Programming and Proving in Lean

David Thrane Christiansen
Lean Focused Research Organization, LLC

November 28, 2023

LEAN

Interactive
theorem prover

Interactive theorem prover

Programming language

Goals

After these three sessions, you'll be able to:

- Get started using Lean for programming and/or proving
- Contextualize Lean in the landscape of related systems
- Know where to look for more information
- Have an idea of whether Lean is relevant for your work

Background Assumptions

Functional programming

Monads

Informal proofs

Background Assumptions

Functional programming

Monads

Informal proofs

You don’t need to be a type theory expert!

About Me

- PhD, ITU, 2015 (advised by Peter Sestoft)
- Second-most commits on Idris 1
- Postdoc, Indiana University, 2016–2017
- Industrial experience at Galois, Deon Digital
- ED of Haskell Foundation, 2022–2023
- Author:
  - *The Little Typer* (with Dan Friedman), 2018, MIT Press
  - *Functional Programming in Lean*, 2023, Microsoft Research (free online)
- Working full-time on Lean at the FRO

The Lean FRO

The Lean FRO

The Lean FRO

Self-
Sustainability

The Lean FRO is made possible by the generous philanthropic support of the Simons Foundation International, the Alfred P. Sloan Foundation, and Richard Merkin, along with operational support and stewardship by Convergent Research.

Outline

Overview - 14/11

Programming and Metaprogramming - 21/11

Foundations and Proofs - 28/11

Outline

Overview - 14/11
- Syntax
- UI
- Programs and Proofs
- Type Classes
- Monads
- Do-Notation
- Dependent Types

Programming and Metaprogramming - 21/11

Foundations and Proofs - 28/11

Outline

Overview - 14/11
- Syntax
- UI
- Programs and Proofs
- Type Classes
- Monads
- Do-Notation
- Dependent Types

Demo: Demo!

SyntaxIntro.lean

Field Notation

```
#eval List.length [14, 11, 23]
-- 3

#eval [14, 11, 23].length
-- 3

#eval List.map (fun x => x + 1) [1, 2, 3]
-- [2, 3, 4]

#eval [1, 2, 3].map (fun x => x + 1)
-- [2, 3, 4]
```

Field Notation

```
#eval List.length [14, 11, 23]
-- 3

#eval [14, 11, 23].length
-- 3
```

Type of argument
before dot: `List Nat`

```
=> x + 1) [1, 2, 3]

#eval [1, 2, 3].map (fun x => x + 1)
-- [2, 3, 4]
```

Field Notation

```
#eval List.length [14, 11, 23]
-- 3
```

```
#eval [14, 11, 23].length
-- 3
```

This call becomes `List.length`

Type of argument before dot: `List Nat`

```
=> x + 1) [1, 2, 3]
```

```
#eval [1, 2, 3].map (fun x => x + 1)
-- [2, 3, 4]
```

Field Notation

```
#eval List.length [14, 11, 23]
-- 3

#eval [14, 11, 23].length
-- 3

#eval List.map (fun x => x + 1) [1, 2, 3]
-- [2, 3, 4]

#eval [1, 2, 3].map (fun x => x + 1)
-- [2, 3, 4]
```

Field Notation

```
#eval List.length [14, 11, 23]
-- 3

#eval [14, 11, 23].length
-- 3

#eval List.map (fun x => x + 1) [1, 2, 3]
-- [2, 3, 4]

#eval [1, 2, 3].map (fun x => x + 1)
-- [2, 3, 4]
```

Type of argument before dot: `List Nat`

Field Notation

```
#eval List.length [14, 11, 23]
-- 3

#eval [14, 11, 23].length
-- 3

#eval List.map (fun x => x + 1) [1, 2, 3]
-- [2, 3, 4]

#eval [1, 2, 3].map (fun
-- [2, 3, 4]
```

This call becomes `List.map`

Type of argument before dot: `List Nat`

Field Notation

```
#eval List.length [14, 11, 23]
-- 3

#eval [14, 11, 23].length
-- 3

#eval List.map (fun x => x + 1) [1, 2, 3]
-- [2, 3, 4]

#eval [1, 2, 3].map (fun
-- [2, 3, 4]
```

This call becomes `List.map`

Argument before dot placed in first type-correct position

Outline

Overview - 14/11

- Syntax
- UI
- Programs and Proofs
- Type Classes
- Monads
- Do-Notation
- Dependent Types

The Infoview

Lean Infoview

▼Examples.lean:14:17

▼Expected type

```
α : Type u_1
x : α
xs : List α
⊢ List α
```

▼All Messages (2)

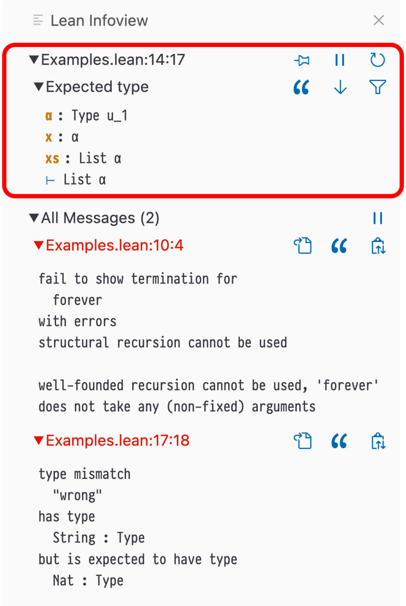
▼Examples.lean:10:4

```
fail to show termination for
  forever
with errors
structural recursion cannot be used

well-founded recursion cannot be used, 'forever'
does not take any (non-fixed) arguments
```

▼Examples.lean:17:18

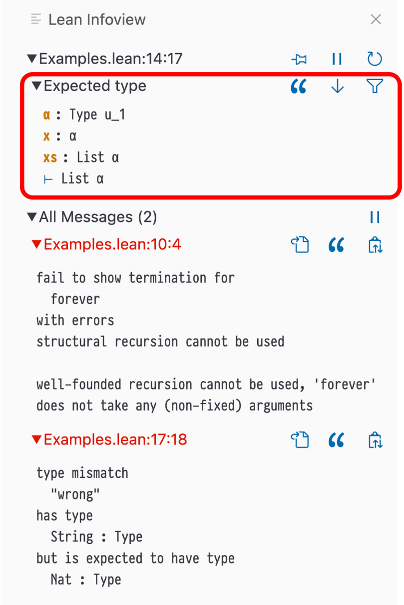
```
type mismatch
  "wrong"
has type
  String : Type
but is expected to have type
  Nat : Type
```

The Infoview



Information about cursor position

The Infoview



Local context and current type

The Infoview

Lean Infoview

▼Examples.lean:14:17

▼Expected type

```
α : Type u_1
x : α
xs : List α
⊢ List α
```

▼All Messages (2)

▼Examples.lean:10:4

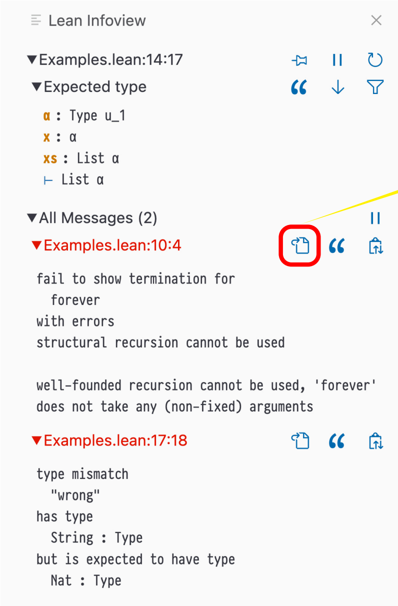
```
fail to show termination for
  forever
with errors
structural recursion cannot be used

well-founded recursion cannot be used, 'forever'
does not take any (non-fixed) arguments
```

▼Examples.lean:17:18

```
type mismatch
  "wrong"
has type
  String : Type
but is expected to have type
  Nat : Type
```

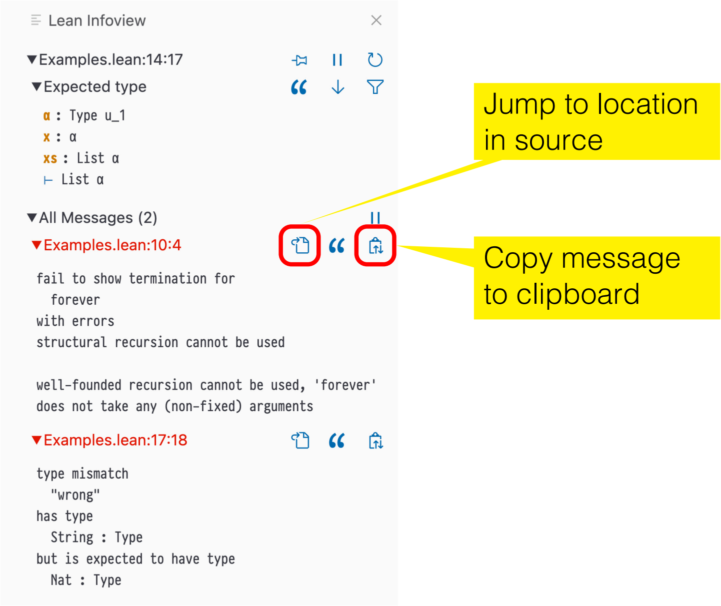
Other errors, warnings, and information

The Infoview

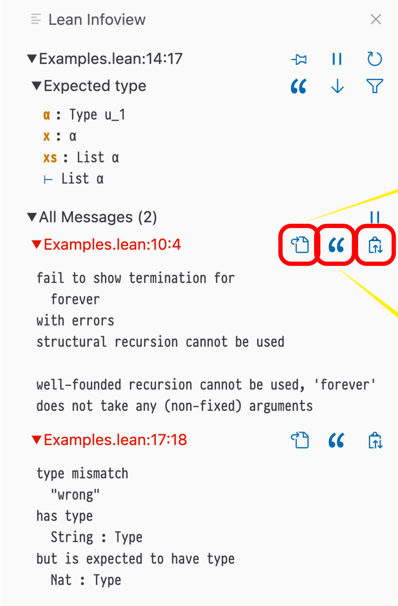


Jump to location in source

The Infoview



The Infoview



Jump to location in source

Copy message to clipboard

Insert message contents as comment in source

The Infoview

Lean Infoview

▼Examples.lean:14:17

▼Expected type

```
α : Type u_1
x : α
xs : List α
⊢ List α
```

▼All Messages (2)

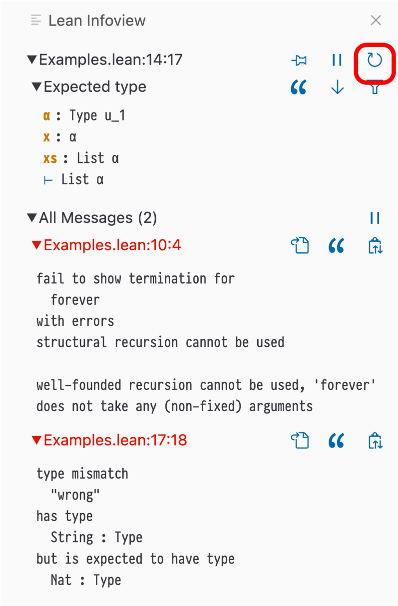
▼Examples.lean:10:4

```
fail to show termination for
  forever
with errors
structural recursion cannot be used

well-founded recursion cannot be used, 'forever'
does not take any (non-fixed) arguments
```

▼Examples.lean:17:18

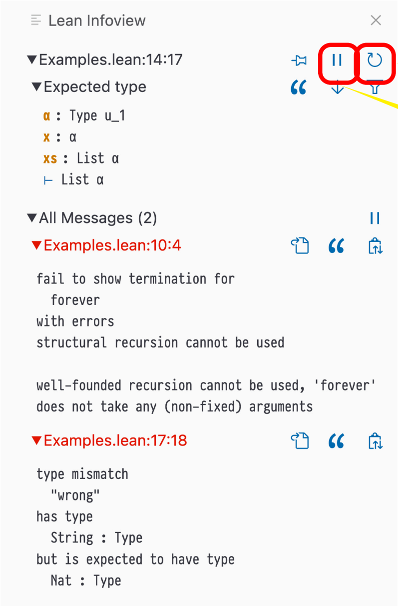
```
type mismatch
  "wrong"
has type
  String : Type
but is expected to have type
  Nat : Type
```

The Infoview



Manually refresh

The Infoview



Manually refresh

Stop automatically updating in response to file changes

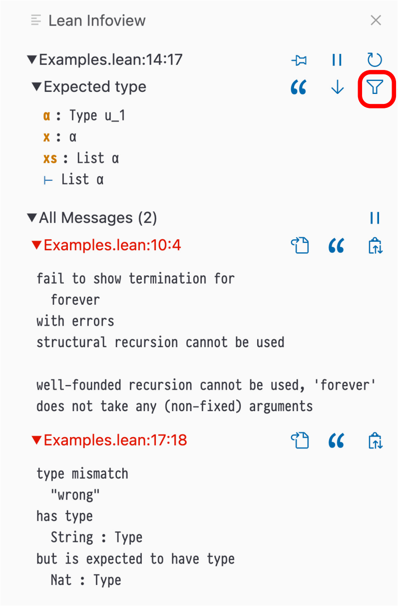
The Infoview

Manually refresh

Stop automatically updating in response to file changes

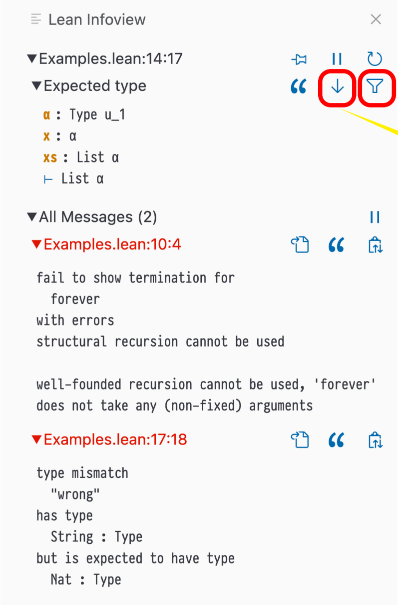
“Pin” this information, retaining it when the cursor is moved

The Infoview



Filter hypotheses (e.g. hiding types)

The Infoview



Filter hypotheses (e.g. hiding types)

Reverse the order of variables and expected type

Breadcrumbs

Namespaces.lean Namespaces.lean/{ } A/{ } B/{ } C/ swap

```
namespace A
namespace B
namespace C
```

⋮

```
def swap : α × β → β × α
  | (x, y) => (y, x)
```

⋮

```
end C
end B
end A
```

Breadcrumbs show you where you are — idiomatic Lean style assumes their presence and doesn't indent

Outline

Overview - 14/11

- Syntax
- UI
- Programs and Proofs
- Type Classes
- Monads
- Do-Notation
- Dependent Types

Programs and Proofs

```
def third (arr : Array Nat) : Nat := arr[2]
```

Programs and Proofs

```
def third (arr : Array Nat) : Nat := arr[2]
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is
    performed, and 'Panic' error message is produced if
    index is not valid
  - Use `a[i]?` notation instead, result is an `Option`
    type
  - Use `a[i]'h` notation instead, where `h` is a proof
    that index is valid
arr : Array Nat
⊢ 2 < Array.size arr
```

Programs and Proofs

```
def third (arr : Array Nat) : Nat := arr[2]
```

Lean requires a proof of bounds-safety by default

```
failed to prove index is valid
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is
    performed, and 'Panic' error message is produced if
    index is not valid
  - Use `a[i]?` notation instead, result is an `Option`
    type
  - Use `a[i]'h` notation instead, where `h` is a proof
    that index is valid
arr : Array Nat
⊢ 2 < Array.size arr
```

Propositions as Types

```
fun x => (x, x) : String → String × String

List.map toString : List Int → List String

[-1, 2, 5, -22] : List Int
```

Propositions as Types

```
fun x => (x, x) : String → String × String

List.map toString : List Int → List String

[-1, 2, 5, -22] : List Int

??? : 2 < arr.size

??? : 2 + 2 = 4

??? : ∀ xs, xs.reverse.reverse = xs

??? : arr.size > 2 → arr.size/2 ≥ 0
```

Evidence

$A \land B$	`And.intro : a → b → a ∧ b`	Evidence of both *A* and *B*

Evidence

$A \wedge B$	`And.intro : a → b → a ∧ b`	Evidence of both *A* and *B*
$A \vee B$	`Or.inl : a → a ∨ b` `Or.inr : b → a ∨ b`	Either evidence of *A* or evidence of *B*

Evidence

$A \land B$	`And.intro : a → b → a ∧ b`	Evidence of both A and B
$A \lor B$	`Or.inl : a → a ∨ b` `Or.inr : b → a ∨ b`	Either evidence of A or evidence of B
$A \to B$	`fun (h : A) => (... : B)`	Given A, produce evidence of B

Evidence

$A \wedge B$	`And.intro : a → b → a ∧ b`	Evidence of both *A* and *B*
$A \vee B$	`Or.inl : a → a ∨ b` `Or.inr : b → a ∨ b`	Either evidence of *A* or evidence of *B*
$A \to B$	`fun (h : A) => (... : B)`	Given *A*, produce evidence of *B*
`True`	`True.intro`	Trivial evidence

Evidence

$A \land B$	`And.intro : a → b → a ∧ b`	Evidence of both *A* and *B*
$A \lor B$	`Or.inl : a → a ∨ b` `Or.inr : b → a ∨ b`	Either evidence of *A* or evidence of *B*
$A \to B$	`fun (h : A) => (... : B)`	Given *A*, produce evidence of *B*
`True`	`True.intro`	Trivial evidence
`False`		No evidence at all!

Evidence

$A \land B$	`And.intro : a → b → a ∧ b`	Evidence of both *A* and *B*
$A \lor B$	`Or.inl : a → a ∨ b` `Or.inr : b → a ∨ b`	Either evidence of *A* or evidence of *B*
$A \to B$	`fun (h : A) => (... : B)`	Given *A*, produce evidence of *B*
`True`	`True.intro`	Trivial evidence
`False`		No evidence at all!
$\neg A$	`fun (h : A) =>` `(... : False)`	Given *A*, derive a contradiction

Evidence

`∀(x : A), P`	`fun (x : A) =>` `(... : P)`	Provide evidence of *P* for **any** given `x : A`

Evidence

`∀(x : A), P`	`fun (x : A) =>` `(... : P)`	Provide evidence of P for **any** given `x : A`

`∃(x : A), P`	`Exists.intro :`
	`(w : A) → P w →`
	`Exists A P`
		Some $w : A$ paired with evidence of $P[w/x]$

Evidence

`∀(x : A), P`	`fun (x : A) =>` `(... : P)`	Provide evidence of *P* for **any** given `x : A`

`∃(x : A), P`	`Exists.intro :`
	`(w : A) → P w →`
	`Exists A P`
		Some $w : A$ paired with evidence of $P[w/x]$

```
Classical.em  :  ∀{p : Prop}, p ∧ ¬ p

propext       :  ∀{p q : Prop}, p ↔ q → p = q
```

Tactics

Writing evidence by hand is slow and error-prone - *tactics* are programs to automate this process.

Demo: Evidence and Tactics

LEAN

`Evidence.lean`

More on proofs later!

Outline

Overview - 14/11

- Syntax
- UI
- Programs and Proofs
- Type Classes
- Monads
- Do-Notation
- Dependent Types

Type Classes

`42 + 23`   `3.4 + 19.33401`   `newYearsDay + fiveMinutes`

What does + mean here?

Type Classes

```
"Hello, " ++ "world!"

[1, 2, 3] ++ [4, 5, 6]

#[1, 2, 3] ++ [4, 5, 6]
```

What about ++ here?

Type Classes

```
structure Add (α : Type) where
  add : α → α → α

def addAnything (impl : Add α) (x y : α) : α :=
  Add.add impl x y

def implAddString : Add String where
  add str1 str2 := str1 ++ str2

#eval addAnything implAddString "Hello, " "world"
-- "Hello, world"
```

`implAddString` describes how to add strings

Type Classes

```
class Add (α : Type) where
  add : α → α → α

def addAnything [Add α] (x y : α) : α :=
  Add.add x y

instance : Add String where
  add str1 str2 := str1 ++ str2

#eval addAnything "Hello, " "world"
-- "Hello, world"
```

implementation found automatically by Lean

Demo: Demo!

`TypeClasses.lean`

Outline

Overview - 14/11

- Syntax
- UI
- Programs and Proofs
- Type Classes
- Monads
- Do-Notation
- Dependent Types

Monads in Lean

Monads capture repeated patterns under a type constructor.

- Data dependencies and ordering
- Passing some external data around
- Error recovery
- Much more!

Demo: Example Monads

`Monads.lean`

Outline

Overview - 14/11

- Syntax
- UI
- Programs and Proofs
- Type Classes
- Monads
- Do-Notation
- Dependent Types

Desugaring Do

`do` E_1	$\Longrightarrow$	E_1
`do` E_1 E_2	$\Longrightarrow$	`Monad.bind` E_1 (**`fun`** `() =>` E_2)
`do let` x `←` E_1 E_2	$\Longrightarrow$	`Monad.bind` E_1 (**`fun`** x `=>` E_2)

More Features

```
def sumArrayFrom [Add α]
    (start : α) (arr : Array α) : α := Id.run do
  let mut sum := start
  for x in arr do
    sum := sum + x
  return sum

def listProduct (xs : List Nat) : Nat := Id.run do
  let mut prod := 0
  for x in xs do
    if x == 0 then
      return 0
    prod := prod * x
  return prod
```

More Features

```
def sumArrayFrom [Add α]
    (start : α) (arr : Array α) : α := Id.run do
  let mut sum := start
  for x in arr do
    sum := sum + x
  return sum

def listProduct                       := Id.run do
  let mut prod := 0
  for x in xs do
    if x == 0 then
      return 0
    prod := prod * x
  return prod
```

Locally-mutable variables

More Features

```
def sumArrayFrom [Add α]
   (start : α) (arr : Array α) : α := Id.run do
  let mut sum := start
  for x in arr do
    sum := sum + x
  return sum

 def listProduct                      := Id.run do
  let mut prod := 0
  for x in xs do
    if x == 0 then
      return 0
    prod := prod * x
  return prod
```

Looping over data structures

More Features

```
def sumArrayFrom [Add α]
    (start : α) (arr : Array α) : α := Id.run do
  let mut sum := start
  for x in arr do
    sum := sum + x
  return sum

def listProduct                      := Id.run do
  let mut prod := 0
  for x in xs do
    if x == 0 then
      return 0
    prod := prod * x
  return prod
```

Elseless **if**

More Features

```
def sumArrayFrom [Add α]
    (start : α) (arr : Array α) : α := Id.run do
  let mut sum := start
  for x in arr do
    sum := sum + x
  return sum
```

```
def listProduct                   := Id.run do
  let mut prod := 0
  for x in xs do
    if x == 0 then
      return 0
    prod := prod * x
  return prod
```

Early return

No Magic

```
def sumArrayFrom' [Add α]
    (start : α) (arr : Array α) : α := Id.run do
  let mut sum := start
  arr.forM (fun x => do sum := sum + x)
  return sum
```

No Magic

```
def sumArrayFrom' [Add α]
    (start : α) (arr : Array α) : α := Id.run do
  let mut sum := start
  arr.forM (fun x => do sum := sum + x)
  return sum
```

Mutation within same **do**-block only

No Magic

```
def sumArrayFrom' [Add α]
      (start : α) (arr : Array α) : α := Id.run do
    let mut sum := start
    arr.forM (fun x => do sum := sum + x)
    return sum
```

Mutation within same **do**-block only

Loops	→	`forM`, with special encodings of **`break`** and **`continue`**
Mutable Variables	→	`StateT`
Early Return	→	`ExceptT α α`

Outline

Overview - 14/11

- Syntax
- UI
- Programs and Proofs
- Type Classes
- Monads
- Do-Notation
- Dependent Types

Dependent Types

`List String` — A list of Strings

Dependent Types

`List String` — A list of `String`s

`Vec String 5` — A list of five `String`s

Dependent Types

`List String`	—	A list of `Strings`
`Vec String 5`	—	A list of five `Strings`
`Fin 5`	—	A number less than 5

Demo: Demo!

Vec.lean

Next time

Programming and Metaprogramming in Lean

- The standard library
- Run-time representations and memory management
- Proving termination
- Macros and Metaprogramming

Reading for Today

Functional Programming in Lean chapters 1–6 (“Getting to know Lean” through “Functors, Applicative Functors, and Monads”)

Thank you!

Happy to answer questions! I'm usually here on Fridays.

- `david@lean-fro.org`
- `https://davidchristiansen.dk`

Documentation and tutorials at: `https://lean-lang.org`

Outline

Overview - 14/11

Programming and Metaprogramming - 21/11
- Run-Time Representations and Memory Management
- Standard Library
- Proving Termination
- Notations, Macros and Metaprogramming

Foundations and Proofs - 28/11

Outline

Programming and Metaprogramming - 21/11
- Run-Time Representations and Memory Management
- Standard Library
- Proving Termination
- Notations, Macros and Metaprogramming

Run-Time Representations and Memory Management

Lean's cost model:

- Simple, predictable memory layout
- Overrides for performance-sensitive cases
- Memory management via reference counting
- Opportunistic mutation

Reference Counting vs Tracing GC

Tracing GC

- Accurately collect cycles
- Pause on allocation
- Requires global notion of roots
- Cheap allocation with "bump pointer"
- Complex implementation

Reference Counting

- Fails for cyclic data
- Pause on deallocation
- Local notion of roots
- `malloc`-like allocation
- Simple implementation

Reference Counting vs Tracing GC

Reference Counting

- Fails for cyclic data
- Pause on deallocation
- Local notion of roots
- `malloc`-like allocation
- Simple implementation

Reference Counting vs Tracing GC

Reference Counting

- **Fails for cyclic data**
- Pause on deallocation
- Local notion of roots
- `malloc`-like allocation
- Simple implementation

Reference Counting vs Tracing GC

Reference Counting

- ~~Fails for cyclic data~~
- Pause on deallocation
- Local notion of roots
- `malloc`-like allocation
- Simple implementation

Reference Counting vs Tracing GC

Reference Counting

- ~~Fails for cyclic data~~
- ~~Pause on deallocation~~
- Local notion of roots
- `malloc`-like allocation
- Simple implementation

Reference Counting vs Tracing GC

Reference Counting

- ~~Fails for cyclic data~~
- ~~Pause on deallocation~~
- Local notion of roots
- `malloc`-like allocation
- Simple implementation*

* Ignoring needed compiler optimizations

Reference Counting and Mutation

```
def List.map (f : α → β) : List α → List β
  | .nil => .nil
  | .cons x xs => .cons (f x) (map f xs)
```

Reference Counting and Mutation

```
def List.map (f : α → β) : List α → List β
  | .nil => .nil
  | .cons x xs => .cons (f x) (map f xs)
```

Decrementing reference count

Reference Counting and Mutation

```
def List.map (f : α → β) : List α → List β
  | .nil => .nil
  | .cons x xs => .cons (f x) (map f xs)
```

Decrementing reference count

Allocating a cons cell

Reference Counting and Mutation

```
def List.map (f : α → β) : List α → List β
  | .nil => .nil
  | .cons x xs => .cons (f x) (map f xs)
```

Decrementing reference count

Reuse the input when RC=0

Reference Counting and Mutation

```
def List.map (f : α → β) : List α → List β
  | .nil => .nil
  | .cons x xs => .cons (f x) (map f xs)
```

`List.map` opportunistically mutates the non-shared prefix of its argument, with no extra programmer work

Reference Counting: Consequences

- Good performance - competitive with OCaml
- Textbook algorithms require modifications to ensure memory reuse
- Ensuring linear use of data is important, but also nonlocal and noncompositional

Reference Counting: Consequences

- Good performance - competitive with OCaml
- Textbook algorithms require modifications to ensure memory reuse
- Ensuring linear use of data is important, but also nonlocal and noncompositional

Further reading:

Counting Immutable Beans, Ullrich and de Moura (IFL ’19)

Perceus: Garbage Free Reference Counting with Reuse, Reinking, Xie, de Moura and Leijen (PLDI ’21)

Memory Layout

1. Erase all types
2. Erase all proofs
3. Argumentless constructors become constants
4. Do the “`newtype`” trick

Values are typically pointers to a 64-bit header and the remaining data

Memory Layout: List

```
inductive List (α : Type u) : Type u where
  | nil
  | cons : α → List α → List α
```

Memory Layout: `List`

```
inductive List (α : Type u) : Type u where
  | nil
  | cons : α → List α → List α
```

with implicit arguments...

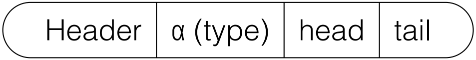
```
inductive List.{u} (α : Type u) : Type u where
  | nil.{u} : {α : Type u} → List α
  | cons.{u} : {α : Type u} → α → List α → List α
```

Memory Layout: `List.cons`

```
cons.{u} : {α : Type u} → α → List α → List α
```

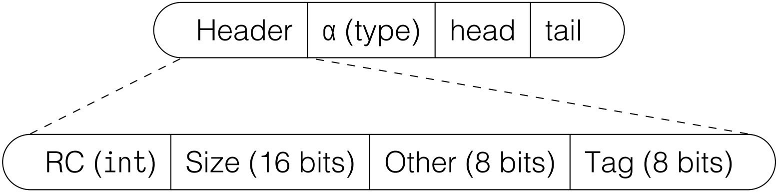
Memory Layout: `List.cons`

```
cons.{u} : {α : Type u} → α → List α → List α
```



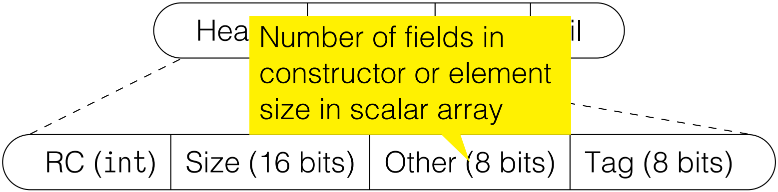
Memory Layout: `List.cons`

`cons.{u} : {α : Type u} → α → List α → List α`



Memory Layout: `List.cons`

$\texttt{cons.}\{u\} : \{\alpha : \texttt{Type}\ u\} \to \alpha \to \texttt{List}\ \alpha \to \texttt{List}\ \alpha$



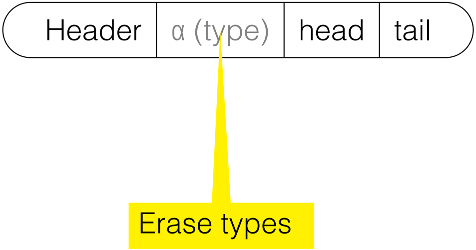
Memory Layout: `List.cons`

```
cons.{u} : {α : Type u} → α → List α → List α
```

Header	α (type)	head	tail

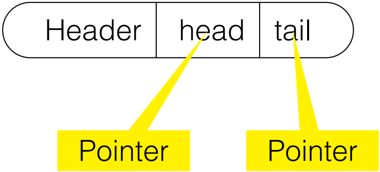
Memory Layout: `List.cons`

```
cons.{u} : {α : Type u} → α → List α → List α
```



Memory Layout: `List.cons`

`cons.{u} : {α : Type u} → α → List α → List α`

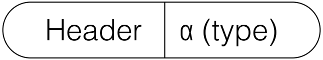


Memory Layout: `List.nil`

```
nil.{u} : {α : Type u} → List α
```

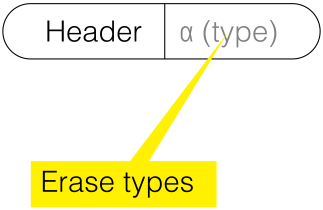
Memory Layout: `List.nil`

```
nil.{u} : {α : Type u} → List α
```



Memory Layout: `List.nil`

`nil.{u} : {α : Type u} → List α`



Memory Layout: `List.nil`

`nil.{u} : {α : Type u} → List α`

Header

No fields!

Memory Layout: `List.nil`

`nil.{u} : {α : Type u} → List α`

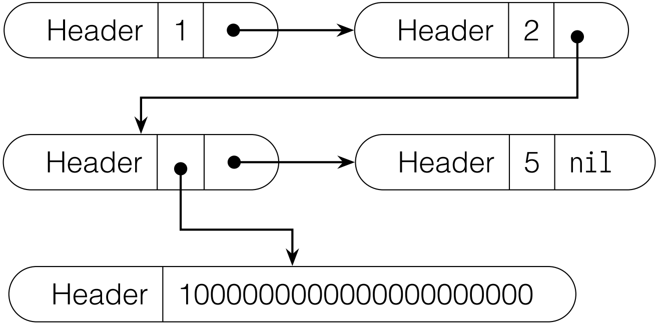
Immediate

Memory Layout: A List

```
[1, 2, 1000000000000000000000, 5] : List Nat
```

Memory Layout: A List

```
[1, 2, 1000000000000000000000, 5] : List Nat
```



Memory Layout: LList

```
structure LList (length : Nat) (α : Type u) where
  list : List α
  hasLength : list.length = length
```

Memory Layout: LList

```
structure LList (length : Nat) (α : Type u) where
  list : List α
  hasLength : list.length = length
```

desugars to...

```
inductive LList (length : Nat) (α : Type u) where
  | mk :
    (list : List α) →
    (hasLength : list.length = length) →
    LList length α
```

Memory Layout: LList

```
structure LList (length : Nat) (α : Type u) where
  list : List α
  hasLength : list.length = length
```

desugars to...

```
inductive LList (length : Nat) (α : Type u) where
  | mk :
    (list : List α) →
    (hasLength : list.length = length) →
    LList length α
```

with implicit arguments...

```
inductive LList (length : Nat) (α : Type u) where
  | mk.{u} : {α : Type u} →
    (list : List α) →
    (hasLength : list.length = length) →
    LList length α
```

Memory Layout: `LList.mk`

```
mk.{u} : {length : Nat} → {α : Type u} →
 (list : List α) →
 (hasLength : list.length = length) →
 LList length α
```

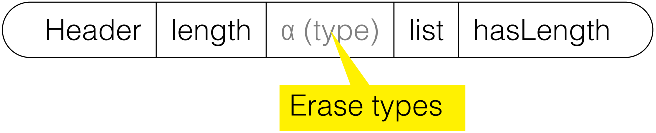
Memory Layout: `LList.mk`

```
mk.{u} : {length : Nat} → {α : Type u} →
 (list : List α) →
 (hasLength : list.length = length) →
 LList length α
```

Header	length	α (type)	list	hasLength

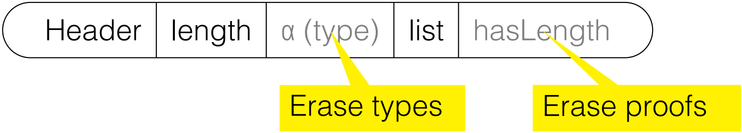
Memory Layout: `LList.mk`

```
mk.{u} : {length : Nat} → {α : Type u} →
 (list : List α) →
 (hasLength : list.length = length) →
 LList length α
```



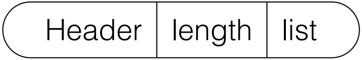
Memory Layout: `LList.mk`

```
mk.{u} : {length : Nat} → {α : Type u} →
 (list : List α) →
 (hasLength : list.length = length) →
 LList length α
```



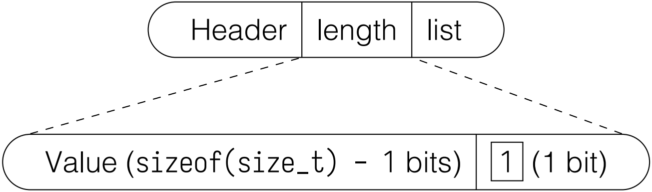
Memory Layout: `LList.mk`

```
mk.{u} : {length : Nat} → {α : Type u} →
 (list : List α) →
 (hasLength : list.length = length) →
 LList length α
```



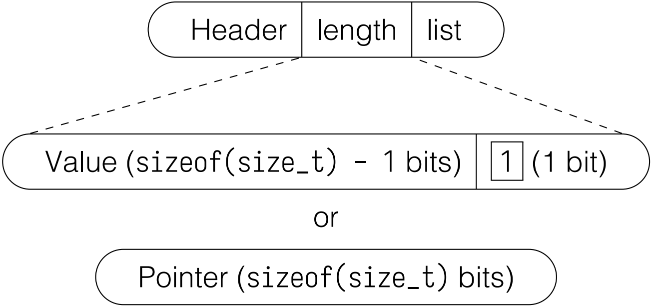
Memory Layout: LList.mk

```
mk.{u} : {length : Nat} → {α : Type u} →
 (list : List α) →
 (hasLength : list.length = length) →
 LList length α
```



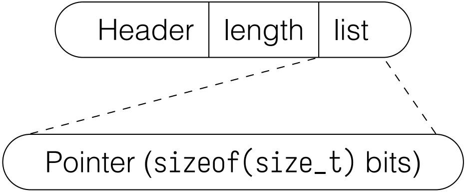
Memory Layout: `LList.mk`

```
mk.{u} : {length : Nat} → {α : Type u} →
 (list : List α) →
 (hasLength : list.length = length) →
 LList length α
```



Memory Layout: `LList.mk`

```
mk.{u} : {length : Nat} → {α : Type u} →
 (list : List α) →
 (hasLength : list.length = length) →
 LList length α
```



Memory Layout: Subtype

```
structure Subtype {α : Sort u} (p : α → Prop) where
  val : α
  property : p val
```

Memory Layout: Subtype

```
structure Subtype {α : Sort u} (p : α → Prop) where
  val : α
  property : p val
```

desugars to...

```
inductive Subtype.{u} {α : Sort u} (p : α → Prop) where
  | mk.{u} :
    {α : Sort u} → {p : α → Prop} →
    (val : α) → (property : p val) →
    @Subtype.{u} α p
```

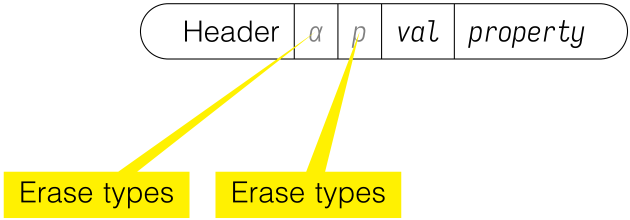
Memory Layout: `Subtype.mk`

```
mk.{u} :
  {α : Sort u} → {p : α → Prop} →
  (val : α) → (property : p val) →
  @Subtype.{u} α p
```

Header	*α*	*p*	*val*	*property*

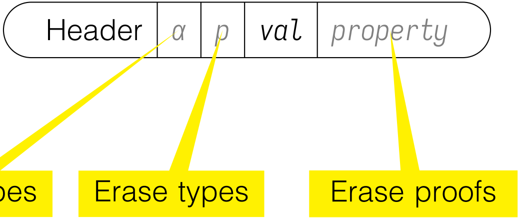
Memory Layout: `Subtype.mk`

```
mk.{u} :
  {α : Sort u} → {p : α → Prop} →
  (val : α) → (property : p val) →
  @Subtype.{u} α p
```



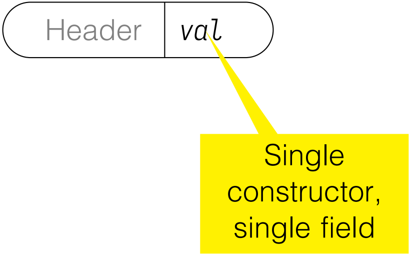
Memory Layout: `Subtype.mk`

```
mk.{u} :
  {α : Sort u} → {p : α → Prop} →
  (val : α) → (property : p val) →
  @Subtype.{u} α p
```



Memory Layout: `Subtype.mk`

```
mk.{u} :
  {α : Sort u} → {p : α → Prop} →
  (val : α) → (property : p val) →
  @Subtype.{u} α p
```



Memory Layout: `Subtype.mk`

```
mk.{u} :
  {α : Sort u} → {p : α → Prop} →
  (val : α) → (property : p val) →
  @Subtype.{u} α p
```

val

No run-time overhead!

Special Types

```
inductive Nat where
  | zero
  | succ (n : Nat)
```

Special Types

```
inductive Nat where
  | zero
  | succ (n : Nat)
```

- Special-cased in kernel and compiler - immediate or GMP
- Logical model must coincide with Peano nats
- $O(n)$ addition is a non-starter

Special Types

```
inductive Nat where
  | zero
  | succ (n : Nat)
```

- Special-cased in kernel and compiler - immediate or GMP
- Logical model must coincide with Peano nats
- $O(n)$ addition is a non-starter

```
@[extern "lean_nat_add"]
def Nat.add : (@& Nat) → (@& Nat) → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)
```

Overriding Functions

```
@[extern "lean_nat_add"]
def Nat.add : (@& Nat) → (@& Nat) → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)
```

Overriding Functions

```
@[extern "lean_nat_add"]
def Nat.add : (@& Nat) → (@& Nat) → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)

lean_obj_res lean_nat_add(
    b_lean_obj_arg a1,
    b_lean_obj_arg a2
) {
    if (lean_is_scalar(a1) && lean_is_scalar(a2))
        return lean_usize_to_nat(
            lean_unbox(a1) + lean_unbox(a2)
        );
    else
        return lean_nat_big_add(a1, a2);
}
```

Overriding Functions

```
@[extern "lean_nat_add"]
def Nat.add : (@& Nat) → (@& Nat) → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)

lean_obj_res lean_nat_add(
    b_lean_obj_arg a1,
    b_lean_obj_arg a2
) {
    if (lean_is_scalar(a1) && lean_is_scalar(a2))
        return lean_usize_to_nat(
            lean_unbox(a1) + lean_unbox(a2)
        );
    else
        return lean_nat_b
}
```

Indicates "borrowed" calling convention - caller must consume/decrement RC

Overriding Functions

```
@[extern "lean_nat_add"]
def Nat.add : (@& Nat) → (@& Nat) → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.ad

lean_obj_res lean_nat_add(
    b_lean_obj_arg a1,
    b_lean_obj_arg a2
) {
    if (lean_is_scalar(a1) && lean_is_scalar(a2))
        return lean_usize_to_nat(
            lean_unbox(a1) + lean_unb
        );
    else
        return lean_nat_big_add(a1, a2);
}
```

Kernel sees this

Running programs see this

Arrays

```
structure Array (α : Type u) where
  mk ::
  data : List α

attribute [extern "lean_array_data"] Array.data
attribute [extern "lean_array_mk"] Array.mk
```

Arrays

```
structure Array (α : Type u) where
  mk ::
  data : List α

attribute [extern "lean_array_data"] Array.data
attribute [extern "lean_array_mk"] Array.mk
```

Logically: a thin wrapper around a list

In programs: $O(n)$ conversions to/from packed arrays

Array Updates

```
def List.set : List α → Nat → α → List α
  | .cons _ as, 0,   b => .cons b as
  | .cons a as, n+1, b => .cons a (set as n b)
  | .nil,       _,   _ => .nil
```

Array Updates

```
def List.set : List α → Nat → α → List α
  | .cons _ as, 0,   b => .cons b as
  | .cons a as, n+1, b => .cons a (set as n b)
  | .nil,       _,   _ => .nil

@[extern "lean_array_fset"]
def Array.set (a : Array α)
    (i : @& Fin a.size) (v : α) : Array α where
  data := a.data.set i.val v
```

Array Updates

```
def List.set : List α → Nat → α → List α
    | .cons _ as, 0,   b => .cons b as
    | .cons a as, n+1, b => .cons a (set as n b)
    | .nil,       _,   _ => .nil
```

```
@[extern "lean_array_fset"]
def Array.set (a : Array α)
    (i : @& Fin a.size) (v : α) : Array α where
  data := a.data.set i.val v
```

`lean_array_fset` **mutates** the array when there is precisely one reference

Special Types

`Nat`	Linked list	Immediate or GMP
`Int`	`Nat` with sign	Immediate or GMP
`Array`	Linked List	Dynamic array (a la `std::vec`)
`String`	List of `Char`	Packed array of bytes (UTF-8)
`UInt8`–`UInt64`	`Fin`	Immediate

`Nat` is special in the kernel - all others only in programs

Outline

Standard Library

Standard Library

Data structures

Standard Library

Data structures	Proofs

Standard Library

Data structures	Proofs
Language features	

Standard Library

Data structures

Proofs

Language features

Automation

Useful Tools

- **`#guard_msgs`** - run a Lean command, and check that the output is what's expected
- `List.attach` - replace ℓ with `Subtype (fun x => x ∈ ℓ)`
- Linters for documentation, lemmas, etc
- Tactics
- Soon: Omega

Under Construction!

`https://github.com/leanprover/std4`

Priorities:

- Proof automation (Sledgehammer, etc)
- Data structures and associated lemmas

Outline

Programming and Metaprogramming - 21/11

- Run-Time Representations and Memory Management
- Standard Library
- Proving Termination
- Notations, Macros and Metaprogramming

Why Termination?

```
fix : (α → α) → α
```

As a program, **general recursion**
As a reasoning principle, **a circular argument**

Red Herrings

- Termination of type checking - in practice, we are not infinitely patient, and many terminating programs may run for decades or centuries

Red Herrings

- Termination of type checking - in practice, we are not infinitely patient, and many terminating programs may run for decades or centuries
- Decidability of type checking - many useful systems are nonetheless undecidable (e.g. GHC, Scala, ...)

Proving Termination

- *Structural recursion* elaborates to eliminators
- Other recursion uses well-founded relations

Demo: Termination

Termination.lean

Well-Founded Recursion

```
def f ... (x : A) ... : T :=
  ... (f e1) ... (f e2) ...
termination_by
  f ... x ... => m x
```

What this means:

1. If `m x : U`, resolve type class `WellFoundedRelation U`
2. For each recursive call `f e`, prove `e < x` w.r.t. `WellFoundedRelation.r` using a default tactic
3. Elaborate to `WellFounded.fix`

Caveat

`WellFounded.fix` is noncomputable

- Compiler generates recursive code directly
- Elaborator (lazily) proves each defining equation of the function

Alternatives

Partial functions don't require termination proofs:

```
partial def interact (n : Nat) : IO Unit := do
  let val ← askUser n
  if val = 0 then return ()
  interact val
```

The code is compiled, but Lean's logic sees an opaque constant.

Alternatives

Partial functions don't require termination proofs:

```
partial def interact (n : Nat) : IO Unit := do
  let val ← askUser n
  if val = 0 then return ()
  interact val
```

The code is compiled, but Lean's logic sees an opaque constant.

Requirements:

- Return type is inhabited
- Non non-function partial values

Alternatives, part 2

`unsafe` functions may use unrestricted general recursion, call the FFI, or use unsafe casts

`unsafe` is "infectious" - use `@[implemented_by f]` to have compiled code use (unsafe) `f`

Outline

Programming and Metaprogramming - 21/11

- Run-Time Representations and Memory Management
- Standard Library
- Proving Termination
- Notations, Macros and Metaprogramming

Notations

Lean code should resemble mathematical syntax when possible

Notations simultaneously extend the parser and provide an interpretation into existing syntax

Demo: Notations

Metaprogramming.lean

Syntax

```
inductive Syntax where
  | missing
  | node (info : SourceInfo) (kind : SyntaxNodeKind)
      (args : Array Syntax)
  | atom (info : SourceInfo) (val : String)
  | ident (info : SourceInfo)
      (rawVal : Substring) (val : Name)
      (preresolved : List Syntax.Preresolved)
```

Syntax

Parse error

```
inductive Syntax where
  | missing
  | node (info : SourceInfo) (kind : SyntaxNodeKind)
      (args : Array Syntax)
  | atom (info : SourceInfo) (val : String)
  | ident (info : SourceInfo)
      (rawVal : Substring) (val : Name)
      (preresolved : List Syntax.Preresolved)
```

Syntax

Parse error

Source location

```
inductive Syntax where
  | missing
  | node (info : SourceInfo) (kind : SyntaxNodeKind)
      (args : Array Syntax)
  | atom (info : SourceInfo) (val : String)
  | ident (info : SourceInfo)
      (rawVal : Substring) (val : Name)
      (preresolved : List Syntax.Preresolved)
```

Syntax

Parse error

Source location

```
inductive Syntax where
  | missing
  | node (info : SourceInfo) (kind : SyntaxNodeKind)
      (args : Array Syntax)
  | atom (info : SourceInfo) (val : String)
  | ident (info : SourceInfo)
      (rawVal : Substring) (val : Name)
      (preresolved : List Syntax.Preresolved)
```

Literal number or string

Syntax

Parse error

Source location

```
inductive Syntax where
  | missing
  | node (info : SourceInfo) (kind : SyntaxNodeKind)
      (args : Array Syntax)
  | atom (info : SourceInfo) (val : String)
  | ident (info : SourceInfo)
      (rawVal : Substring) (val : Name)
      (preresolved : List Syntax.Preresolved)
```

Identifier

Literal number or string

Macros

Macros allow arbitrary analysis of input syntax to produce output syntax

```
macro : Syntax → MacroM Syntax
```

```
macro "if" e:term "then" t:term "else" f:term =>
  `(ite $e (fun () => $t) (fun () => $f))
```

Quasiquotation

```
`($e + 2)
```

Constructs a syntax tree for the expression, evaluating *e* as usual.

Quasiquotation

```
`($e + 2)
```

Constructs a syntax tree for the expression, evaluating *e* as usual.

```
do let info ← Lean.MonadRef.mkInfoFromRefPos
   let scp ← Lean.getCurrMacroScope
   let mainModule ← Lean.getMainModule
   pure ⟨Lean.Syntax.node3 info
          `term_+_ e.raw (Lean.Syntax.atom info "+")
          (Lean.Syntax.node1 info `num
            (Lean.Syntax.atom info "2"))⟩
```

Hygiene

Quotation is *monadic* to avoid capture:

```
def x := 5

macro_rules
  | `(myMacro $e) =>
    `(let x := 4; x + $e) : MacroM Syntax

#eval myMacro x
```

Hygiene

Quotation is *monadic* to avoid capture:

```
def x := 5

macro_rules
  | `(myMacro $e) =>
    `(let x := 4; x + $e) : MacroM Syntax

#eval myMacro x
```

A *scope* is attached to each *x*

Hygiene

Quotation is *monadic* to avoid capture:

```
def x := 5

macro_rules
  | `(myMacro $e) =>
    `(let x := 4; x + $e) : MacroM Syntax

#eval myMacro x
```

A *scope* is attached to each *x*

The scope is not added to splices

Hygiene

Quotation is *monadic* to avoid capture:

```
def x := 5

macro_rules
  | `(myMacro $e) =>
    `(let x := 4; x + $e) : MacroM Syntax

#eval myMacro x
```

`MacroM` ensures that macro scopes are kept unique, so each act of quotation cannot interfere with others

Demo: Macros

Metaprogramming.lean

Other Metaprogramming Features

- *Elaborators* translate syntax into Lean's core language
- *Custom tactics* allow custom proof automation (more next week)
- *Language server extensions* allow custom IDE features (e.g. outline view, code actions)

Next time

Foundations and Proofs

- Lean's type theory
- Writing proofs
- Proof automation

Reading for Today

- *Functional Programming in Lean* chapters 7–10 (“Monad Transformers” through “Programming, Proving, and Performance”)
- *Counting Immutable Beans* by Ullrich and de Moura describes Lean’s memory management
- *Beyond Notations* by Ullrich and de Moura describes Lean’s metaprogramming features

Thank you!

Happy to answer questions! I'm usually here on Fridays.

- `david@lean-fro.org`
- `https://davidchristiansen.dk`

Documentation and tutorials at: `https://lean-lang.org`

Outline

Overview - 14/11

Programming and Metaprogramming - 21/11

Foundations and Proofs - 28/11
- An Example Proof
- Lean's Type Theory
- Proof Ecosystem

Tactics and Proofs

Tactics are metaprograms that construct proof terms

Tactics and Proofs

Tactics are metaprograms that construct proof terms

Macros: `Syntax → MacroM Syntax`

Tactics and Proofs

Tactics are metaprograms that construct proof terms

Macros: `Syntax → MacroM Syntax`

Elaborators : `Syntax → TermElabM Expr`

Tactics and Proofs

Tactics are metaprograms that construct proof terms

Macros: `Syntax → MacroM Syntax`

Elaborators : `Syntax → TermElabM Expr`

Tactics: `Syntax → TacticM Unit`

Outline

Foundations and Proofs - 28/11

- An Example Proof
- Lean's Type Theory
- Proof Ecosystem

Demo: Proofs

LEAN

`Quotients.lean`

Proofs and Tactics

- Lean tactics have *hygiene*
- New tactics definable as macros or directly
- Freely intermix term and tactic mode proofs

Outline

Foundations and Proofs - 28/11

- An Example Proof
- Lean's Type Theory
- Proof Ecosystem

Foundations and Culture

- Lean co-evolved with a classical community
- Wholehearted embrace of classical reasoning
- Proof automation prioritized over metatheoretic elegance

Lean's Type Theory

A variant of Coq's CIC with:

Recursion via Eliminators

$$
\begin{aligned}
&\text{plus} := \\
&\quad \lambda n\,. \\
&\qquad \mathbb{N}.\text{rec} \\
&\qquad\quad (\lambda \_ \,.\mathbb{N} \to \mathbb{N}) \\
&\qquad\quad (\lambda\ k\,.k) \\
&\qquad\quad (\lambda \_\ .\ \lambda f\ .\ \lambda k\ .\ f\,(\text{succ}\ k))
\end{aligned}
$$

Lean's Type Theory

A variant of Coq's CIC with:

Recursion via Eliminators

plus $:=$
λn .
$\mathbb{N}$.rec
$(\lambda\ \_\ .\mathbb{N} \to \mathbb{N})$
$(\lambda\ k\ .k)$
$(\lambda\_\ .\ \lambda f\ .\ \lambda k\ .\ f\,(\text{succ}\ k))$

No Cumulativity

$A : \mathsf{U}_u \not\Rightarrow A : \mathsf{U}_{u+1}$

Lean's Type Theory

A variant of Coq's CIC with:

Recursion via Eliminators

plus :=
  λn .
    $\mathbb{N}$.rec
      $(\lambda\ \_\ .\mathbb{N} \to \mathbb{N})$
      $(\lambda\ k\ .k)$
      $(\lambda\_\ .\ \lambda f\ .\ \lambda k\ .\ f\,(\mathrm{succ}\ k))$

No Cumulativity

$$A : \mathsf{U}_u \not\Rightarrow A : \mathsf{U}_{u+1}$$

Definitional Proof Irrelevance

$$\frac{P : \mathbb{P} \qquad p_1 : P \qquad p_2 : P}{p_1 \equiv p_2}$$

Lean's Type Theory

A variant of Coq's CIC with:

Recursion via Eliminators

plus :=
  λn .
    $\mathbb{N}$.rec
      $(\lambda\ \_\ .\mathbb{N} \to \mathbb{N})$
      $(\lambda\ k\ .k)$
      $(\lambda\_\ .\ \lambda f\ .\ \lambda k\ .\ f\ (\mathrm{succ}\ k))$

No Cumulativity

$$A : \mathsf{U}_u \not\Rightarrow A : \mathsf{U}_{u+1}$$

Definitional Proof Irrelevance

$$\frac{P : \mathbb{P} \qquad p_1 : P \qquad p_2 : P}{p_1 \equiv p_2}$$

Quotients with Reduction

$$\mathrm{lift}_R\ \beta\ f\ h\ (\mathrm{mk}_R\ a) \rightsquigarrow f\ a$$

Lean's Type Theory

A variant of Coq's CIC with:

Recursion via Eliminators

plus :=
  λn .
    $\mathbb{N}$.rec
      $(\lambda \_ . \mathbb{N} \to \mathbb{N})$
      $(\lambda\, k . k)$
      $(\lambda \_ .\ \lambda f .\ \lambda k .\ f\,(\mathrm{succ}\ k))$

No Cumulativity

$$A : \mathrm{U}_u \not\Rightarrow A : \mathrm{U}_{u+1}$$

Can't support HoTT

Definitional Proof Irrelevance

$$\frac{P : \mathbb{P} \qquad p_1 : P \qquad p_2 : P}{p_1 \equiv p_2}$$

Quotients with Reduction

$$\mathrm{lift}_R\ \beta\ f\ h\ (\mathrm{mk}_R\ a) \rightsquigarrow f\ a$$

Not Present

- Induction-recursion
- Coinductive types
- Higher-dimensional structure
- Sized types
- Kernel options like `--without-k` or HoTT

Standard Axioms

$$\text{propext} : \forall A, B : \mathbb{P}\ .\ A \leftrightarrow B \to A = B$$

Standard Axioms

$$\text{propext} : \forall A, B : \mathbb{P} \; . \; A \leftrightarrow B \to A = B$$

$$\text{choice} : \forall \alpha : \mathrm{U}_u \; . \; \text{nonempty } \alpha \to \alpha$$

Quotients

Axioms:

$$\begin{aligned}
\alpha/R &: \mathsf{U}_u \\
\mathsf{mk}_R &: \alpha \to \alpha/R \\
\mathsf{sound}_R &: \forall x\, y : \alpha.\, R\, x\, y \to \mathsf{mk}_R\, x = \mathsf{mk}_R\, y \\
\mathsf{lift}_R &: \forall \beta : \mathsf{U}_v. \\
&\qquad \forall f : \alpha \to \beta. \\
&\qquad\quad (\forall x\, y : \alpha.\, R\, x\, y \to f\, x = f\, y) \to \\
&\qquad\quad \alpha/R \to \beta
\end{aligned}$$

Computation:

$$\mathsf{lift}_R\ \beta\ f\ h\ (\mathsf{mk}_R\ a) \rightsquigarrow f\ a$$

Demo: Quotients

Quotients.lean

Theorem: Function Extensionality

Let $f \sim g = \forall x \,.\, f\,x = g\,x$.

Theorem: Function Extensionality

Let $f \sim g = \forall x \,.\, f\,x = g\,x$.

Assume $f, g : (x : \alpha) \to \beta\ x$ and $f \sim g$. Show $f = g$.

Theorem: Function Extensionality

Let $f \sim g = \forall x \,.\, f\, x = g\, x$.

Assume $f, g : (x : \alpha) \to \beta\ x$ and $f \sim g$. Show $f = g$.

Theorem: Function Extensionality

Let $f \sim g = \forall x \,.\, f\, x = g\, x$.

Assume $f, g : (x : \alpha) \to \beta\ x$ and $f \sim g$. Show $f = g$.

Define "extensional application":

$$f \mathbin{\$} x := \mathrm{lift}_{\sim}(\beta\ x)(\lambda g \,.\, g\, x)$$

(which trivially respects $\sim$)

Theorem: Function Extensionality

Let $f \sim g = \forall x \,.\, f\,x = g\,x$.

Assume $f, g : (x : \alpha) \to \beta\ x$ and $f \sim g$. Show $f = g$.

Define "extensional application":

$$f \mathbin{\$} x := \mathrm{lift}_{\sim}(\beta\ x)(\lambda g \,.\, g\,x)$$

(which trivially respects $\sim$)

Definitionally:

$$\begin{aligned} f &\equiv \lambda x \,.\, f\,x && \eta \\ &\equiv \lambda x \,.\, \mathrm{mk}_{\sim}\ f \mathbin{\$} x && \text{Computation rule for lift} \\ &\equiv \mathrm{mk}_{\sim}\ f \mathbin{\$} && \eta \end{aligned}$$

Theorem: Function Extensionality

To show $f = g$, we can show $\mathrm{mk}_{\sim}\ f\,\$\ = \mathrm{mk}_{\sim}\ g\,\$$.

We have:

$$\mathrm{sound}_{\sim} : f \sim g \rightarrow \mathrm{mk}_{\sim}\ f = \mathrm{mk}_{\sim}\ g$$

Thus, we can show $\mathrm{mk}_{\sim}\ f\,\$\ = \mathrm{mk}_{\sim}\ f\,\$$, which is true by reflexivity.

Demo: Function Extensionality

`Funext.lean`

Metatheory

We have:

- Consistency
- Unique typing

But:

- No normalization
- Undecidable definitional equality
- No subject reduction

Sources of Undecidability

- Proof irrelevance and subsingleton elimination (e.g. Acc)
- Proof irrelevance and imprecativity
- Quotients of propositions

Sources of Undecidability

- Proof irrelevance and subsingleton elimination (e.g. Acc)
- Proof irrelevance and imprecativity
- Quotients of propositions

Proof Irrelevance and Impredicativity

- Impredicativity and definitional proof irrelevance imply failure of normalization in an inconsistent context
- Impredicativity, definitional proof irrelevance, and propext also imply failure of normalization

See: *Failure of Normalization in Impredicative Type Theory With Proof-Irrelevant Propositional Equality*, Abel and Coquand (LMCS 16(2), 2020)

Definitional Equality

Split between undecidable "ideal" $\Gamma \vdash e \equiv e'$ and "implemented" $\Gamma \vdash e \Leftrightarrow e'$

Outline

Mathlib

More than 1,000,000 lines of Lean, formalizing lots of math

Basis for complicated work like Scholze's liquid tensor challenge and the sphere eversion project

`https://github.com/leanprover-community/mathlib4`

Aesop

Automated **E**xtensible **S**earch for **O**bvious **P**roofs

Recursively and efficiently applies a large, extensible set of rules to dispatch proof goals

Aesop: White-Box Best-First Proof Search for Lean, Limperg and From (CPP '23)

Demo: Aesop

Aesop.lean

Other Proof Tools

- Lean Auto - use existing automated provers and extract Lean proofs when possible
- Duper - a superposition prover build on top of Auto
- Loogle - search the Lean libraries by name or type
- Moogle - LLM-powered natural language theorem search

Reading for Today

- *The Type Theory of Lean* by Carneiro describes Lean's core theory
- *An Extensible Theorem Proving Frontend* by Ullrich, section 3.2, which describes later updates to the theory
- *Aesop: White-Box Best-First Proof Search for Lean* by Limperg and From describes Aesop

Thank you!

Happy to answer questions! I'm usually here on Fridays. Talk to Rasmus or Marco about PhD course credit.

- `david@lean-fro.org`
- `https://davidchristiansen.dk`

Documentation and tutorials at: `https://lean-lang.org`