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 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.
The Lean FRO

LEAN  

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

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

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

Type of argument before dot: List Nat

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

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

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

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

Type of argument before dot: List Nat

This call becomes List.length
Field Notation

```haskell
#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

```ocaml
#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

```plaintext
#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]
```

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

Examples.lean:14:17

Expected type

\[\begin{aligned}
\alpha & : \text{Type } u_1 \\
x & : \alpha \\
xs & : \text{List } \alpha \\
\text{List } \alpha & 
\end{aligned}\]

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

Other errors, warnings, and information
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

Jump to location 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

Jump to location in source

Copy message to clipboard
The Infoview

Lean Infoview

Examples.lean:14:17
Expected type
a : Type u_1
x : a
xs : List a
⊢ List a

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

Jump to location in source
Copy message to clipboard
Insert message contents as comment in source
The Infoview

Examples.lean:14:17

Expected type

\[ \alpha : \text{Type} \ u_1 \]
\[ x : \alpha \]
\[ \text{xs} : \text{List} \ \alpha \]
\[ \text{⊢ List} \ \alpha \]

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
Manually refresh
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 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

```lean
```

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
```

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

Lean requires a proof of bounds-safety by default
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

\[ A \land B \quad \text{And.intro} : a \rightarrow b \rightarrow a \land b \quad \text{Evidence of both } A \text{ and } B \]
### Evidence

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \land B$</td>
<td>Evidence of both $A$ and $B$</td>
</tr>
<tr>
<td>$A \lor B$</td>
<td>Either evidence of $A$ or evidence of $B$</td>
</tr>
</tbody>
</table>

- **And.intro**: $a \rightarrow b \rightarrow a \land b$
- **Or.inl**: $a \rightarrow a \lor b$
- **Or.inr**: $b \rightarrow a \lor b$
Evidence

\[ A \land B \quad \text{And.intro} : a \to b \to a \land b \]  
Evidence of both \( A \) and \( B \)

\[ A \lor B \quad \text{Or.inl} : a \to a \lor b \]
\[ \quad \text{Or.inr} : b \to a \lor b \]  
Either evidence of \( A \) or evidence of \( B \)

\[ A \to B \quad \text{fun (h : A) => (... : B)} \]  
Given \( A \), produce evidence of \( B \)
### Evidence

- **$A \land B$**  
  - **And.intro**: $a \to b \to a \land b$  
  - **Evidence of both $A$ and $B$**

- **$A \lor B$**  
  - **Or.inl**: $a \to a \lor b$  
  - **Or.inr**: $b \to a \lor b$  
  - **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

\[
\begin{align*}
A \land B & \quad \text{And.intro : } a \rightarrow b \rightarrow a \land b \\
A \lor B & \quad \text{Or.inl : } a \rightarrow a \lor b \\
& \quad \text{Or.inr : } b \rightarrow a \lor b \\
A \rightarrow B & \quad \text{fun } (h : A) \Rightarrow (\ldots : B) \\
\text{True} & \quad \text{True.intro} \\
\text{False} & \quad \text{Trivial evidence} \\
\end{align*}
\]

Evidence of both \( A \) and \( B \)
Either evidence of \( A \) or evidence of \( B \)
Given \( A \), produce evidence of \( B \)
Trivial evidence
No evidence at all!
Evidence

\( A \land B \quad \text{And.intro : } a \to b \to a \land b \quad \text{Evidence of both } A \text{ and } B \)

\( A \lor B \quad \text{Or.inl : } a \to a \lor b \quad \text{Either evidence of } A \text{ or evidence of } B \)

\quad \quad \text{Or.inr : } b \to a \lor b \quad \text{}\)

\( A \to B \quad \text{fun (h : A) \to (\ldots : B) \quad \text{Given } A, \text{ produce evidence of } B} \)

\( \text{True} \quad \text{True.intro} \quad \text{Trivial evidence} \)

\( \text{False} \quad \text{}\)

\( \neg A \quad \text{fun (h : A) \to (\ldots : \text{False}) \quad \text{Given } A, \text{ derive a contradiction}} \)
Evidence

∀(x : A), P  fun (x : A) =>  
(... : P)  

Provide evidence of
P for any given
x : A
Evidence

\[\forall (x : A), \, P \quad \text{fun} \quad (x : A) \implies (\ldots : P)\]

Provide evidence of \(P\) for \textbf{any} given \(x : A\)

\[\exists (x : A), \, P \quad \text{Exists.intro} : \quad (w : A) \rightarrow P \, w \rightarrow \text{Exists} \, A \, P\]

Some \(w : A\) paired with evidence of \(P[w/x]\)
Evidence

\[ \forall (x : A), \ P \ \text{fun} \ (x : A) \Rightarrow \ (\ldots : P) \]

Provide evidence of \( P \) for any given \( x : A \)

\[ \exists (x : A), \ P \ \text{Exists.intro} : \ (w : A) \rightarrow P \ w \rightarrow \ \text{Exists} \ A \ P \]

Some \( w : A \) paired with evidence of \( P[w/x] \)

Classical.em : \( \forall \{p : \text{Prop}\}, \ p \land \lnot \ p \)

propext : \( \forall \{p \ q : \text{Prop}\}, \ p \leftrightarrow q \rightarrow p = q \)
Tactics

Writing evidence by hand is slow and error-prone - tactics are programs to automate this process.
Demo: Evidence and Tactics

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

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

\[
\text{do } E_1 \quad \rightarrow \quad E_1
\]

\[
\text{do } E_1 \quad E_2 \quad \rightarrow \quad \text{Monad.bind } E_1 (\text{fun } () \Rightarrow E_2)
\]

\[
\text{do let } x \leftarrow E_1 \quad E_2 \quad \rightarrow \quad \text{Monad.bind } E_1 (\text{fun } x \Rightarrow E_2)
\]
More Features

```haskell
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

```python
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) := 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

```python
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
```

Looping over data structures
More Features

```python
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
```

def sumArrayFrom [Add $\alpha$] 
  ($start : \alpha$) ($arr : \text{Array} \ \alpha$) : $\alpha$ := 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
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
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
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 Strings

Vec String 5  —  A list of five Strings
Dependent Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List String</td>
<td>A list of Strings</td>
</tr>
<tr>
<td>Vec String 5</td>
<td>A list of five Strings</td>
</tr>
<tr>
<td>Fin 5</td>
<td>A number less than 5</td>
</tr>
</tbody>
</table>
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")
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

Tracing GC

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

```plaintext
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

```python
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

```haskell
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
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

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

\[
\text{cons.}\{u\} : \{\alpha : \text{Type } u\} \to \alpha \to \text{List } \alpha \to \text{List } \alpha
\]
Memory Layout: List.cons

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

<table>
<thead>
<tr>
<th>Header</th>
<th>α (type)</th>
<th>head</th>
<th>tail</th>
</tr>
</thead>
</table>
Memory Layout: List.cons

\[
\text{cons.}\{u\} : \{\alpha : \text{Type } u\} \to \alpha \to \text{List } \alpha \to \text{List } \alpha
\]

- **Header**
  - \(\alpha\) (type)
  - head
  - tail

- **RC (int)**
- **Size (16 bits)**
- **Other (8 bits)**
- **Tag (8 bits)**
Memory Layout: List.cons

\[
\text{cons.}\{u\} : \{\alpha : \text{Type}\ u\} \rightarrow \alpha \rightarrow \text{List } \alpha \rightarrow \text{List } \alpha
\]

Number of fields in constructor or element size in scalar array

RC (int)  |  Size (16 bits)  |  Other (8 bits)  |  Tag (8 bits)
Memory Layout: List.cons

\[
\text{cons.}\{u\} : \{\alpha : \text{Type } u\} \rightarrow \alpha \rightarrow \text{List } \alpha \rightarrow \text{List } \alpha
\]

| Header | \(\alpha\) (type) | head | tail |
Memory Layout: List.cons

\[\text{cons.}\{u\} : \{\alpha : \text{Type } u\} \rightarrow \alpha \rightarrow \text{List } \alpha \rightarrow \text{List } \alpha\]

**Header**

- \(\alpha\) (type)
- head
- tail

**Erase types**

RC (int)

Size (16 bits)

Other (8 bits)

Tag (8 bits)

Number of fields in constructor or element size in scalar array

Pointer

Pointer
Memory Layout: List.cons

\[
\text{cons.}\{u\} : \{\alpha : \text{Type } u\} \to \alpha \to \text{List } \alpha \to \text{List } \alpha
\]
Memory Layout: List.nil

\[
nil.\{u\} : \{\alpha : \text{Type } u\} \rightarrow \text{List } \alpha
\]
Memory Layout: List.nil

nil.\{u\} : \{\alpha : \text{Type} u\} \rightarrow \text{List} \alpha

\begin{tabular}{|c|c|}
\hline
Header & \alpha \ (\text{type}) \\
\hline
\end{tabular}
Memory Layout: List.nil

\[
\text{nil.\{u\} : \{\alpha : \text{Type } u\} \rightarrow \text{List } \alpha}
\]
Memory Layout: List.nil

nil.\{u\} : \{\alpha : \text{Type } u\} \to \text{List } \alpha

Header

No fields!
Memory Layout: List.nil

\[
\text{nil.}\{u\} : \{\alpha : \text{Type } u\} \rightarrow \text{List } \alpha
\]
Memory Layout: A List

[1, 2, 1000000000000000000000, 5] : List Nat
Memory Layout: A List

\[ [1, 2, 1000000000000000000000, 5] : \text{List Nat} \]
Memory Layout: LList

```haskell
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

```plaintext
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

\[ \text{mk.}\{u\} : \{\text{length} :\text{ Nat}\} \to \{\alpha : \text{Type } u\} \to \\
(\text{list} : \text{List } \alpha) \to \\
(\text{hasLength} : \text{list.length} = \text{length}) \to \\
\text{LList length } \alpha \]
Memory Layout: LList.mk

\[ \text{mk.}\{u\} : \{\text{length} : \text{Nat}\} \rightarrow \{\alpha : \text{Type } u\} \rightarrow \\
(\text{list} : \text{List } \alpha) \rightarrow \\
(\text{hasLength} : \text{list.length} = \text{length}) \rightarrow \\
\text{LList } \text{length } \alpha \]

Header | length | \(\alpha\) (type) | list | hasLength
Memory Layout: LList.mk

\[
\text{mk.}\{u\} : \{\text{length : Nat}\} \rightarrow \{\alpha : \text{Type } u\} \rightarrow \\
(\text{list : List } \alpha) \rightarrow \\
(\text{hasLength : list.length } = \text{length}) \rightarrow \\
LList \text{ length } \alpha
\]
Memory Layout: LList.mk

\[
\text{mk.}\{u\} : \{\text{length} : \text{Nat}\} \to \{\alpha : \text{Type} \ u\} \to \\
(\text{list} : \text{List} \ \alpha) \to \\
(\text{hasLength} : \text{list[length} = \text{length}) \to \\
\text{LList} \ \text{length} \ \alpha
\]

- Header
- length
- \(\alpha\) (type)
- list
- hasLength

Erase types
Erase proofs
Memory Layout: LList.mk

\[
\text{mk}\{u\} : \{\text{length} : \text{Nat}\} \to \{\alpha : \text{Type } u\} \to \\
(\text{list} : \text{List } \alpha) \to \\
(\text{hasLength} : \text{list}.\text{length} = \text{length}) \to \\
\text{LList length } \alpha
\]

<table>
<thead>
<tr>
<th>Header</th>
<th>length</th>
<th>list</th>
</tr>
</thead>
</table>
Memory Layout: LList.mk

\[
\text{mk.}\{u\} : \{\text{length} : \text{Nat}\} \to \{\alpha : \text{Type} \ u\} \to \\
(\text{list} : \text{List} \ \alpha) \to \\
(\text{hasLength} : \text{list.length} = \text{length}) \to \\
\text{LList length } \alpha
\]

Header | length | list

Value (sizeof(size_t) - 1 bits) | 1 (1 bit)
Memory Layout: LList.mk

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

Header | length | list

Value (sizeof(size_t) - 1 bits) | 1 (1 bit)

or

Pointer (sizeof(size_t) bits)
Memory Layout: LList.mk

\[\text{mk.}\{u}\ : \ \{\text{length} : \text{Nat}\} \to \{\alpha : \text{Type } u\} \to \]
\[(\text{list} : \text{List } \alpha) \to \]
\[(\text{hasLength} : \text{list.length} = \text{length}) \to \]
\[\text{LList } \text{length} \ \alpha\]
Memory Layout: Subtype

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

Memory Layout: Subtype

```plaintext
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
```

<table>
<thead>
<tr>
<th>Header</th>
<th>α</th>
<th>p</th>
<th>val</th>
<th>property</th>
</tr>
</thead>
</table>

Erase types
Erase proofs
Single constructor, single field
Memory Layout: Subtype.mk

\[
\text{mk.}\{u\} : \\
\{\alpha : \text{Sort } u\} \rightarrow \{p : \alpha \rightarrow \text{Prop}\} \rightarrow \\
(\text{val} : \alpha) \rightarrow (\text{property} : p \text{ val}) \rightarrow \\
@\text{Subtype.}\{u\} \alpha p
\]
Memory Layout: Subtype.mk

```plaintext
mk.\{u\} : 
   \{\alpha : \text{Sort } u\} \to \{p : \alpha \to \text{Prop}\} \to 
   (val : \alpha) \to (\text{property} : p \text{ val}) \to 
   @\text{Subtype.}\{u\} \alpha p
```

<table>
<thead>
<tr>
<th>Header</th>
<th>$\alpha$</th>
<th>$p$</th>
<th>$\text{val}$</th>
<th>$\text{property}$</th>
</tr>
</thead>
</table>

Erase types

Erase types

Erase proofs
Memory Layout: Subtype.mk

\[ \text{mk.}\{u\} : \{\alpha : \text{Sort } u\} \to \{p : \alpha \to \text{Prop}\} \to (\text{val : } \alpha) \to (\text{property : } p \text{ val}) \to @\text{Subtype.}\{u\} \alpha p \]

Single constructor, single field
Memory Layout: Subtype.mk

\[ \text{mk.}\{u\} : \{\alpha : \text{Sort } u\} \to \{p : \alpha \to \text{Prop}\} \to \]
\[ (\text{val} : \alpha) \to (\text{property} : p \text{ val}) \to \]
\[ @\text{Subtype.}\{u\} \alpha p \]

\[ \text{val} \]

No run-time overhead!
Special Types

inductive Nat where
| zero
| succ \(n : \text{Nat}\)
Special Types

```lean
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

\[
\text{inductive} \ \text{Nat} \ \text{where}
\]
\[
| \ \text{zero} \\
| \ \text{succ} \ (n : \text{Nat})
\]

▶ 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

```python
@[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

```python
@[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);
}
```

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.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);
}
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
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
def List.set : List a → Nat → a → List a
  | .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 a)
  (i : @& Fin a.size) (v : a) : Array a where
data := a.data.set i.val v
Array Updates

```lean
def List.set : List α → Nat → a → 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 : a) : Array α where
  data := a.data.set i.val v

lean_array_fset mutates the array when there is precisely one reference
```
## Special Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
<th>Storage Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nat</td>
<td>Linked list</td>
<td>Immediate or GMP</td>
</tr>
<tr>
<td>Int</td>
<td>Nat with sign</td>
<td>Immediate or GMP</td>
</tr>
<tr>
<td>Array</td>
<td>Linked List</td>
<td>Dynamic array (a la std::vec)</td>
</tr>
<tr>
<td>String</td>
<td>List of Char</td>
<td>Packed array of bytes (UTF-8)</td>
</tr>
<tr>
<td>UInt8–UInt64</td>
<td>Fin</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

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

Programming and Metaprogramming - 21/11
   Run-Time Representations and Memory Management
   Standard Library
   Proving Termination
   Notations, Macros and Metaprogramming
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 $\ell$ with Subtype $(\text{fun } x \Rightarrow x \in \ell)$
- 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

```haskell
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:

```lean
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:

```lean
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
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
Lean code should resemble mathematical syntax when possible

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

Metaprogramming.lean
inductive Syntax where
| missing
| node (info : SourceInfo) (kind : SyntaxNodeKind) (args : Array Syntax)
| atom (info : SourceInfo) (val : String)
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)
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

```plaintext
inductive Syntax where
  | missing
  | node (info : SourceInfo) (kind : SyntaxNodeKind) (args : Array Syntax)
  | atom (info : SourceInfo) (val : String)
```

Syntax

`inductive Syntax where`

| missing
| node (info : SourceInfo) (kind : SyntaxNodeKind) (args : Array Syntax)
| atom (info : SourceInfo) (val : String)

Parse error

Source location

Identifier

Literal number or string
Macros allow arbitrary analysis of input syntax to produce output syntax

\[ macro : \text{Syntax} \rightarrow \text{MacroM Syntax} \]

\[ macro \ "if" \ e:term \ "then" \ t:term \ "else" \ f:term \Rightarrow \\
\quad \left( \text{ite} \ e \ (\text{fun} () \Rightarrow t) \ (\text{fun} () \Rightarrow f) \right) \]
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.

```lean
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:

```python
def x := 5

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

#eval myMacro x
```
Hygiene

Quotation is monadic to avoid capture:

```python
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:

```python
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:

```python
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 $\rightarrow$ 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

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

\[
\text{plus} := \\
\lambda n . \\
\mathbb{N}.\text{rec} \\
(\lambda _ . \mathbb{N} \to \mathbb{N}) \\
(\lambda k . k) \\
(\lambda_ . \lambda f . \lambda k . f (\text{succ } k))
\]
Lean’s Type Theory

A variant of Coq’s CIC with:

Recursion via Eliminators

plus :=
\[ \lambda n \cdot \]
\[ \mathbb{N}.\text{rec} \]
\[ (\lambda _\cdot \mathbb{N} \rightarrow \mathbb{N}) \]
\[ (\lambda k .k) \]
\[ (\lambda _\cdot \lambda f . \lambda k . f (\text{succ } k)) \]

No Cumulativity

\[ A : U_u \nRightarrow A : U_{u+1} \]
Lean’s Type Theory

A variant of Coq’s CIC with:

Recursion via Eliminators

plus :=
\( \lambda n . \)
\( \mathbb{N}.\text{rec} \)
\( (\lambda _ . \mathbb{N} \rightarrow \mathbb{N}) \)
\( (\lambda k .k) \)
\( (\lambda_ . \lambda f . \lambda k . f (\text{succ } k)) \)

No Cumulativity

\( A : U_u \not\Rightarrow A : U_{u+1} \)

Definitional Proof Irrelevance

\[
\frac{P : \mathbb{P} \quad p_1 : P \quad p_2 : P}{p_1 \equiv p_2}
\]
Lean’s Type Theory

A variant of Coq’s CIC with:

Recursion via Eliminators

plus :=

\( \lambda n \cdot 
\mathbb{N}.\text{rec}
\)

\[ (\lambda \_ . \mathbb{N} \rightarrow \mathbb{N}) \]
\[ (\lambda k . k) \]
\[ (\lambda \_ . \lambda f . \lambda k . f (\text{succ } k)) \]

No Cumulativity

\( A : U_u \not\Rightarrow A : U_{u+1} \)

Definitional Proof Irrelevance

\[ \frac{P : \mathbb{P} \quad p_1 : P \quad p_2 : P}{p_1 \equiv p_2} \]

Quotients with Reduction

\[ \text{lift}_R \beta f h (\text{mk}_R a) \rightsquigarrow f a \]
Lean’s Type Theory

A variant of Coq’s CIC with:

Recursion via Eliminators

\[
\begin{align*}
\text{plus} & := \\
& \lambda n . \\
& \mathbb{N}.\text{rec} \\
& \quad (\lambda _\_ . \mathbb{N} \to \mathbb{N}) \\
& \quad (\lambda k . k) \\
& \quad (\lambda _\_ . \lambda f . \lambda k . f (\text{succ} k))
\end{align*}
\]

No Cumulativity

\[ A : U_u \nRightarrow A : U_{u+1} \]

Definitional Proof Irrelevance

\[
\begin{align*}
P : \mathbb{P} & \quad p_1 : P \\
\quad p_2 : P & \\
\hline
\quad p_1 \equiv p_2
\end{align*}
\]

Quotients with Reduction

\[
lift_R \beta f h (\text{mk}_R a) \rightsquigarrow f a
\]

Can’t support HoTT
Not Present

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

\[ \text{propext : } \forall A, B : \mathbb{P} . \ A \leftrightarrow B \rightarrow A = B \]
Standard Axioms

\[ \text{propext} : \forall A, B : \mathbb{P} . A \leftrightarrow B \rightarrow A = B \]

\[ \text{choice} : \forall \alpha : U_u . \text{nonempty } \alpha \rightarrow \alpha \]
Quotients

Axioms:

\[
\begin{align*}
\alpha/R : \mathcal{U}_u \\
\text{mk}_R : \alpha \to \alpha/R \\
\text{sound}_R : \forall x y : \alpha. R x y \to \text{mk}_R x = \text{mk}_R y \\
\text{lift}_R : \forall \beta : \mathcal{U}_v. \\
& \quad \forall f : \alpha \to \beta. \\
& \quad (\forall x y : \alpha. R x y \to f x = f y) \to \\
& \quad \alpha/R \to \beta
\end{align*}
\]

Computation:

\[
\text{lift}_R \beta f h (\text{mk}_R a) \rightsquigarrow f a
\]
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 \( f, g : (x : \alpha) \rightarrow \beta x \) 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”:

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

(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”:

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

(which trivially respects $\sim$)

Definitionally:

$$f \equiv \lambda x . f x \quad \eta$$

$$\equiv \lambda x . \text{mk}_{\sim} f \$ x \quad \text{Computation rule for lift}$$

$$\equiv \text{mk}_{\sim} f \$ \quad \eta$$
To show $f = g$, we can show $\text{mk}_\sim f \$ = \text{mk}_\sim g \$.

We have:

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

Thus, we can show $\text{mk}_\sim f \$ = \text{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 imprecavity
- Quotients of propositions
Sources of Undecidability

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

Foundations and Proofs - 28/11
  An Example Proof
  Lean’s Type Theory
Proof Ecosystem
  Mathlib
  Aesop
  Other Tools
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 Extensible Search for Obvious Proofs

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
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