x :: (merge xs (y :: ys)).ret)
      else
        pure (y :: (merge (x :: xs) ys).ret)
```

The local file uses the same cost convention without importing CSLib.

Live Lean: `code/D6_04_ComplexityProof.lean`

Why Merge Has a Linear Bound

When both lists are nonempty:

- one comparison is charged;

Therefore induction proves

$$\texttt{(merge xs ys).time} \leq \texttt{xs.length + ys.length}.$$

The bound is slightly loose because the last nonempty tail is appended for free.

Why Merge Has a Linear Bound

When both lists are nonempty:

- one comparison is charged;
- one head is removed;

Therefore induction proves

$$\texttt{(merge xs ys).time} \leq \texttt{xs.length + ys.length}.$$

The bound is slightly loose because the last nonempty tail is appended for free.

Why Merge Has a Linear Bound

When both lists are nonempty:

- one comparison is charged;
- one head is removed;
- total remaining length drops by one.

Therefore induction proves

$$\texttt{(merge xs ys).time} \leq \texttt{xs.length + ys.length}.$$

The bound is slightly loose because the last nonempty tail is appended for free.

Try Now: Count Before Proving

Without running Lean, predict both values:

```
#eval (merge [1, 4] [2, 3]).ret
#eval (merge [1, 4] [2, 3]).time
```

Live Lean: `code/D6_04_ComplexityProof.lean`

Try Now: Count Before Proving

Without running Lean, predict both values:

```
#eval (merge [1, 4] [2, 3]).ret
#eval (merge [1, 4] [2, 3]).time
```

Result: `[1, 2, 3, 4]`.

Live Lean: `code/D6_04_ComplexityProof.lean`

Try Now: Count Before Proving

Without running Lean, predict both values:

```
#eval (merge [1, 4] [2, 3]).ret
#eval (merge [1, 4] [2, 3]).time
```

Result: `[1, 2, 3, 4]`. Time: 3.

Live Lean: `code/D6_04_ComplexityProof.lean`

Try Now: Count Before Proving

Without running Lean, predict both values:

```
#eval (merge [1, 4] [2, 3]).ret
#eval (merge [1, 4] [2, 3]).time
```

Result: `[1, 2, 3, 4]`. Time: 3. Now compare 3 with the general bound $2 + 2 = 4$.

Live Lean: `code/D6_04_ComplexityProof.lean`

Correctness Has Two Parts

CSLib proves

```
mergeSort_correct (xs) :  IsSorted output ∧ output ~ xs.
```

- Sortedness alone is not enough: returning `[]` is sorted.

Correctness Has Two Parts

CSLib proves

```
mergeSort_correct (xs) :  IsSorted output ∧ output ~ xs.
```

- Sortedness alone is not enough: returning `[]` is sorted.
- Permutation alone is not enough: the input itself is a permutation.

Correctness Has Two Parts

CSLib proves

```
mergeSort_correct (xs) :  IsSorted output ∧ output ~ xs.
```

- Sortedness alone is not enough: returning `[]` is sorted.
- Permutation alone is not enough: the input itself is a permutation.
- Together, they state the usual functional correctness of sorting.

The Return-Value Proof Chain

1. `ret_merge`: timed merge returns ordinary merge.

Every theorem in this chain is about `.ret`, not `.time`.

The Return-Value Proof Chain

1. `ret_merge`: timed merge returns ordinary merge.
2. `sorted_merge`: merging sorted lists is sorted.

Every theorem in this chain is about `.ret`, not `.time`.

The Return-Value Proof Chain

1. `ret_merge`: timed merge returns ordinary merge.
2. `sorted_merge`: merging sorted lists is sorted.
3. `mergeSort_sorted`: recursive output is sorted.

Every theorem in this chain is about `.ret`, not `.time`.

The Return-Value Proof Chain

1. `ret_merge`: timed merge returns ordinary merge.
2. `sorted_merge`: merging sorted lists is sorted.
3. `mergeSort_sorted`: recursive output is sorted.
4. `mergeSort_perm`: recursive output keeps all elements.

Every theorem in this chain is about `.ret`, not `.time`.

The Return-Value Proof Chain

1. `ret_merge`: timed merge returns ordinary merge.
2. `sorted_merge`: merging sorted lists is sorted.
3. `mergeSort_sorted`: recursive output is sorted.
4. `mergeSort_perm`: recursive output keeps all elements.
5. `mergeSort_correct`: combine the last two facts.

Every theorem in this chain is about `.ret`, not `.time`.

One Merge-Sort Run

Sort `[4, 1, 3, 2]`.

1. Split into `[4, 1]` and `[3, 2]`.

One Merge-Sort Run

Sort `[4, 1, 3, 2]`.

1. Split into `[4, 1]` and `[3, 2]`.
2. Split again until lists have length zero or one.

One Merge-Sort Run

Sort `[4, 1, 3, 2]`.

1. Split into `[4, 1]` and `[3, 2]`.
2. Split again until lists have length zero or one.
3. Merge `[4]` and `[1]` to get `[1, 4]`.

One Merge-Sort Run

Sort `[4, 1, 3, 2]`.

1. Split into `[4, 1]` and `[3, 2]`.
2. Split again until lists have length zero or one.
3. Merge `[4]` and `[1]` to get `[1, 4]`.
4. Merge `[3]` and `[2]` to get `[2, 3]`.

One Merge-Sort Run

Sort `[4, 1, 3, 2]`.

1. Split into `[4, 1]` and `[3, 2]`.
2. Split again until lists have length zero or one.
3. Merge `[4]` and `[1]` to get `[1, 4]`.
4. Merge `[3]` and `[2]` to get `[2, 3]`.
5. Merge both results to get `[1, 2, 3, 4]`.

Program Time Becomes a Recurrence

For $n \geq 2$, define

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n.$$

- The first two terms pay for recursive sorting.

Program Time Becomes a Recurrence

For $n \geq 2$, define

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n.$$

- The first two terms pay for recursive sorting.
- The last term uses the merge bound.

Program Time Becomes a Recurrence

For $n \geq 2$, define

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n.$$

- The first two terms pay for recursive sorting.
- The last term uses the merge bound.
- $T(0) = T(1) = 0$.

Program Time Becomes a Recurrence

For $n \geq 2$, define

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n.$$

- The first two terms pay for recursive sorting.
- The last term uses the merge bound.
- $T(0) = T(1) = 0$.
- This recurrence is a mathematical object separate from the program.

The Recurrence in Lean

```
def timeMergeSortRec : Nat -> Nat
  | 0 => 0
  | 1 => 0
  | n + 2 =>
      timeMergeSortRec ((n + 2) / 2) +
      timeMergeSortRec (((n + 2) - 1) / 2 + 1) +
      (n + 2)
```

Floor and ceiling halves appear as natural-number arithmetic.

Live Lean: `code/D6_04_ComplexityProof.lean`

How CSLib Finishes Return Correctness

The current CSLib proof uses functional induction on the actual definitions.

```
ret_merge := by
  fun_induction merge with
  grind [nil_merge, merge_right, cons_merge_cons]

mergeSort_sorted := by
  fun_induction mergeSort
  -- base cases: simplify
  -- recursive case: exact sorted_merge ih2 ih1

mergeSort_perm := by
  fun_induction mergeSort
  -- use merge_perm, Perm.append, take_append_drop
```

`fun_induction` creates one goal for each defining equation and gives induction hypotheses for recursive calls. The named lemmas supply the mathematics.

What `grind` Does in the Return Proof

In one nonempty merge case, CSLib already knows:

- the recursive timed merge returns the ordinary recursive merge;

`grind` selects and combines these local facts. It does not discover the definitions of sortedness, permutation, or the needed merge lemmas.

What `grind` Does in the Return Proof

In one nonempty merge case, CSLib already knows:

- the recursive timed merge returns the ordinary recursive merge;
- the comparison tells which head is chosen;

`grind` selects and combines these local facts. It does not discover the definitions of sortedness, permutation, or the needed merge lemmas.

What `grind` Does in the Return Proof

In one nonempty merge case, CSLib already knows:

- the recursive timed merge returns the ordinary recursive merge;
- the comparison tells which head is chosen;
- ordinary list merge has sortedness and permutation lemmas.

`grind` selects and combines these local facts. It does not discover the definitions of sortedness, permutation, or the needed merge lemmas.

How CSLib Proves the Merge Cost

```
theorem merge_time :
    (merge xs ys).time <= xs.length + ys.length := by
  fun_induction merge
  -- an empty input: simp
  -- two nonempty inputs:
  --   unfold one tick, use the recursive hypothesis,
  --   then grind the length arithmetic
```

Functional induction matches the two-list recursion. In every recursive case, exactly one head disappears, so the right side also drops by one.

From Program Time to the Recurrence

The proof of `mergeSort_time_le` follows the `do` block.

```
fun_induction mergeSort
-- rewrite each bind with time_bind
-- grw [merge_time]
-- rewrite recursive output lengths with
--   mergeSort_same_length
-- unfold timeMergeSortRec
-- grind
```

The length lemma is the bridge: the merge bound needs the lengths of the two sorted outputs, while the recurrence is written with the input half lengths.

Second Break: Trace One Merge Proof

Use ten minutes for one already prepared task.

1. In `D6_04_ComplexityProof_Exercises.lean`, evaluate the return value and time of merging `[1,4]` with `[2,3]`.

`native_decide` evaluates a closed decidable statement and asks Lean to check the resulting proof. It is useful here because every input is concrete.

Second Break: Trace One Merge Proof

Use ten minutes for one already prepared task.

1. In `D6_04_ComplexityProof_Exercises.lean`, evaluate the return value and time of merging `[1,4]` with `[2,3]`.
2. Mark where the three comparisons happen.

`native_decide` evaluates a closed decidable statement and asks Lean to check the resulting proof. It is useful here because every input is concrete.

Second Break: Trace One Merge Proof

Use ten minutes for one already prepared task.

1. In `D6_04_ComplexityProof_Exercises.lean`, evaluate the return value and time of merging `[1,4]` with `[2,3]`.
2. Mark where the three comparisons happen.
3. Prove the concrete return and time theorems.

`native_decide` evaluates a closed decidable statement and asks Lean to check the resulting proof. It is useful here because every input is concrete.

Second Break: Trace One Merge Proof

Use ten minutes for one already prepared task.

1. In `D6_04_ComplexityProof_Exercises.lean`, evaluate the return value and time of merging `[1,4]` with `[2,3]`.
2. Mark where the three comparisons happen.
3. Prove the concrete return and time theorems.
4. Then label each fact as a `ret` fact, a `time` fact, or the length bridge between them.

`native_decide` evaluates a closed decidable statement and asks Lean to check the resulting proof. It is useful here because every input is concrete.

Solving the Recurrence Is a Separate Proof

CSLib next proves logarithm facts for both half sizes.

```
clog2_half_le
clog2_floor_half_le
some_algebra
timeMergeSortRec_le
```

- The two `clog` lemmas control the depth of recursive calls.

Solving the Recurrence Is a Separate Proof

CSLib next proves logarithm facts for both half sizes.

```
clog2_half_le
clog2_floor_half_le
some_algebra
timeMergeSortRec_le
```

- The two `clog` lemmas control the depth of recursive calls.
- `some_algebra` rearranges the two half-size products.

Solving the Recurrence Is a Separate Proof

CSLib next proves logarithm facts for both half sizes.

```
clog2_half_le
clog2_floor_half_le
some_algebra
timeMergeSortRec_le
```

- The two `clog` lemmas control the depth of recursive calls.
- `some_algebra` rearranges the two half-size products.
- `timeMergeSortRec_le` uses functional induction on the recurrence.

Automation Finishes Local Work, Not the Design

The proof succeeds because the authors first supplied:

1. a tick exactly beside each comparison;

After that, `simp`, `grw`, and `grind` can close the small equations and arithmetic goals created by induction.

Automation Finishes Local Work, Not the Design

The proof succeeds because the authors first supplied:

1. a tick exactly beside each comparison;
2. return lemmas for merge;

After that, `simp`, `grw`, and `grind` can close the small equations and arithmetic goals created by induction.

Automation Finishes Local Work, Not the Design

The proof succeeds because the authors first supplied:

1. a tick exactly beside each comparison;
2. return lemmas for merge;
3. sortedness and permutation lemmas;

After that, `simp`, `grw`, and `grind` can close the small equations and arithmetic goals created by induction.

Automation Finishes Local Work, Not the Design

The proof succeeds because the authors first supplied:

1. a tick exactly beside each comparison;
2. return lemmas for merge;
3. sortedness and permutation lemmas;
4. the output-length bridge;

After that, `simp`, `grw`, and `grind` can close the small equations and arithmetic goals created by induction.

Automation Finishes Local Work, Not the Design

The proof succeeds because the authors first supplied:

1. a tick exactly beside each comparison;
2. return lemmas for merge;
3. sortedness and permutation lemmas;
4. the output-length bridge;
5. a recurrence and two logarithm bounds.

After that, `simp`, `grw`, and `grind` can close the small equations and arithmetic goals created by induction.

The Final CSLib Cost Theorem

For $n = \texttt{xs.length}$, CSLib proves

$$\texttt{(mergeSort xs).time} \leq n \cdot \texttt{Nat.clog 2 n}.$$

- It holds for every input list.

The Final CSLib Cost Theorem

For $n = \texttt{xs.length}$, CSLib proves

$$\texttt{(mergeSort xs).time} \leq n \cdot \texttt{Nat.clog 2 n}.$$

- It holds for every input list.
- It counts comparisons under the stated tick convention.

The Final CSLib Cost Theorem

For $n = \texttt{xs.length}$, CSLib proves

$$\texttt{(mergeSort xs).time} \leq n \cdot \texttt{Nat.clog 2 n}.$$

- It holds for every input list.
- It counts comparisons under the stated tick convention.
- It is a concrete inequality, stronger than only saying $O(n \log n)$.

Try Now: Classify the Theorems

Mark each theorem as `ret`, `time`, or a bridge between them.

1. `mergeSort_sorted`

Try Now: Classify the Theorems

Mark each theorem as `ret`, `time`, or a bridge between them.

1. `mergeSort_sorted`
2. `mergeSort_perm`

Try Now: Classify the Theorems

Mark each theorem as `ret`, `time`, or a bridge between them.

1. `mergeSort_sorted`
2. `mergeSort_perm`
3. `merge_time`

Try Now: Classify the Theorems

Mark each theorem as `ret`, `time`, or a bridge between them.

1. `mergeSort_sorted`
2. `mergeSort_perm`
3. `merge_time`
4. `mergeSort_same_length`

Try Now: Classify the Theorems

Mark each theorem as `ret`, `time`, or a bridge between them.

1. `mergeSort_sorted`
2. `mergeSort_perm`
3. `merge_time`
4. `mergeSort_same_length`
5. `mergeSort_time_le`

Try Now: Classify the Theorems

Mark each theorem as `ret`, `time`, or a bridge between them.

1. `mergeSort_sorted`
2. `mergeSort_perm`
3. `merge_time`
4. `mergeSort_same_length`
5. `mergeSort_time_le`

The length theorem is a return-value fact used inside a time proof.

Subsection Outline

Exact Bounds and Big-O Answer Different Questions

The exact theorem says what constant and logarithm CSLib proved for every list.
Big-O says that, for large n, one function is at most a constant multiple of another.

- Keep the exact theorem as the main verified result.

Exact Bounds and Big-O Answer Different Questions

The exact theorem says what constant and logarithm CSLib proved for every list.
Big-O says that, for large n, one function is at most a constant multiple of another.

- Keep the exact theorem as the main verified result.
- Add Big-O as a standard summary for comparison with other algorithms.

Exact Bounds and Big-O Answer Different Questions

The exact theorem says what constant and logarithm CSLib proved for every list.
Big-O says that, for large n, one function is at most a constant multiple of another.

- Keep the exact theorem as the main verified result.
- Add Big-O as a standard summary for comparison with other algorithms.
- Do not replace a precise theorem with a weaker slogan.

The Current CSLib File Stops at the Exact Bound

The merge-sort file does not currently state the result with Mathlib's `Asymptotics.IsBigO`. A rigorous bridge must:

1. turn list cost into a function of input size n;

The Current CSLib File Stops at the Exact Bound

The merge-sort file does not currently state the result with Mathlib's `Asymptotics.IsBigO`. A rigorous bridge must:

1. turn list cost into a function of input size n;
2. choose a normed codomain and the filter `atTop`;

The Current CSLib File Stops at the Exact Bound

The merge-sort file does not currently state the result with Mathlib's `Asymptotics.IsBigO`. A rigorous bridge must:

1. turn list cost into a function of input size n;
2. choose a normed codomain and the filter `atTop`;
3. relate `Nat.clog 2 n` to the chosen logarithm;

The Current CSLib File Stops at the Exact Bound

The merge-sort file does not currently state the result with Mathlib's `Asymptotics.IsBigO`. A rigorous bridge must:

1. turn list cost into a function of input size n;
2. choose a normed codomain and the filter `atTop`;
3. relate `Nat.clog 2 n` to the chosen logarithm;
4. prove an eventual inequality;

The Current CSLib File Stops at the Exact Bound

The merge-sort file does not currently state the result with Mathlib's `Asymptotics.IsBigO`. A rigorous bridge must:

1. turn list cost into a function of input size n;
2. choose a normed codomain and the filter `atTop`;
3. relate `Nat.clog 2 n` to the chosen logarithm;
4. prove an eventual inequality;
5. use `Asymptotics.IsBigO.of_bound` or a related lemma.

A Small Big-O Model for Practice

```
def EventuallyAtTop (property : Nat -> Prop) : Prop :=
  exists threshold,
    forall n, threshold <= n -> property n

def IsBigO (function comparison : Nat -> Nat) : Prop :=
  exists constant,
    0 < constant /\
    EventuallyAtTop
      (fun n => function n <= constant * comparison n)
```

This is simpler than Mathlib's norm-and-filter definition.

Live Lean: `code/D6_04_ComplexityProof.lean`

Try Now: Package a Pointwise Bound

```
theorem pointwise_linear_isBigO
    (function : Nat -> Nat)
    (constant : Nat)
    (constantPositive : 0 < constant)
    (bound : forall n, function n <= constant * n) :
    IsBigO function (fun n => n) := by
  sorry
```

Live Lean: code/D6_04_ComplexityProof_Exercises.lean

Try Now: Package a Pointwise Bound

```
theorem pointwise_linear_isBigO
    (function : Nat -> Nat)
    (constant : Nat)
    (constantPositive : 0 < constant)
    (bound : forall n, function n <= constant * n) :
    IsBigO function (fun n => n) := by
  sorry
```

Choose the same constant and threshold 0, then use `bound`.

Live Lean: `code/D6_04_ComplexityProof_Exercises.lean`

Check: Reuse the General Wrapper

```
theorem pointwise_linear_isBigO
    (function : Nat -> Nat)
    (constant : Nat)
    (constantPositive : 0 < constant)
    (bound : forall n, function n <= constant * n) :
    IsBigO function (fun n => n) := by
  exact pointwise_isBigO function (fun n => n)
    constant constantPositive bound
```

The full Mathlib proof has the same logical shape but richer types.

Section Outline

Subsection Outline

The Project Question

Can a small formal system translate a supported functional program into a symbolic upper bound?

- The first version handles only a documented first-order fragment.

The Project Question

Can a small formal system translate a supported functional program into a symbolic upper bound?

- The first version handles only a documented first-order fragment.
- The output is a `CostExpr` tree, not a wall-clock prediction.

The Project Question

Can a small formal system translate a supported functional program into a symbolic upper bound?

- The first version handles only a documented first-order fragment.
- The output is a `CostExpr` tree, not a wall-clock prediction.
- Every language construct receives an explicit cost rule.

The Project Question

Can a small formal system translate a supported functional program into a symbolic upper bound?

- The first version handles only a documented first-order fragment.
- The output is a `CostExpr` tree, not a wall-clock prediction.
- Every language construct receives an explicit cost rule.
- Reduction and automation should produce kernel-checked certificates.

Three Ways to Attach the Analyzer

Design	Input	Main proof duty
external	a small term AST	interpreter gives a sound bound
intrinsic	`TimeM` program	ticks match the chosen cost model
metaprogram	Lean syntax or `Lean.Expr`	translation produces a certificate

This project starts with the external design. Later, a translator can connect real Lean terms to the same AST. TiML shows a larger alternative in which indexed types carry sizes and time bounds.

Subsection Outline

Reuse the Interpreter Design from D5

The D5 and Haskell Scheme interpreter pipeline was

$$\text{source} \rightarrow \text{parser} \rightarrow \text{Expr AST} \rightarrow \text{environment} \rightarrow \text{value or error.}$$

The complexity project adds a second interpretation:

$$\text{FP term AST} \rightarrow \text{cost evaluator} \rightarrow \text{CostExpr tree} \rightarrow \text{upper bound.}$$

Both interpretations recurse over the same syntactic structure.

Choose a Small Supported Fragment

- natural and Boolean values;

General higher-order functions, arbitrary recursion, mutation, and IO are rejected until the language has cost rules for them.

Choose a Small Supported Fragment

- natural and Boolean values;
- variables and primitive addition;

General higher-order functions, arbitrary recursion, mutation, and IO are rejected until the language has cost rules for them.

Choose a Small Supported Fragment

- natural and Boolean values;
- variables and primitive addition;
- `if`;

General higher-order functions, arbitrary recursion, mutation, and IO are rejected until the language has cost rules for them.

Choose a Small Supported Fragment

- natural and Boolean values;
- variables and primitive addition;
- `if`;
- concrete and symbolic list inputs;

General higher-order functions, arbitrary recursion, mutation, and IO are rejected until the language has cost rules for them.

Choose a Small Supported Fragment

- natural and Boolean values;
- variables and primitive addition;
- `if`;
- concrete and symbolic list inputs;
- a fixed map operation and list append.

General higher-order functions, arbitrary recursion, mutation, and IO are rejected until the language has cost rules for them.

Values and Terms

```
inductive Value where
  | nat (value : Nat)
  | bool (value : Bool)
  | list (values : List Nat)

inductive Term where
  | nat (value : Nat)
  | bool (value : Bool)
  | variable (name : String)
  | add (left right : Term)
  | ifThenElse (condition thenBranch elseBranch : Term)
  | list (values : List Nat)
  | listInput (lengthName valueName : String)
  | mapAdd (increment : Nat)
      (functionCost : CostExpr) (source : Term)
  | append (left right : Term)
```

Live Lean: code/D6_05_AutomaticComplexity.lean

Value Semantics and Cost Semantics Are Separate

Value evaluator

Value Semantics and Cost Semantics Are Separate

Value evaluator

- Reads concrete values.

Value Semantics and Cost Semantics Are Separate

Value evaluator

- Reads concrete values.
- Returns a value or semantic error.

Value Semantics and Cost Semantics Are Separate

Value evaluator

- Reads concrete values.
- Returns a value or semantic error.
- Determines which branch executes.

Value Semantics and Cost Semantics Are Separate

Value evaluator

- Reads concrete values.
- Returns a value or semantic error.
- Determines which branch executes.

Cost evaluator

Value Semantics and Cost Semantics Are Separate

Value evaluator

- Reads concrete values.
- Returns a value or semantic error.
- Determines which branch executes.

Cost evaluator

- Reads syntax and symbolic lengths.

Value Semantics and Cost Semantics Are Separate

Value evaluator

- Reads concrete values.
- Returns a value or semantic error.
- Determines which branch executes.

Cost evaluator

- Reads syntax and symbolic lengths.
- Returns a formula or unsupported-case error.

Value Semantics and Cost Semantics Are Separate

Value evaluator

- Reads concrete values.
- Returns a value or semantic error.
- Determines which branch executes.

Cost evaluator

- Reads syntax and symbolic lengths.
- Returns a formula or unsupported-case error.
- Uses maximum for a branch-independent upper bound.

The Cost Expression Language

```
inductive CostExpr where
  | const (value : Nat)
  | var (name : String)
  | add (left right : CostExpr)
  | mul (left right : CostExpr)
  | max (left right : CostExpr)

def CostExpr.eval : CostExpr -> CostEnv -> Nat
  | .const value, _ => value
  | .var name, env => env name
  | .add left right, env =>
      left.eval env + right.eval env
  | .mul left right, env =>
      left.eval env * right.eval env
  | .max left right, env =>
      Nat.max (left.eval env) (right.eval env)
```

Live Lean: code/D6_05_AutomaticComplexity.lean

Write the Cost Rules Before Automation

The first model uses these conventions:

- constants, variables, and list inputs cost zero to access;

Write the Cost Rules Before Automation

The first model uses these conventions:

- constants, variables, and list inputs cost zero to access;
- primitive addition charges one after evaluating both operands;

Write the Cost Rules Before Automation

The first model uses these conventions:

- constants, variables, and list inputs cost zero to access;
- primitive addition charges one after evaluating both operands;
- `if` charges its condition, one branch decision, and the maximum of the branch bounds;

Write the Cost Rules Before Automation

The first model uses these conventions:

- constants, variables, and list inputs cost zero to access;
- primitive addition charges one after evaluating both operands;
- `if` charges its condition, one branch decision, and the maximum of the branch bounds;
- map charges source construction plus length times function cost;

Write the Cost Rules Before Automation

The first model uses these conventions:

- constants, variables, and list inputs cost zero to access;
- primitive addition charges one after evaluating both operands;
- `if` charges its condition, one branch decision, and the maximum of the branch bounds;
- map charges source construction plus length times function cost;
- append charges both source terms plus traversal of the left list.

Why `if` Uses a Maximum

For a branch-independent upper bound,

$$C(\texttt{if } b \texttt{ then } t \texttt{ else } e) = C(b) + 1 + \max(C(t), C(e)).$$

- Runtime executes only one branch.

Why `if` Uses a Maximum

For a branch-independent upper bound,

$$C(\texttt{if } b \texttt{ then } t \texttt{ else } e) = C(b) + 1 + \max(C(t), C(e)).$$

- Runtime executes only one branch.
- Without knowing the condition value statically, either branch may run.

Why `if` Uses a Maximum

For a branch-independent upper bound,

$$C(\texttt{if } b \texttt{ then } t \texttt{ else } e) = C(b) + 1 + \max(C(t), C(e)).$$

- Runtime executes only one branch.
- Without knowing the condition value statically, either branch may run.
- Summing both branch costs is sound but unnecessarily weak.

Why `if` Uses a Maximum

For a branch-independent upper bound,

$$C(\texttt{if } b \texttt{ then } t \texttt{ else } e) = C(b) + 1 + \max(C(t), C(e)).$$

- Runtime executes only one branch.
- Without knowing the condition value statically, either branch may run.
- Summing both branch costs is sound but unnecessarily weak.
- A value-sensitive analyzer may simplify the condition and choose one branch.

The `if` Cost Rule

```
def Term.cost : Term -> Except String CostExpr
  | .ifThenElse condition thenBranch elseBranch => do
      let conditionCost <- condition.cost
      let thenCost <- thenBranch.cost
      let elseCost <- elseBranch.cost
      pure (.add conditionCost
        (.add (.const 1) (.max thenCost elseCost)))
  | ...
```

Returning `Except` lets the analyzer reject a term whose cost semantics is not yet defined.

Live Lean: `code/D6_05_AutomaticComplexity.lean`

Map Needs a Symbolic Length

For source cost C_s, symbolic length n, and constant function bound c_f, the rule is

$$C(\texttt{map}\ f\ xs) = C_s + n \cdot c_f.$$

- A concrete list contributes its known length.

Map Needs a Symbolic Length

For source cost C_s, symbolic length n, and constant function bound c_f, the rule is

$$C(\texttt{map}\ f\ xs) = C_s + n \cdot c_f.$$

- A concrete list contributes its known length.
- A symbolic input contributes a variable such as `"n"`.

Map Needs a Symbolic Length

For source cost C_s, symbolic length n, and constant function bound c_f, the rule is

$$C(\texttt{map}\ f\ xs) = C_s + n \cdot c_f.$$

- A concrete list contributes its known length.
- A symbolic input contributes a variable such as `"n"`.
- Mapping preserves list length, so nested map analysis can reuse it.

Map Needs a Symbolic Length

For source cost C_s, symbolic length n, and constant function bound c_f, the rule is

$$C(\texttt{map}\ f\ xs) = C_s + n \cdot c_f.$$

- A concrete list contributes its known length.
- A symbolic input contributes a variable such as `"n"`.
- Mapping preserves list length, so nested map analysis can reuse it.
- Value-dependent function cost requires a sum or a separately proved bound.

Shape Analysis Is a Separate Interpreter

```
def Term.lengthExpr : Term -> Option CostExpr
  | .list values => some (.const values.length)
  | .listInput lengthName _ => some (.var lengthName)
  | .mapAdd _ _ source => source.lengthExpr
  | .append left right => do
      let leftLength <- left.lengthExpr
      let rightLength <- right.lengthExpr
      pure (.add leftLength rightLength)
  | _ => none
```

The analyzer refuses to invent a list length for a term outside the list fragment.

Live Lean: `code/D6_05_AutomaticComplexity.lean`

Map and Append Build Cost Trees

```
| .mapAdd _ functionCost source => do
    let sourceCost <- source.cost
    match source.lengthExpr with
    | some length =>
        pure (.add sourceCost (.mul length functionCost))
    | none => throw "map source has no list length"

| .append left right => do
    let leftCost <- left.cost
    let rightCost <- right.cost
    match left.lengthExpr with
    | some leftLength =>
        pure (.add (.add leftCost rightCost) leftLength)
    | none => throw "append left side has no list length"
```

Live Lean: code/D6_05_AutomaticComplexity.lean

From an Expression Tree to a Time-Cost Tree

Source part	Cost part
condition	C_{cond}
branch choice	1
map branch	nc_f
empty-list branch	0
both possible branches	$\max(nc_f, 0)$

$$C_{\text{cond}} + 1 + \max(nc_f, 0).$$

Expansion is structural: each syntax constructor contributes one declared rule.

Try Now: Derive a Map Cost

```
theorem map_input_cost
    (increment : Nat) (functionCost : CostExpr)
    (lengthName valueName : String) :
    (Term.mapAdd increment functionCost
      (.listInput lengthName valueName)).cost =
      Except.ok
        (.add (.const 0)
          (.mul (.var lengthName) functionCost)) := by
  sorry
```

Live Lean: code/D6_05_AutomaticComplexity_Exercises.lean

Try Now: Derive a Map Cost

```
theorem map_input_cost
    (increment : Nat) (functionCost : CostExpr)
    (lengthName valueName : String) :
    (Term.mapAdd increment functionCost
      (.listInput lengthName valueName)).cost =
      Except.ok
        (.add (.const 0)
          (.mul (.var lengthName) functionCost)) := by
  sorry
```

Read it as source cost 0, plus length times one-call cost.

Live Lean: `code/D6_05_AutomaticComplexity_Exercises.lean`

Reduction Must Preserve Evaluation

```
def CostExpr.reduceZeroLeft : CostExpr -> CostExpr
  | .add (.const 0) right => right
  | expression => expression

theorem CostExpr.reduceZeroLeft_sound
    (env : CostEnv) (expression : CostExpr) :
    expression.reduceZeroLeft.eval env =
      expression.eval env := by
  cases expression with
  | add left right =>
      cases left with
      | const value =>
          cases value <;>
            simp [CostExpr.reduceZeroLeft, CostExpr.eval]
      | _ => rfl
  | _ => rfl
```

Live Lean: code/D6_05_AutomaticComplexity.lean

Cost Rules Should Not Be Unchecked Axioms

The project may call its rewrite clauses cost axioms informally, but the Lean implementation should prefer definitions and proved lemmas.

- An axiom asserting any inferred cost makes later proofs conditional.

Cost Rules Should Not Be Unchecked Axioms

The project may call its rewrite clauses cost axioms informally, but the Lean implementation should prefer definitions and proved lemmas.

- An axiom asserting any inferred cost makes later proofs conditional.
- A definition computes a formula from supported syntax.

Cost Rules Should Not Be Unchecked Axioms

The project may call its rewrite clauses cost axioms informally, but the Lean implementation should prefer definitions and proved lemmas.

- An axiom asserting any inferred cost makes later proofs conditional.
- A definition computes a formula from supported syntax.
- A theorem connects that formula to an intrinsic or operational cost model.

Cost Rules Should Not Be Unchecked Axioms

The project may call its rewrite clauses cost axioms informally, but the Lean implementation should prefer definitions and proved lemmas.

- An axiom asserting any inferred cost makes later proofs conditional.
- A definition computes a formula from supported syntax.
- A theorem connects that formula to an intrinsic or operational cost model.
- Kernel checking is useful only when the generated certificate has content.

Certificate-Producing Automation

- Bugs in the metaprogram may produce a rejected certificate.

Certificate-Producing Automation

- Bugs in the metaprogram may produce a rejected certificate.
- The trusted theorem states what the certificate actually establishes.

Certificate-Producing Automation

- Bugs in the metaprogram may produce a rejected certificate.
- The trusted theorem states what the certificate actually establishes.
- Definition fidelity and cost conventions still require human review.

Minimum Project Milestones

1. Define `CostExpr`, its environment, and evaluator.

Minimum Project Milestones

1. Define `CostExpr`, its environment, and evaluator.
2. Define the supported `Term` language and value evaluator.

Minimum Project Milestones

1. Define `CostExpr`, its environment, and evaluator.
2. Define the supported `Term` language and value evaluator.
3. Implement symbolic length and cost interpretations.

Minimum Project Milestones

1. Define `CostExpr`, its environment, and evaluator.
2. Define the supported `Term` language and value evaluator.
3. Implement symbolic length and cost interpretations.
4. Prove one simplification rule sound.

Minimum Project Milestones

1. Define `CostExpr`, its environment, and evaluator.
2. Define the supported `Term` language and value evaluator.
3. Implement symbolic length and cost interpretations.
4. Prove one simplification rule sound.
5. Give executable examples for `if`, map, and append.

Minimum Project Milestones

1. Define `CostExpr`, its environment, and evaluator.
2. Define the supported `Term` language and value evaluator.
3. Implement symbolic length and cost interpretations.
4. Prove one simplification rule sound.
5. Give executable examples for `if`, map, and append.
6. Document every supported and rejected construct.

What the Project Does Not Promise

- It is not a general analyzer for arbitrary Lean programs.

What the Project Does Not Promise

- It is not a general analyzer for arbitrary Lean programs.
- It does not infer real hardware execution time.

What the Project Does Not Promise

- It is not a general analyzer for arbitrary Lean programs.
- It does not infer real hardware execution time.
- It does not treat all higher-order functions as constant cost.

What the Project Does Not Promise

- It is not a general analyzer for arbitrary Lean programs.
- It does not infer real hardware execution time.
- It does not treat all higher-order functions as constant cost.
- It does not replace recurrence mathematics with syntax traversal.

What the Project Does Not Promise

- It is not a general analyzer for arbitrary Lean programs.
- It does not infer real hardware execution time.
- It does not treat all higher-order functions as constant cost.
- It does not replace recurrence mathematics with syntax traversal.
- It does provide a small, testable core that can grow by proved rules.

Subsection Outline

Final Writing Block: 2:50–3:00

Write a one-page project outline containing:

- the supported grammar;

Final Writing Block: 2:50–3:00

Write a one-page project outline containing:

- the supported grammar;
- the value and cost semantics;

Final Writing Block: 2:50–3:00

Write a one-page project outline containing:

- the supported grammar;
- the value and cost semantics;
- the theorem that makes the analyzer trustworthy;

Final Writing Block: 2:50–3:00

Write a one-page project outline containing:

- the supported grammar;
- the value and cost semantics;
- the theorem that makes the analyzer trustworthy;
- one unsupported example and its error;

Final Writing Block: 2:50–3:00

Write a one-page project outline containing:

- the supported grammar;
- the value and cost semantics;
- the theorem that makes the analyzer trustworthy;
- one unsupported example and its error;
- the smallest demonstration that can be completed first.

References: Formalization and Verification

- Duolei Wang, *Scheme-Interpreter*: `https://github.com/Duolei-Wang/Scheme-Interpreter`

References: Formalization and Verification

- Duolei Wang, *Scheme-Interpreter*:
  `https://github.com/Duolei-Wang/Scheme-Interpreter`
- *Write Yourself a Scheme in 48 Hours*:
  `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`

References: Formalization and Verification

- Duolei Wang, *Scheme-Interpreter*:
  `https://github.com/Duolei-Wang/Scheme-Interpreter`
- *Write Yourself a Scheme in 48 Hours*:
  `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`
- Sonoda et al., *Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral*:
  `https://arxiv.org/abs/2503.19605`

References: Formalization and Verification

- Duolei Wang, *Scheme-Interpreter*:
  `https://github.com/Duolei-Wang/Scheme-Interpreter`
- *Write Yourself a Scheme in 48 Hours*:
  `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`
- Sonoda et al., *Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral*:
  `https://arxiv.org/abs/2503.19605`
- Companion Lean code: `https://github.com/auto-res/lean-rademacher`

References: Formalization and Verification

- Duolei Wang, *Scheme-Interpreter*:
  `https://github.com/Duolei-Wang/Scheme-Interpreter`
- *Write Yourself a Scheme in 48 Hours*:
  `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`
- Sonoda et al., *Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral*:
  `https://arxiv.org/abs/2503.19605`
- Companion Lean code: `https://github.com/auto-res/lean-rademacher`
- Leroy, *A formally verified compiler back-end*:
  `https://arxiv.org/abs/0902.2137`

References: Formalization and Verification

- Duolei Wang, *Scheme-Interpreter*: `https://github.com/Duolei-Wang/Scheme-Interpreter`
- *Write Yourself a Scheme in 48 Hours*: `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`
- Sonoda et al., *Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral*: `https://arxiv.org/abs/2503.19605`
- Companion Lean code: `https://github.com/auto-res/lean-rademacher`
- Leroy, *A formally verified compiler back-end*: `https://arxiv.org/abs/0902.2137`
- seL4 verification research: `https://sel4.systems/Research/`

References: Formalization and Verification

- Duolei Wang, *Scheme-Interpreter*: `https://github.com/Duolei-Wang/Scheme-Interpreter`
- *Write Yourself a Scheme in 48 Hours*: `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`
- Sonoda et al., *Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral*: `https://arxiv.org/abs/2503.19605`
- Companion Lean code: `https://github.com/auto-res/lean-rademacher`
- Leroy, *A formally verified compiler back-end*: `https://arxiv.org/abs/0902.2137`
- seL4 verification research: `https://sel4.systems/Research/`
- CakeML verified language and compiler: `https://cakeml.org/`

References: Local Libraries and Complexity

- Discrete rate-distortion project:
  `https://github.com/Mu-Tsing/Discreted-Rate-Distortion`

References: Local Libraries and Complexity

- Discrete rate-distortion project: `https://github.com/Mu-Tsing/Discreted-Rate-Distortion`
- CSLib repository: `https://github.com/leanprover/cslib`

References: Local Libraries and Complexity

- Discrete rate-distortion project: `https://github.com/Mu-Tsing/Discreted-Rate-Distortion`
- CSLib repository: `https://github.com/leanprover/cslib`
- CSLib timed merge sort: `https://github.com/leanprover/cslib/blob/main/Cslib/Algorithms/Lean/MergeSort/MergeSort.lean`

References: Local Libraries and Complexity

- Discrete rate-distortion project:
  `https://github.com/Mu-Tsing/Discreted-Rate-Distortion`
- CSLib repository: `https://github.com/leanprover/cslib`
- CSLib timed merge sort:
  `https://github.com/leanprover/cslib/blob/main/Cslib/Algorithms/Lean/MergeSort/MergeSort.lean`
- CSLib `TimeM` API:
  `https://api.cslib.io/docs/Cslib/Algorithms/Lean/TimeM.html`

References: Local Libraries and Complexity

- Discrete rate-distortion project:
  `https://github.com/Mu-Tsing/Discreted-Rate-Distortion`
- CSLib repository: `https://github.com/leanprover/cslib`
- CSLib timed merge sort:
  `https://github.com/leanprover/cslib/blob/main/Cslib/Algorithms/Lean/MergeSort/MergeSort.lean`
- CSLib `TimeM` API:
  `https://api.cslib.io/docs/Cslib/Algorithms/Lean/TimeM.html`
- Mathlib asymptotics:
  `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Analysis/Asymptotics/Defs.html`

References: Local Libraries and Complexity

- Discrete rate-distortion project:
  `https://github.com/Mu-Tsing/Discreted-Rate-Distortion`
- CSLib repository: `https://github.com/leanprover/cslib`
- CSLib timed merge sort:
  `https://github.com/leanprover/cslib/blob/main/Cslib/Algorithms/Lean/MergeSort/MergeSort.lean`
- CSLib `TimeM` API:
  `https://api.cslib.io/docs/Cslib/Algorithms/Lean/TimeM.html`
- Mathlib asymptotics:
  `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Analysis/Asymptotics/Defs.html`
- Wang, Wang, and Chlipala, *TiML*:
  `https://adam.chlipala.net/papers/TimlOOPSLA17/`

References: Local Libraries and Complexity

- Discrete rate-distortion project:
  `https://github.com/Mu-Tsing/Discreted-Rate-Distortion`
- CSLib repository: `https://github.com/leanprover/cslib`
- CSLib timed merge sort:
  `https://github.com/leanprover/cslib/blob/main/Cslib/Algorithms/Lean/MergeSort/MergeSort.lean`
- CSLib `TimeM` API:
  `https://api.cslib.io/docs/Cslib/Algorithms/Lean/TimeM.html`
- Mathlib asymptotics:
  `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Analysis/Asymptotics/Defs.html`
- Wang, Wang, and Chlipala, *TiML*:
  `https://adam.chlipala.net/papers/TimlOOPSLA17/`
- Meek et al., *Formalizing Numerical Analysis*:
  `https://arxiv.org/abs/2606.14000`

Section Outline

4. Reference Case Studies
   - Top-Down Error Bound
   - Verification of a System
   - Local Theory to Mathlib

Subsection Outline

The Formalization Case Study

Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral

Sho Sonoda[1]
RIKEN AIP, Tokyo, Japan
CyberAgent Inc., Tokyo, Japan
Kazumi Kasaura
OMRON SINIC X Corporation, Tokyo, Japan
Yuma Mizuno
University College Cork, Cork, Ireland
Kei Tsukamoto
The University of Tokyo, Tokyo, Japan
Naoto Onda
OMRON SINIC X Corporation, Tokyo, Japan

Abstract

Understanding and certifying the generalization performance of machine learning algorithms—i.e. obtaining theoretical estimates of the test error from the training error—is a central theme of statistical learning theory. Among the many complexity measures used to derive such guarantees, *Rademacher complexity* yields sharp, data-dependent bounds that apply well beyond classical VC-dimension theory. In this study, we formalize the generalization error bound by *Rademacher complexity* in Lean 4, building on measure-theoretic probability theory available in the Mathlib library. Our development provides a mechanically-checked pipeline from the definitions of empirical and expected Rademacher complexity, through a formal symmetrization argument and a bounded-differences analysis, to high-probability uniform deviation bounds via a formally proved McDiarmid inequality. A key technical contribution is a reusable mechanism for lifting results from *countable* hypothesis classes (where measurability of suprema is straightforward in Mathlib) to *separable* topological index sets via a reduction to a countable dense subset. As worked applications of the abstract theorem, we mechanize standard empirical Rademacher bounds for linear predictors under ℓ_2 and ℓ_1 regularizations, and we also formalize a Dudley-type entropy integral bound based on covering numbers and a chaining construction.

2012 ACM Subject Classification Theory of computation → Sample complexity and generalization

The Formalization Case Study

Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral

Sho Sonoda[1]
RIKEN AIP, Tokyo, Japan
CyberAgent Inc., Tokyo, Japan
Kazumi Kasaura
OMRON SINIC X Corporation, Tokyo, Japan
Yuma Mizuno
University College Cork, Cork, Ireland
Kei Tsukamoto
The University of Tokyo, Tokyo, Japan
Naoto Onda
OMRON SINIC X Corporation, Tokyo, Japan

Abstract

Understanding and certifying the generalization performance of machine learning algorithms—i.e. obtaining theoretical estimates of the test error from the training error—is a central theme of statistical learning theory. Among the many complexity measures used to derive such guarantees, *Rademacher complexity* yields sharp, data-dependent bounds that apply well beyond classical VC-dimension theory. In this study, we formalize the generalization error bound by *Rademacher complexity* in Lean 4, building on measure-theoretic probability theory available in the Mathlib library. Our development provides a mechanically-checked pipeline from the definitions of empirical and expected Rademacher complexity, through a formal symmetrization argument and a bounded-differences analysis, to high-probability uniform deviation bounds via a formally proved McDiarmid inequality. A key technical contribution is a reusable mechanism for lifting results from *countable* hypothesis classes (where measurability of suprema is straightforward in Mathlib) to *separable* topological index sets via a reduction to a countable dense subset. As worked applications of the abstract theorem, we mechanize standard empirical Rademacher bounds for linear predictors under ℓ_2 and ℓ_1 regularizations, and we also formalize a Dudley-type entropy integral bound based on covering numbers and a chaining construction.

2012 ACM Subject Classification Theory of computation → Sample complexity and generalization

Start from the wanted result:

Sonoda et al., arXiv:2503.19605

The Formalization Case Study

Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral

Sho Sonoda[1]
RIKEN AIP, Tokyo, Japan
CyberAgent Inc., Tokyo, Japan
Kazumi Kasaura
OMRON SINIC X Corporation, Tokyo, Japan
Yuma Mizuno
University College Cork, Cork, Ireland
Kei Tsukamoto
The University of Tokyo, Tokyo, Japan
Naoto Onda
OMRON SINIC X Corporation, Tokyo, Japan

Abstract

Understanding and certifying the generalization performance of machine learning algorithms—i.e. obtaining theoretical estimates of the test error from the training error—is a central theme of statistical learning theory. Among the many complexity measures used to derive such guarantees, *Rademacher complexity* yields sharp, data-dependent bounds that apply well beyond classical VC-dimension theory. In this study, we formalize the generalization error bound by *Rademacher complexity* in Lean 4, building on measure-theoretic probability theory available in the Mathlib library. Our development provides a mechanically-checked pipeline from the definitions of empirical and expected Rademacher complexity, through a formal symmetrization argument and a bounded-differences analysis, to high-probability uniform deviation bounds via a formally proved McDiarmid inequality. A key technical contribution is a reusable mechanism for lifting results from *countable* hypothesis classes (where measurability of suprema is straightforward in Mathlib) to *separable* topological index sets via a reduction to a countable dense subset. As worked applications of the abstract theorem, we mechanize standard empirical Rademacher bounds for linear predictors under ℓ_2 and ℓ_1 regularizations, and we also formalize a Dudley-type entropy integral bound based on covering numbers and a chaining construction.

2012 ACM Subject Classification Theory of computation → Sample complexity and generalization

Start from the wanted result:

- a learning procedure sees a finite sample;

Sonoda et al., arXiv:2503.19605

The Formalization Case Study

Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral

Sho Sonoda[1]
RIKEN AIP, Tokyo, Japan
CyberAgent Inc., Tokyo, Japan
Kazumi Kasaura
OMRON SINIC X Corporation, Tokyo, Japan
Yuma Mizuno
University College Cork, Cork, Ireland
Kei Tsukamoto
The University of Tokyo, Tokyo, Japan
Naoto Onda
OMRON SINIC X Corporation, Tokyo, Japan

Abstract

Understanding and certifying the generalization performance of machine learning algorithms—i.e. obtaining theoretical estimates of the test error from the training error—is a central theme of statistical learning theory. Among the many complexity measures used to derive such guarantees, *Rademacher complexity* yields sharp, data-dependent bounds that apply well beyond classical VC-dimension theory. In this study, we formalize the generalization error bound by *Rademacher complexity* in Lean 4, building on measure-theoretic probability theory available in the Mathlib library. Our development provides a mechanically-checked pipeline from the definitions of empirical and expected Rademacher complexity, through a formal symmetrization argument and a bounded-differences analysis, to high-probability uniform deviation bounds via a formally proved McDiarmid inequality. A key technical contribution is a reusable mechanism for lifting results from *countable* hypothesis classes (where measurability of suprema is straightforward in Mathlib) to *separable* topological index sets via a reduction to a countable dense subset. As worked applications of the abstract theorem, we mechanize standard empirical Rademacher bounds for linear predictors under ℓ_2 and ℓ_1 regularizations, and we also formalize a Dudley-type entropy integral bound based on covering numbers and a chaining construction.

2012 ACM Subject Classification Theory of computation → Sample complexity and generalization

Start from the wanted result:

- a learning procedure sees a finite sample;
- it chooses a model from a class;

Sonoda et al., arXiv:2503.19605

The Formalization Case Study

Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral

Sho Sonoda[1]
RIKEN AIP, Tokyo, Japan
CyberAgent Inc., Tokyo, Japan
Kazumi Kasaura
OMRON SINIC X Corporation, Tokyo, Japan
Yuma Mizuno
University College Cork, Cork, Ireland
Kei Tsukamoto
The University of Tokyo, Tokyo, Japan
Naoto Onda
OMRON SINIC X Corporation, Tokyo, Japan

Abstract

Understanding and certifying the generalization performance of machine learning algorithms—i.e. obtaining theoretical estimates of the test error from the training error—is a central theme of statistical learning theory. Among the many complexity measures used to derive such guarantees, *Rademacher complexity* yields sharp, data-dependent bounds that apply well beyond classical VC-dimension theory. In this study, we formalize the generalization error bound by *Rademacher complexity* in Lean 4, building on measure-theoretic probability theory available in the Mathlib library. Our development provides a mechanically-checked pipeline from the definitions of empirical and expected Rademacher complexity, through a formal symmetrization argument and a bounded-differences analysis, to high-probability uniform deviation bounds via a formally proved McDiarmid inequality. A key technical contribution is a reusable mechanism for lifting results from *countable* hypothesis classes (where measurability of suprema is straightforward in Mathlib) to *separable* topological index sets via a reduction to a countable dense subset. As worked applications of the abstract theorem, we mechanize standard empirical Rademacher bounds for linear predictors under ℓ_2 and ℓ_1 regularizations, and we also formalize a Dudley-type entropy integral bound based on covering numbers and a chaining construction.

2012 ACM Subject Classification Theory of computation → Sample complexity and generalization

Start from the wanted result:

- a learning procedure sees a finite sample;
- it chooses a model from a class;
- we want a bound on new, unseen data;

Sonoda et al., arXiv:2503.19605

The Formalization Case Study

Lean Formalization of Generalization Error Bound by Rademacher Complexity and Dudley's Entropy Integral

Sho Sonoda[1]
RIKEN AIP, Tokyo, Japan
CyberAgent Inc., Tokyo, Japan
Kazumi Kasaura
OMRON SINIC X Corporation, Tokyo, Japan
Yuma Mizuno
University College Cork, Cork, Ireland
Kei Tsukamoto
The University of Tokyo, Tokyo, Japan
Naoto Onda
OMRON SINIC X Corporation, Tokyo, Japan

Abstract

Understanding and certifying the generalization performance of machine learning algorithms—i.e. obtaining theoretical estimates of the test error from the training error—is a central theme of statistical learning theory. Among the many complexity measures used to derive such guarantees, *Rademacher complexity* yields sharp, data-dependent bounds that apply well beyond classical VC-dimension theory. In this study, we formalize the generalization error bound by *Rademacher* complexity in Lean 4, building on measure-theoretic probability theory available in the Mathlib library. Our development provides a mechanically-checked pipeline from the definitions of empirical and expected Rademacher complexity, through a formal symmetrization argument and a bounded-differences analysis, to high-probability uniform deviation bounds via a formally proved McDiarmid inequality. A key technical contribution is a reusable mechanism for lifting results from *countable* hypothesis classes (where measurability of suprema is straightforward in Mathlib) to *separable* topological index sets via a reduction to a countable dense subset. As worked applications of the abstract theorem, we mechanize standard empirical Rademacher bounds for linear predictors under ℓ_2 and ℓ_1 regularizations, and we also formalize a Dudley-type entropy integral bound based on covering numbers and a chaining construction.

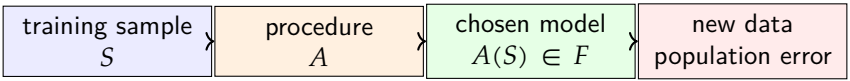
2012 ACM Subject Classification Theory of computation → Sample complexity and generalization

Start from the wanted result:

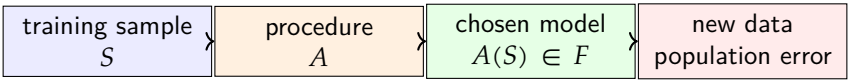
- a learning procedure sees a finite sample;
- it chooses a model from a class;
- we want a bound on new, unseen data;
- the proof is checked in Lean using Mathlib probability.

Sonoda et al., arXiv:2503.19605

Motivation: A Procedure Chooses After Seeing Data



Motivation: A Procedure Chooses After Seeing Data



Because $A(S)$ depends on S, a bound for one fixed model is not enough. We need one event that covers every model in F.

A Tiny Error Calculation

Input	True label	Model output	Loss
false	false	false	0
true	true	true	0
false	true	false	1

$$\hat{L}_S(f) = \frac{0+0+1}{3} = \frac{1}{3}, \qquad L(f) = \mathbb{E}[\ell(f, X)].$$

The sample error is known. The population error depends on the unknown data source.

The Same Tiny Example Runs in Lean

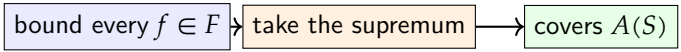
```
structure LabeledBool where
  input : Bool
  label : Bool

structure TinyClassifier where
  predict : Bool -> Bool

def mistake (h : TinyClassifier) (x : LabeledBool) : Nat :=
  if h.predict x.input = x.label then 0 else 1

def empiricalMistakes (h : TinyClassifier) :
    List LabeledBool -> Nat
  | [] => 0
  | x :: xs => mistake h x + empiricalMistakes h xs
```

Live Lean: code/D6_02_FormalizationVerification.lean

Target 1: Make the Bound Uniform

Define the worst gap in the whole class:

$$\Delta_S(F) = \sup_{f \in F} \left| \hat{L}_S(f) - L(f) \right|.$$



If $\Delta_S(F) \leq B$, then the selected model also has gap at most B.

Target 2: State the Bad Event

The optimized countable-class result controls

$$\Pr(2\mathcal{R}_n(F) + \varepsilon \le \Delta_S(F)) \le \exp\left(-\frac{\varepsilon^2 n}{2b^2}\right).$$

- n: number of sample points.

Target 2: State the Bad Event

The optimized countable-class result controls

$$\Pr(2\mathcal{R}_n(F) + \varepsilon \le \Delta_S(F)) \le \exp\left(-\frac{\varepsilon^2 n}{2b^2}\right).$$

- n: number of sample points.
- b: a bound with $|f(x)| \le b$.

Target 2: State the Bad Event

The optimized countable-class result controls

$$\Pr(2\mathcal{R}_n(F) + \varepsilon \le \Delta_S(F)) \le \exp\left(-\frac{\varepsilon^2 n}{2b^2}\right).$$

- n: number of sample points.
- b: a bound with $|f(x)| \le b$.
- $\mathcal{R}_n(F)$: expected Rademacher complexity.

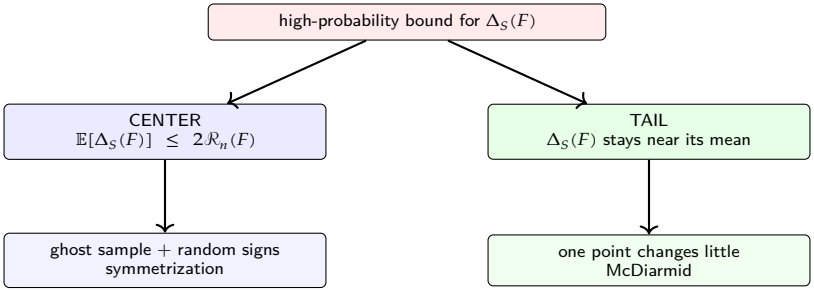
Target 2: State the Bad Event

The optimized countable-class result controls

$$\Pr(2\mathcal{R}_n(F) + \varepsilon \le \Delta_S(F)) \le \exp\left(-\frac{\varepsilon^2 n}{2b^2}\right).$$

- n: number of sample points.
- b: a bound with $|f(x)| \le b$.
- $\mathcal{R}_n(F)$: expected Rademacher complexity.
- The left side is the probability that the wanted bound fails.

Now Split the Goal Top-Down



The split tells us which definitions and lemmas must exist.

Center Branch: Why Random Signs Appear

Population error is hard to compare directly with one sample.

1. Introduce an independent ghost sample S'.

This route forces the definitions of `Signs` and Rademacher complexity.

Center Branch: Why Random Signs Appear

Population error is hard to compare directly with one sample.

1. Introduce an independent ghost sample S'.
2. Compare the averages on S and S'.

This route forces the definitions of `Signs` and Rademacher complexity.

Center Branch: Why Random Signs Appear

Population error is hard to compare directly with one sample.

1. Introduce an independent ghost sample S'.
2. Compare the averages on S and S'.
3. Use random signs $\sigma_k \in \{-1, +1\}$ to swap the two samples.

This route forces the definitions of `Signs` and Rademacher complexity.

Center Branch: Why Random Signs Appear

Population error is hard to compare directly with one sample.

1. Introduce an independent ghost sample S'.
2. Compare the averages on S and S'.
3. Use random signs $\sigma_k \in \{-1, +1\}$ to swap the two samples.
4. Bound the signed supremum by Rademacher complexity.

This route forces the definitions of `Signs` and Rademacher complexity.

Formal Definition Forced by the Center Branch

For one sample S, define

$$\widehat{\mathcal{R}}_S(F) = \mathbb{E}_\sigma \left[\sup_{f \in F} \left| \frac{1}{n} \sum_{k=1}^{n} \sigma_k f(S_k) \right| \right].$$

Then average over samples:

$$\mathcal{R}_n(F) = \mathbb{E}_S[\widehat{\mathcal{R}}_S(F)].$$

The project theorem `expectation_le_rademacher` closes this branch.

Tail Branch: Why Bounded Difference Appears

Replace only the j-th sample point:

$$S = (S_1, \dots, S_j, \dots, S_n) \quad \rightsquigarrow \quad S' = (S_1, \dots, S'_j, \dots, S_n).$$

- Only one term in each sample average changes.

Tail Branch: Why Bounded Difference Appears

Replace only the j-th sample point:

$$S = (S_1, \dots, S_j, \dots, S_n) \quad \rightsquigarrow \quad S' = (S_1, \dots, S'_j, \dots, S_n).$$

- Only one term in each sample average changes.
- $|f(x)| \leq b$ controls the size of that change.

Tail Branch: Why Bounded Difference Appears

Replace only the j-th sample point:

$$S = (S_1, \dots, S_j, \dots, S_n) \quad \rightsquigarrow \quad S' = (S_1, \dots, S'_j, \dots, S_n).$$

- Only one term in each sample average changes.
- $|f(x)| \le b$ controls the size of that change.
- Therefore $S \mapsto \Delta_S(F)$ has bounded difference.

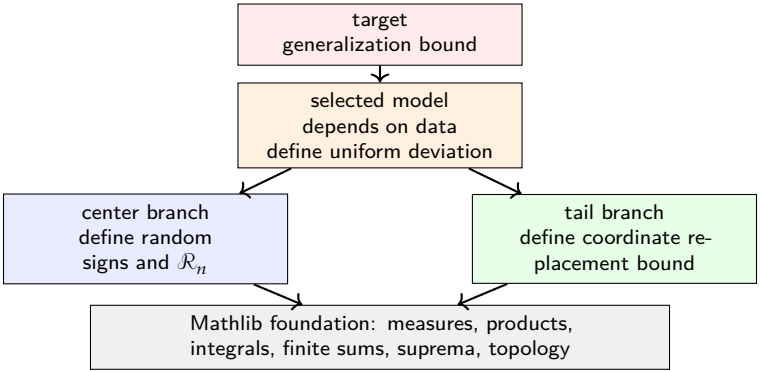
Tail Branch: Why Bounded Difference Appears

Replace only the j-th sample point:

$$S = (S_1, \dots, S_j, \dots, S_n) \quad \rightsquigarrow \quad S' = (S_1, \dots, S'_j, \dots, S_n).$$

- Only one term in each sample average changes.
- $|f(x)| \le b$ controls the size of that change.
- Therefore $S \mapsto \Delta_S(F)$ has bounded difference.
- McDiarmid turns this local fact into a probability tail.

Definitions Are Outputs of the Proof Plan



The Project's Basic Definition API

```
def Signs (n : Nat) : Type :=
  Fin n -> {-1, 1}

def empiricalRademacherComplexity
    (n : Nat) (f : Iota -> X -> Real)
    (S : Fin n -> X) : Real := ...

def rademacherComplexity
    (n : Nat) (f : Iota -> X -> Real)
    (mu : Measure Omega) (Xvar : Omega -> X) : Real := ...

def uniformDeviation
    (n : Nat) (f : Iota -> X -> Real)
    (mu : Measure Omega) (Xvar : Omega -> X)
    (S : Fin n -> X) : Real := ...
```

These are simplified signatures; `FoML/Defs.lean` contains the exact code.

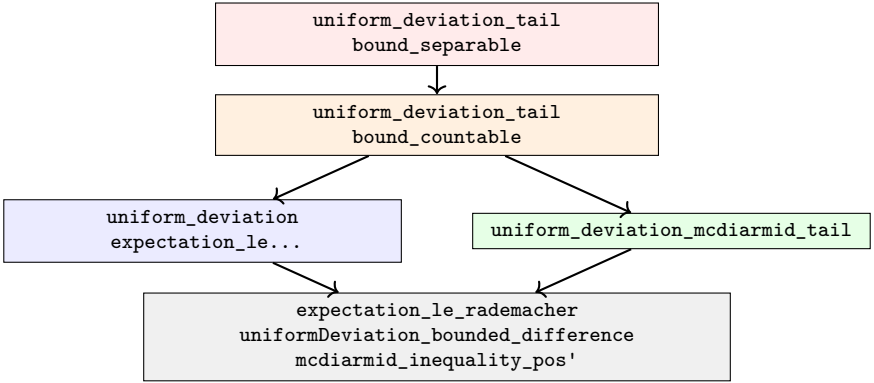
Mathlib API Forced by the Target

Paper object	Lean / Mathlib object	Why it appears
distribution	`Measure Omega`	population probability
probability law	`IsProbabilityMeasure mu`	total mass is one
sample of size n	`Fin n -> X`	exactly n coordinates
independent sample	`Measure.pi`	product distribution
population mean	measure integral	expectation of $f(X)$
sample mean	`Finset.sum`	finite average
worst model	`iSup`	uniform over the class

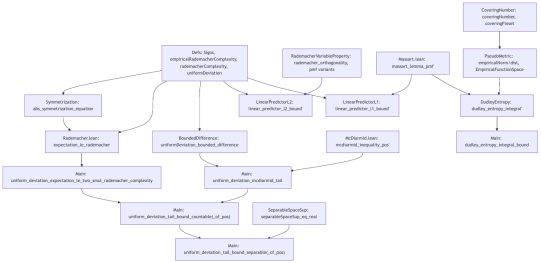
Mathlib API Forced by the Assumptions

Proof need	Mathlib structure	Where it is used
measurable event	`Measurable`	probability of the bad event
countable supremum	`Countable Iota`	measurable uniform deviation
bounded change	`Function.update`	replace one sample point
dense subclass	`SeparableSpace`	reduce to a countable sequence
chosen dense points	`denseSeq`	build the countable family
explicit hidden inputs	`@definition`	inspect all parameters

Read the Main API from the Goal Downward



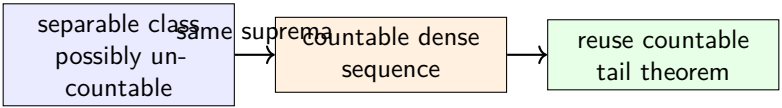
The Authors' Full Dependency Graph



This graph is useful for implementation. For project design, read it from the final theorem upward toward the missing lemmas and definitions.

Why the Separable Step Comes Last

The countable theorem is easier because countable suprema have a direct measurability path.



Three bridge lemmas show that both Rademacher complexities and uniform deviation are unchanged on the dense sequence.

Exercise: Design from the Theorem

Suppose the new target is:

$$\Pr(|\hat{L}_S(A(S)) - L(A(S))| > B(n, \delta)) \leq \delta.$$

Draw before writing Lean:

1. Why must the bound cover every possible $A(S)$?

Exercise: Design from the Theorem

Suppose the new target is:

$$\Pr(|\hat{L}_S(A(S)) - L(A(S))| > B(n, \delta)) \leq \delta.$$

Draw before writing Lean:

1. Why must the bound cover every possible $A(S)$?
2. What uniform quantity will you define?

Exercise: Design from the Theorem

Suppose the new target is:

$$\Pr(|\hat{L}_S(A(S)) - L(A(S))| > B(n,\delta)) \le \delta.$$

Draw before writing Lean:

1. Why must the bound cover every possible $A(S)$?
2. What uniform quantity will you define?
3. What gives its expected bound?

Exercise: Design from the Theorem

Suppose the new target is:

$$\Pr(|\hat{L}_S(A(S)) - L(A(S))| > B(n,\delta)) \le \delta.$$

Draw before writing Lean:

1. Why must the bound cover every possible $A(S)$?
2. What uniform quantity will you define?
3. What gives its expected bound?
4. What gives its probability tail?

Exercise: Design from the Theorem

Suppose the new target is:

$$\Pr(|\hat{L}_S(A(S)) - L(A(S))| > B(n,\delta)) \le \delta.$$

Draw before writing Lean:

1. Why must the bound cover every possible $A(S)$?
2. What uniform quantity will you define?
3. What gives its expected bound?
4. What gives its probability tail?
5. Which Mathlib objects are needed to state the event?

Project: Write Signatures Before Proofs

Produce a one-page design document with:

- the final theorem in ordinary mathematics;

Only after peer review should the project begin filling tactic proofs.

Project: Write Signatures Before Proofs

Produce a one-page design document with:

- the final theorem in ordinary mathematics;
- its Lean signature with explicit parameters;

Only after peer review should the project begin filling tactic proofs.

Project: Write Signatures Before Proofs

Produce a one-page design document with:

- the final theorem in ordinary mathematics;
- its Lean signature with explicit parameters;
- a two-branch proof tree;

Only after peer review should the project begin filling tactic proofs.

Project: Write Signatures Before Proofs

Produce a one-page design document with:

- the final theorem in ordinary mathematics;
- its Lean signature with explicit parameters;
- a two-branch proof tree;
- one new definition forced by each branch;

Only after peer review should the project begin filling tactic proofs.

Project: Write Signatures Before Proofs

Produce a one-page design document with:

- the final theorem in ordinary mathematics;
- its Lean signature with explicit parameters;
- a two-branch proof tree;
- one new definition forced by each branch;
- a table of reused Mathlib APIs;

Only after peer review should the project begin filling tactic proofs.

Project: Write Signatures Before Proofs

Produce a one-page design document with:

- the final theorem in ordinary mathematics;
- its Lean signature with explicit parameters;
- a two-branch proof tree;
- one new definition forced by each branch;
- a table of reused Mathlib APIs;
- one small finite example that checks the definitions.

Only after peer review should the project begin filling tactic proofs.

Subsection Outline

Why Verify a Compiler?

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A formally verified compiler back-end

Xavier Leroy

Received: 21 July 2009 / Accepted: 22 October 2009

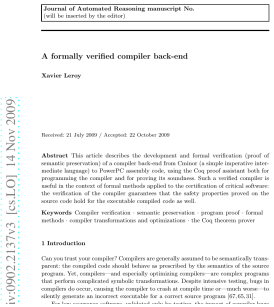
Abstract This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the verification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Keywords Compiler verification · semantic preservation · program proof · formal methods · compiler transformations and optimizations · the Coq theorem prover

1 Introduction

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compiler to crash at compile time or—much worse—to silently generate an incorrect executable for a correct source program [67,45,31].

Why Verify a Compiler?



Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A formally verified compiler back-end

Xavier Leroy

Received: 21 July 2009 / Accepted: 22 October 2009

Abstract This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the certification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Keywords Compiler verification · semantic preservation · program proof · formal methods · compiler transformations and optimizations · the Coq theorem prover

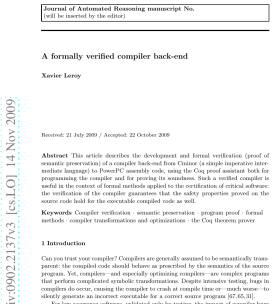
1 Introduction

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compiler to crash at compile time or—much worse—to silently generate an incorrect executable for a correct source program [67,45,31].

A compiler may silently change program behavior.

Leroy, *A formally verified compiler back-end*

Why Verify a Compiler?



Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A formally verified compiler back-end

Xavier Leroy

Received: 21 July 2009 / Accepted: 22 October 2009

Abstract This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the verification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Keywords Compiler verification · semantic preservation · program proof · formal methods · compiler transformations and optimizations · the Coq theorem prover

1 Introduction

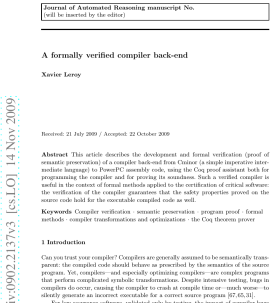
Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compiler to crash at compile time or—much worse—to silently generate an incorrect executable for a correct source program [67,45,31].

A compiler may silently change program behavior.

- Testing checks selected source programs.

Leroy, *A formally verified compiler back-end*

Why Verify a Compiler?



Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A formally verified compiler back-end

Xavier Leroy

Received: 21 July 2009 / Accepted: 22 October 2009

Abstract This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the certification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Keywords Compiler verification · semantic preservation · program proof · formal methods · compiler transformations and optimizations · the Coq theorem prover

1 Introduction

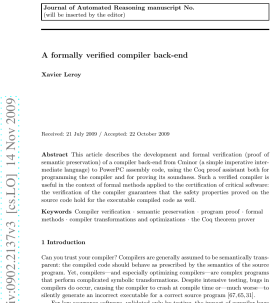
Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compiler to crash at compile time or—much worse—to silently generate an incorrect executable for a correct source program [37,45,31].

A compiler may silently change program behavior.

- Testing checks selected source programs.
- Optimization has many interacting cases.

Leroy, *A formally verified compiler back-end*

Why Verify a Compiler?



Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A formally verified compiler back-end

Xavier Leroy

Received: 21 July 2009 / Accepted: 22 October 2009

Abstract This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the certification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Keywords Compiler verification · semantic preservation · program proof · formal methods · compiler transformations and optimizations · the Coq theorem prover

1 Introduction

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compiler to crash at compile time or—much worse—to silently generate an incorrect executable for a correct source program [37,45,31].

For low-assurance software, validated only by testing, the impact of compiler bugs

A compiler may silently change program behavior.

- Testing checks selected source programs.
- Optimization has many interacting cases.
- Verification states a relation between all source and target runs.

Leroy, *A formally verified compiler back-end*

Motivation: One Mini-C Program

```
int main(void) {
  int x = 2;
  x = x + 1;
  return x;
}
```

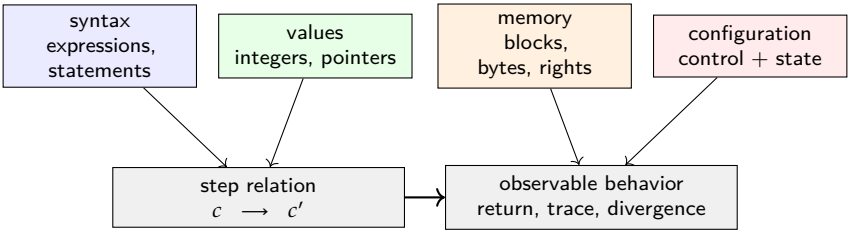
Running this example and seeing 3 is a test. A verification project must first define what assignment, memory, sequence, and return mean for every supported program.

Direct Translation: Execution as States

Control	Local environment	Memory	Observable result
`x = 2; ...`	no `x` yet	allocated frame	none
`x = x + 1; ...`	`x` names a location	location stores 2	none
`return x`	same location	location stores 3	none
finished	frame released	final memory	return 3

Each row is a configuration; each change is one or more semantic steps.

What Must Be Formalized for Mini-C?



Memory Cannot Be Left as "A Map"

In a C-like language, memory semantics may need:

- allocated blocks and valid offsets;

A memory-safety theorem is meaningful only after these choices are explicit.

Memory Cannot Be Left as "A Map"

In a C-like language, memory semantics may need:

- allocated blocks and valid offsets;
- bytes stored at addresses;

A memory-safety theorem is meaningful only after these choices are explicit.

Memory Cannot Be Left as "A Map"

In a C-like language, memory semantics may need:

- allocated blocks and valid offsets;
- bytes stored at addresses;
- permissions for reading and writing;

A memory-safety theorem is meaningful only after these choices are explicit.

Memory Cannot Be Left as "A Map"

In a C-like language, memory semantics may need:

- allocated blocks and valid offsets;
- bytes stored at addresses;
- permissions for reading and writing;
- pointer arithmetic and alignment;

A memory-safety theorem is meaningful only after these choices are explicit.

Memory Cannot Be Left as "A Map"

In a C-like language, memory semantics may need:

- allocated blocks and valid offsets;
- bytes stored at addresses;
- permissions for reading and writing;
- pointer arithmetic and alignment;
- undefined or unspecified behavior chosen for the model.

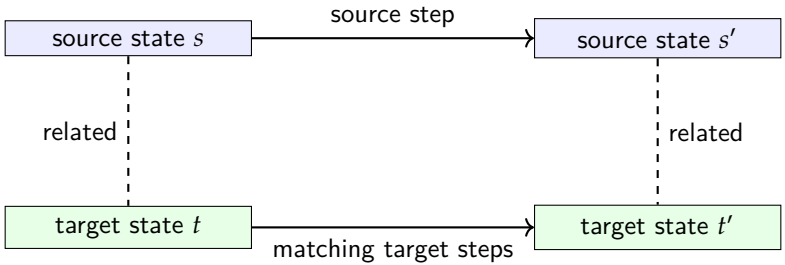
A memory-safety theorem is meaningful only after these choices are explicit.

Choose the Property After the Model

Wanted claim	Formal property	Extra definitions
one next state	determinism	transition relation
never stuck	progress / safety	final and stuck states
valid loads/stores	memory safety	permissions and validity
function meets contract	functional correctness	pre/postcondition
compiler keeps behavior	semantic preservation	source/target behavior

Different goals require different formal systems. “Verify C” is not yet a theorem.

Compiler Correctness as a Square



A simulation says the target can match source behavior while the relation is preserved.

Begin with a Tiny Expression Compiler

Mini-C is too large for a first proof. Keep only literals and addition.

```
inductive SrcExpr where
  | lit (value : Nat)
  | add (left right : SrcExpr)

inductive Instr where
  | push (value : Nat)
  | add

def evalSrc : SrcExpr -> Nat
  | .lit value => value
  | .add left right => evalSrc left + evalSrc right
```

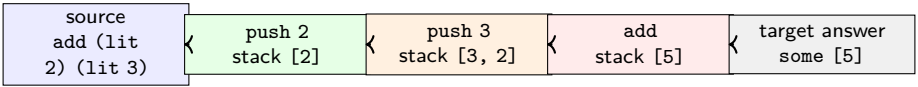
This still has the central compiler question: does target execution preserve the source answer?

Define the Target Machine and Compiler

```
inductive Instr where | push (value : Nat) | add

def exec : List Instr -> List Nat -> Option (List Nat)
  | [], stack => some stack
  | .push n :: code, stack => exec code (n :: stack)
  | .add :: code, right :: left :: stack =>
      exec code ((left + right) :: stack)
  | .add :: _code, _stack => none

def compile : SrcExpr -> List Instr
  | .lit value => [.push value]
  | .add left right =>
      compile left ++ compile right ++ [.add]
```

`Option` records stack failure. Compiled code will never take that failure path when it begins with a valid stack.

Trace the Program 2 + 3



This is one test. The theorem must cover every expression and every starting stack.

State the Exact Preservation Theorem

```
theorem compile_correct
    (expression : SrcExpr) (stack : List Nat) :
    exec (compile expression) stack =
      some (evalSrc expression :: stack)
```

Read both sides before proving anything:

- left: compile, then execute on an arbitrary old stack;

State the Exact Preservation Theorem

```
theorem compile_correct
    (expression : SrcExpr) (stack : List Nat) :
    exec (compile expression) stack =
      some (evalSrc expression :: stack)
```

Read both sides before proving anything:

- left: compile, then execute on an arbitrary old stack;
- right: evaluate in the source and push the same answer;

State the Exact Preservation Theorem

```
theorem compile_correct
    (expression : SrcExpr) (stack : List Nat) :
    exec (compile expression) stack =
      some (evalSrc expression :: stack)
```

Read both sides before proving anything:

- left: compile, then execute on an arbitrary old stack;
- right: evaluate in the source and push the same answer;
- the old stack is preserved below the new answer;

State the Exact Preservation Theorem

```
theorem compile_correct
    (expression : SrcExpr) (stack : List Nat) :
    exec (compile expression) stack =
      some (evalSrc expression :: stack)
```

Read both sides before proving anything:

- left: compile, then execute on an arbitrary old stack;
- right: evaluate in the source and push the same answer;
- the old stack is preserved below the new answer;
- `some` also says compiled code does not fail.

The Literal Case Computes by Definition

For `expression = lit value`, the goal becomes:

```
exec (compile (.lit value)) stack =
  some (evalSrc (.lit value) :: stack)

-- unfold compile, exec, and evalSrc:
some (value :: stack) = some (value :: stack)
```

Therefore the first induction case is `rfl`. No search is needed.

Why the Addition Case Needs One Helper

Compilation joins code with ++, but execution reads one instruction at a time. We first connect these two views.

```
theorem exec_append
    (first second : List Instr) (stack : List Nat) :
    exec (first ++ second) stack =
      (exec first stack).bind (exec second)
```

Prove it by induction on `first`. Then the addition case can run the left compiled code, run the right compiled code, and finally execute `Instr.add`.

The Addition Proof Follows the Compiler

```
theorem compile_correct
    (expression : SrcExpr) (stack : List Nat) :
    exec (compile expression) stack =
      some (evalSrc expression :: stack) := by
  induction expression generalizing stack with
  | lit value => rfl
  | add left right leftIH rightIH =>
      simp [compile, exec_append,
        leftIH, rightIH, exec, evalSrc]
```

1. exec_append separates the three code parts.

Live Lean: code/D6_02_FormalizationVerification_Exercises.lean

The Addition Proof Follows the Compiler

```
theorem compile_correct
    (expression : SrcExpr) (stack : List Nat) :
    exec (compile expression) stack =
      some (evalSrc expression :: stack) := by
  induction expression generalizing stack with
  | lit value => rfl
  | add left right leftIH rightIH =>
      simp [compile, exec_append,
        leftIH, rightIH, exec, evalSrc]
```

1. `exec_append` separates the three code parts.
2. `leftIH` and `rightIH` replace recursive runs by answers.

Live Lean: `code/D6_02_FormalizationVerification_Exercises.lean`

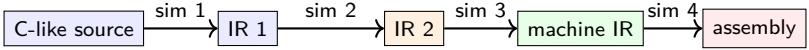
The Addition Proof Follows the Compiler

```
theorem compile_correct
    (expression : SrcExpr) (stack : List Nat) :
    exec (compile expression) stack =
      some (evalSrc expression :: stack) := by
  induction expression generalizing stack with
  | lit value => rfl
  | add left right leftIH rightIH =>
      simp [compile, exec_append,
        leftIH, rightIH, exec, evalSrc]
```

1. `exec_append` separates the three code parts.
2. `leftIH` and `rightIH` replace recursive runs by answers.
3. `simp` computes the final stack instruction.

Live Lean: `code/D6_02_FormalizationVerification_Exercises.lean`

A Verified Compiler Has Many Small Squares



Join the pass simulations to get an end-to-end semantic-preservation theorem.

What the Final Theorem Can Say

A typical compiler theorem has this direction:

$$\text{target behavior of } \texttt{compile}(p) \implies \text{allowed source behavior of } p.$$

- It does not prove that every source program is safe.

What the Final Theorem Can Say

A typical compiler theorem has this direction:

$$\text{target behavior of } \texttt{compile}(p) \implies \text{allowed source behavior of } p.$$

- It does not prove that every source program is safe.
- It says the verified compiler does not invent disallowed behavior.

What the Final Theorem Can Say

A typical compiler theorem has this direction:

$$\text{target behavior of } \texttt{compile}(p) \implies \text{allowed source behavior of } p.$$

- It does not prove that every source program is safe.
- It says the verified compiler does not invent disallowed behavior.
- Front end, linker, runtime, hardware, and undefined behavior remain inside or outside the trust boundary according to the model.

Easy Compiler Checks

Do not begin by reproving the induction theorem. Check two small cases first.

```
theorem compile_literal (value : Nat) (stack : List Nat) :
    exec (compile (.lit value)) stack =
      some (value :: stack) := by
  sorry

theorem compile_two_plus_three :
    exec (compile (.add (.lit 2) (.lit 3))) [] =
      some [5] := by
  sorry
```

Both solutions are `rfl`. The general proof is provided in the same standalone file so that students can read it after these checks.

Live Lean: `code/D6_02_FormalizationVerification_Exercises.lean`

Project: A Small Verifiable C Fragment

First milestone:

- integers, variables, addition, assignment, sequence, and return;

Loops, pointers, functions, and optimization are later milestones.

Project: A Small Verifiable C Fragment

First milestone:

- integers, variables, addition, assignment, sequence, and return;
- an environment and a simple memory model;

Loops, pointers, functions, and optimization are later milestones.

Project: A Small Verifiable C Fragment

First milestone:

- integers, variables, addition, assignment, sequence, and return;
- an environment and a simple memory model;
- configurations and one-step semantics;

Loops, pointers, functions, and optimization are later milestones.

Project: A Small Verifiable C Fragment

First milestone:

- integers, variables, addition, assignment, sequence, and return;
- an environment and a simple memory model;
- configurations and one-step semantics;
- the complete trace of the `x = 2` example;

Loops, pointers, functions, and optimization are later milestones.

Project: A Small Verifiable C Fragment

First milestone:

- integers, variables, addition, assignment, sequence, and return;
- an environment and a simple memory model;
- configurations and one-step semantics;
- the complete trace of the `x = 2` example;
- one theorem: determinism or preservation by one compiler pass.

Loops, pointers, functions, and optimization are later milestones.

Subsection Outline

Motivation: Start with a Finite Table

A student may understand this probability table before measure theory:

Outcome	Head	Tail
Mass	$1/2$	$1/2$

We can first define finite mass, total mass, expectation, and entropy. But a final project should explain how this local model connects to Mathlib probability.

Two Mathematical Formalization Patterns

Mathlib-first

Two Mathematical Formalization Patterns

Mathlib-first

- use `Measure`, integrals, topology;

Two Mathematical Formalization Patterns

Mathlib-first

- use `Measure`, integrals, topology;
- define only missing domain ideas;

Two Mathematical Formalization Patterns

Mathlib-first

- use `Measure`, integrals, topology;
- define only missing domain ideas;
- Sonoda follows this route.

Two Mathematical Formalization Patterns

Mathlib-first

- use `Measure`, integrals, topology;
- define only missing domain ideas;
- Sonoda follows this route.

Local theory first

Two Mathematical Formalization Patterns

Mathlib-first

- use `Measure`, integrals, topology;
- define only missing domain ideas;
- Sonoda follows this route.

Local theory first

- define small types and laws;

Two Mathematical Formalization Patterns

Mathlib-first

- use `Measure`, integrals, topology;
- define only missing domain ideas;
- Sonoda follows this route.

Local theory first

- define small types and laws;
- prove the core argument locally;

Two Mathematical Formalization Patterns

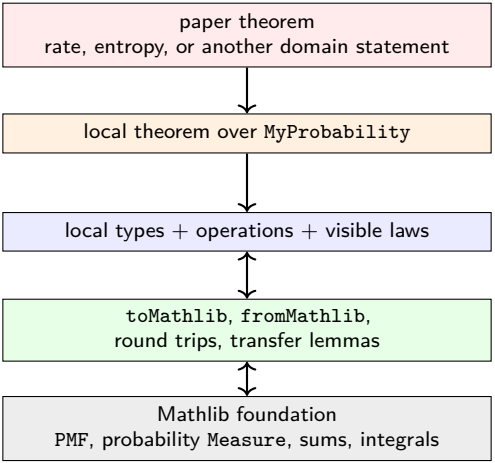
Mathlib-first

- use `Measure`, integrals, topology;
- define only missing domain ideas;
- Sonoda follows this route.

Local theory first

- define small types and laws;
- prove the core argument locally;
- build a checked Mathlib bridge.

The Complete Bridge



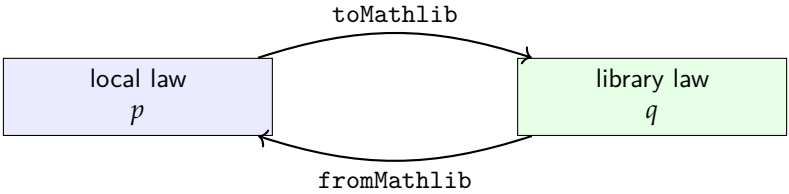
Direct Formal Translation: Two Representations

```
@[ext] structure FiniteLaw (Outcome : Type) where
  mass : Outcome -> Nat
  total : Nat
  positive : 0 < total

@[ext] structure LibraryLaw (Outcome : Type) where
  mass : Outcome -> Nat
  total : Nat
  positive : 0 < total
```

`LibraryLaw` is only a runnable stand-in. A real bridge should target a standard Mathlib `PMF` or probability measure.

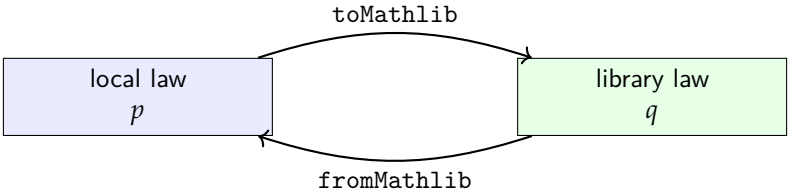
Live Lean: `code/D6_02_FormalizationVerification.lean`

Four Proofs Make the Bridge Trustworthy



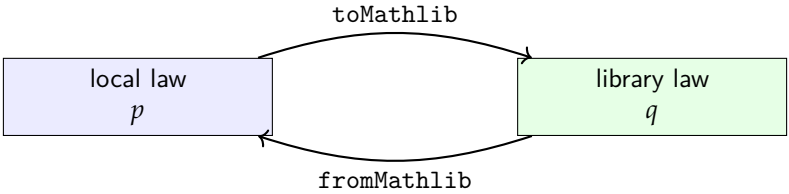
1. `fromMathlib (toMathlib p) = p;`

Four Proofs Make the Bridge Trustworthy



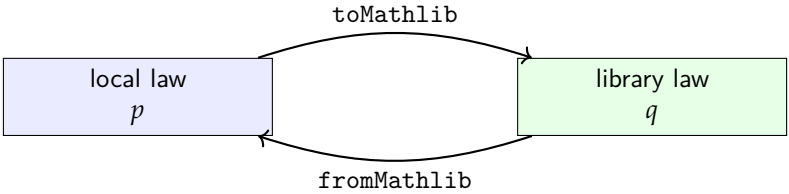
1. `fromMathlib (toMathlib p) = p;`
2. `toMathlib (fromMathlib q) = q;`

Four Proofs Make the Bridge Trustworthy



1. `fromMathlib (toMathlib p) = p;`
2. `toMathlib (fromMathlib q) = q;`
3. local operations are preserved by conversion;

Four Proofs Make the Bridge Trustworthy



1. `fromMathlib (toMathlib p) = p;`
2. `toMathlib (fromMathlib q) = q;`
3. local operations are preserved by conversion;
4. local theorems transfer to the library representation.

Exercise: Prove One Round Trip

```
theorem bridge_round_trip
    (Outcome : Type) (law : FiniteLaw Outcome) :
    (finiteLawEquiv Outcome).symm
      (finiteLawEquiv Outcome law) = law := by
  sorry
```

Live Lean: code/D6_02_FormalizationVerification_Exercises.lean

Exercise: Prove One Round Trip

```
theorem bridge_round_trip
    (Outcome : Type) (law : FiniteLaw Outcome) :
    (finiteLawEquiv Outcome).symm
      (finiteLawEquiv Outcome law) = law := by
  sorry
```

The local equivalence stores this as `left_inv`. In a Mathlib development, package the real maps with the standard `Equiv`.

Live Lean: `code/D6_02_FormalizationVerification_Exercises.lean`

What an Unfinished Bridge Can Prove

The `Discreted-Rate-Distortion` repository is a useful design case study.

- It begins with a small discrete probability layer.

What an Unfinished Bridge Can Prove

The `Discreted-Rate-Distortion` repository is a useful design case study.

- It begins with a small discrete probability layer.
- It builds expectation, entropy, and rate-distortion ideas above it.

What an Unfinished Bridge Can Prove

The `Discreted-Rate-Distortion` repository is a useful design case study.

- It begins with a small discrete probability layer.
- It builds expectation, entropy, and rate-distortion ideas above it.
- Some laws are still axioms and the final development has proof gaps.

What an Unfinished Bridge Can Prove

The `Discreted-Rate-Distortion` repository is a useful design case study.

- It begins with a small discrete probability layer.
- It builds expectation, entropy, and rate-distortion ideas above it.
- Some laws are still axioms and the final development has proof gaps.
- Therefore it shows a project design, not a completed theorem.

When No Large Library Exists

A project may need to define its own types and axioms first.

- Make every assumption visible in structures or theorem parameters.

The kernel checks the proof from the written assumptions; peers still check whether the model is the right one.

When No Large Library Exists

A project may need to define its own types and axioms first.

- Make every assumption visible in structures or theorem parameters.
- Ask peers whether the assumptions describe the intended subject.

The kernel checks the proof from the written assumptions; peers still check whether the model is the right one.

When No Large Library Exists

A project may need to define its own types and axioms first.

- Make every assumption visible in structures or theorem parameters.
- Ask peers whether the assumptions describe the intended subject.
- Separate “proved from these axioms” from “proved about the real model.”

The kernel checks the proof from the written assumptions; peers still check whether the model is the right one.

When No Large Library Exists

A project may need to define its own types and axioms first.

- Make every assumption visible in structures or theorem parameters.
- Ask peers whether the assumptions describe the intended subject.
- Separate “proved from these axioms” from “proved about the real model.”
- Later provide an interpretation, equivalence, or consistency argument.

The kernel checks the proof from the written assumptions; peers still check whether the model is the right one.

Project: Draw Your Own Mathlib Bridge

Before proving the main theorem, draw five layers:

1. informal paper theorem;

Mark each unproved axiom in red and each proved bridge in green.

Project: Draw Your Own Mathlib Bridge

Before proving the main theorem, draw five layers:

1. informal paper theorem;
2. local theorem signature;

Mark each unproved axiom in red and each proved bridge in green.

Project: Draw Your Own Mathlib Bridge

Before proving the main theorem, draw five layers:

1. informal paper theorem;
2. local theorem signature;
3. local definitions and assumptions;

Mark each unproved axiom in red and each proved bridge in green.

Project: Draw Your Own Mathlib Bridge

Before proving the main theorem, draw five layers:

1. informal paper theorem;
2. local theorem signature;
3. local definitions and assumptions;
4. conversion and round-trip theorems;

Mark each unproved axiom in red and each proved bridge in green.

Project: Draw Your Own Mathlib Bridge

Before proving the main theorem, draw five layers:

1. informal paper theorem;
2. local theorem signature;
3. local definitions and assumptions;
4. conversion and round-trip theorems;
5. Mathlib objects and reused results.

Mark each unproved axiom in red and each proved bridge in green.