Lean 4: A Guided Preview

Sebastian Ullrich
A brief history of Lean

- Lean 0.1 (2014)
- Lean 2 (2015)
  - first official release
  - fixed tactic language
- Lean 3 (2017)
  - make Lean a meta-programming language: build tactics in Lean
  - backed by a bytecode interpreter
- Lean 4 (201X)
  - make Lean a general-purpose language: native back end, FFI, ...
  - reimplement Lean in Lean
Lean 3 system overview

- C++
- Lean
- Other

Diagram:
- Editor
- Language server
- Parser
- Elaborator
- Tactic framework
- Kernel
- Compiler
- Module system

Tools:
- Editor
- Language server
- Parser
- Elaborator
- Tactic framework
- Kernel
- Compiler
- Module system

Languages:
- C++
- Lean
- Other
Lean 4 system overview and progress

- C++
- Lean
- Other

Diagram:
- Editor
- Language server
- Parser
- Elaborator
- Tactic framework
- Kernel
- Compiler
- Module system
Lean 4 system overview and progress

- C++
- Lean
- other

editor

language server

parser

module system

elaborator

tactic framework

kernel

compiler
Lean 4 system overview and progress

- C++
- Lean
- other

Diagram:
- Editor
- Language server
- Parser
- Elaborator
- Tactic framework
- Kernel
- Compiler
- Module system
- FFI

FFI: Foreign Function Interface
Lean 4 system overview and progress

- Editor
- Language server
- Parser
- Elaborator
- Tactic framework
- Kernel
- Compiler
- FFI
- Module system

Languages:
- C++
- Lean
- Other
Lean 4 system overview and progress

C++
Lean
other

editor
language server

parser
macro expander

tactic framework

module system

elaborator

kernel
compiler

FFI
Lean 4 system overview and progress

- **C++**
- **Lean**
- **other**

Diagram:
- Editor
  - Language server
    - Parser
      - Macro expander
      - Elaborator
        - Kernel
        - Compiler
          - FFI
Lean 4 system overview and progress

- Editor
- Language server
- Parser
- Macro expander
- Elaborator
- Tactic framework
- Kernel
- Compiler
- FFI
- Module system

C++
Lean
Other
Lean 4 system overview and progress

- C++
- Lean
- other

Diagram:
- Editor
- Language server
- Parser
- Macro expander
- Elaborator
- Tactic framework
- Kernel
- Compiler
- FFI

Module system
Lean 4 system overview and progress

- Editor
- Language server
- Parser
- Macro expander
- Elaborator
- Tactic framework
- Kernel
- Compiler
- FFI
- Module system

Languages:
- C++
- Lean
- Other
New parser [mostly implemented]

- completely accessible and extensible

```
@[parser]
def my_inductive.parser : command_parser :=
node! my_inductive ["inductive",
  name: ident_univ_params.parser,
  sig: opt_decl_sig.parser,
  ext: node! my_inductive_base ["extends", base: term.parser]?,
  local_notation: notation_like.parser?,
  intro_rules: intro_rule.parser*]
```

arbitrary local backtracking and tokenizing
concrete syntax tree fully accessible to tooling
New parser [mostly implemented]

- completely accessible and extensible
- arbitrary local backtracking and tokenizing

notation `{ `xs:(foldr ` `, `(x xs, set.insert x xs) ∅ `})` := xs
notation `{ `binder ` // ` r:(scoped p, subtype p) `}` := r
notation `{ `binder ` ∈ ` s ` | ` r:(scoped p, set.sep p s) `}` := r

```lean
def symbol_quote.parser : term_parser :=
node! symbol_quote [
  left_quote: raw_str "`",
  symbol: raw $ take_until (= `\`),
  right_quote: raw_str "`" tt, -- consume trailing ws
  prec: precedence.parser?]
```

New parser [mostly implemented]

- completely accessible and extensible
- arbitrary local backtracking and tokenizing
- concrete syntax tree fully accessible to tooling
  - auto completion, document generation, code formatting, refactoring, ...
  - jump to definition and documentation of any syntax
Macros [mostly implemented?]

- most general syntax sugars: arbitrary syntax tree transformations

```python
@parser
def set_lit.parser : term_parser :=
node! set_lit ["{", elems: sep_by ",", " term.parser, "}"]

@transformer
def set_lit.transformer : transformer :=
\lambda stx, 
  let v := view set_lit stx in
  pure $ v.elems.foldr (\lambda x xs, `(set.insert %x %xs)) `(\emptyset)
```
Macros [mostly implemented?]

- most general syntax sugars: arbitrary syntax tree transformations

```lean
syntax set_lit := `\{ (sep_by ", " term.parser) `\}

syntax_translations set_lit
| {} := ∅
| {%%x, %%xs...} := set.insert %%x {%%xs...}
```

(hypothetical Isabelle-like macro-macros)
Macros [mostly implemented?]

- most general syntax sugars: arbitrary syntax tree transformations

```lean
def my_inductive.transformer : transformer :=
\ stx,
  let v := view my_inductive stx in
pure $ review "inductive" {v with
    intro_rules := match v.ext with
    | some ext := {name := 'base, sig := {params := ["a, ext.base"]}} :: v.intro_rules
    | none := v.intro_rules
  }
end
```
Macros [mostly implemented?]

- most general syntax sugars: arbitrary syntax tree transformations
- names are resolved (hygienically) only after expansion

```lean
@[parser]
def subty.parser : term.parser :=
node! subty ["{", x: binder.parser, " // ", cond: term.parser, "}"]

@[transformer]
def subtype.transformer : transformer :=
\ stx,
  let v := view subty stx in
pure `(subtype (\ %%v.x, %%v.cond))
Macros [mostly implemented?]

- most general syntax sugars: arbitrary syntax tree transformations
- names are resolved (hygienically) only after expansion

```lean
syntax subty := `{ binder.parser `// term.parser `}`

syntax_translations subty
| `{%x // %cond} := subtype (\ %x, %cond)
```
Managing syntax [planned]

“How do I manage my domain-specific set of notations?”
Managing syntax [planned]

“How do I manage my domain-specific set of notations?”

```
namespace my_domain
    -- @[parser]
    def my_notation1.parser : term_parser := ...
    ...
end my_domain
...
local attribute [parser] my_domain.my_notation1
local attribute [parser] my_domain.my_notation2
local attribute [parser] my_domain.my_notation3
...
```

Hardly scalable...
Managing syntax [planned]

“How do I manage my domain-specific set of notations?”

```lean
namespace my_domain
  @[parser] -- scoped by default
  def my_notation.parser : term_parser := ...
  ...
end my_domain
...
open [parser] my_domain
...
```

Lean 2’s scoped attributes return!
Managing syntax [planned]

“How do I manage my domain-specific set of notations?”

```lean
namespace my_domain
  @[parser] -- scoped by default
  def my_notation.parser : term_parser := ...
  ...
end my_domain
...

open [parser] my_domain
...
```

Lean 2’s scoped attributes return!
Main lesson we learned from Lean 2: 
*Most* attributes, like `[reducible]` and `[simp]`, should *not* be scoped (by default)
Better trace logs [planned]

make traces structured and lazy

- collect trace points during initial elaboration

<table>
<thead>
<tr>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>elaborator trace</td>
</tr>
<tr>
<td>class instances trace</td>
</tr>
<tr>
<td>decidable (n = 0)</td>
</tr>
<tr>
<td>has_add nat</td>
</tr>
<tr>
<td>simp trace</td>
</tr>
</tbody>
</table>
Better trace logs [planned]

make traces structured and lazy

- collect trace points during initial elaboration
- when full trace is requested, re-elaborate

<table>
<thead>
<tr>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>elaborator trace</td>
</tr>
<tr>
<td>class instances trace</td>
</tr>
<tr>
<td>decidable (n = 0)</td>
</tr>
<tr>
<td>?x_0 : decidable (n = 0) := coe_decidable_eq ?x_48 is_def_eq failed</td>
</tr>
<tr>
<td>?x_0 : decidable (n = 0) := @ne_decidable ?x_89 ?x_90 ?x_91 ?x_92 is_def_eq failed</td>
</tr>
<tr>
<td>?x_0 : decidable (n = 0) := @forall_prop_decidable ?x_93 ?x_94 ?x_95 is_def_eq failed</td>
</tr>
<tr>
<td>?x_0 : decidable (n = 0) := @implies_decidable ?x_109 ?x_110 ?x_111 ?x is_def_eq failed</td>
</tr>
<tr>
<td>?x_0 : decidable (n = 0) := @decidable_of_decidable_eq ?x_123 ?x_124 ?x_126 : decidable_eq ℕ := nat.decidable_eq</td>
</tr>
<tr>
<td>has_add nat</td>
</tr>
<tr>
<td>simp trace</td>
</tr>
</tbody>
</table>
More consistent namespacing [in progress]

- open is now “sticky”

```coq
open nat
namespace nat
  def random := 0
end nat
#check random
```

\[1\] https://github.com/coq/coq/issues/6254#issuecomment-450641538
More consistent namespacing [in progress]

- open is now “sticky”

```coq
oop nat
namespace nat
  def random := 0
end nat
#check random
```

- parameters have been removed to simplify resolution

---

1[https://github.com/coq/coq/issues/6254#issuecomment-450641538](https://github.com/coq/coq/issues/6254#issuecomment-450641538)
Clarifying imports [proposal]

import init.data.set
import data.set -- ?

open set -- ??

import ...two_dirs_up

Connection between modules, packages, and namespaces in Lean 3 is not very clear
Connection between modules, packages, and namespaces in Lean 3 is not very clear
Proposal: Prefix module name with package name, use syntax more reminiscent of file paths

```lean
import init.data.set
import data.set  -- ?

open set  -- ??

import ../../two_dirs_up
```

```lean
import "init/data/set"
import "mathlib/data/set"

open set

import "../../two_dirs_up"
```
Thoughts about eventual porting of Lean 3 code

- syntax changes: mostly superficial, automatable
Thoughts about eventual porting of Lean 3 code

- syntax changes: mostly superficial, automatable
  One possible path: Incrementally reimplement Lean 3 syntax as macros first, then unfold them as final step

```lean
#lang lean3
import data.set
...
```
Thoughts about eventual porting of Lean 3 code

- syntax changes: mostly superficial, automatable
- elaborator changes: probably not too drastic
Thoughts about eventual porting of Lean 3 code

- syntax changes: mostly superficial, automatable
- elaborator changes: probably not too drastic
- library changes: mostly *missing* API, needs to be reimplemented
  - but not necessarily in the stdlib
Conclusion

- Many core features are starting to take shape
- Still much to be done
- Eventually should have many opportunities for community to get us back to and beyond Lean 3’s library
Conclusion

- Many core features are starting to take shape
- Still much to be done
- Eventually should have many opportunities for community to get us back to and beyond Lean 3’s library

Thank you!
Conclusion

- Many core features are starting to take shape
- Still much to be done
- Eventually should have many opportunities for community to get us back to and beyond Lean 3’s library

More presentations about Lean 4:
- 2018/08/03 Lean: past, present and future by Leo
- 2018/10/12 My internship report - new parser, mostly
- 2018/12/12 An optimized memory model for an interactive theorem prover

Find these and more at

https://leanprover.github.io/publications