Metaprograms and Proofs: Macros in Lean 4

Sebastian Ullrich (KIT)
Mathematicians welcome computer-assisted proof in ‘grand unification’ theory

Proof-assistant software handles an abstract concept at the cutting edge of research, revealing a bigger role for software in mathematics.

https://www.nature.com/articles/d41586-021-01627-2
The Liquid Tensor Experiment

Peter Scholze, 2020: “Is my proof of this really correct?”

Let $0 < p' < p \leq 1$ be real numbers. Let $S$ be a profinite set, and let $\mathcal{M}_{p'}(S)$ be the space of $p'$-measures on $S$. Let $V$ be a $p$-Banach space. Then

$$\text{Ext}^i(\mathcal{M}_{p'}(S), V) = 0$$

for all $i \geq 1$.

Johan Commelin et al., 2022: “It is.”

**Theorem Liquid Tensor Experiment** ($p' \ p : \mathbb{R} \geq 0$) \[\text{fact (} 0 < p' ) \text{]} \ [\text{fact (} p' < p \text{)}] \ [\text{fact (} p \leq 1 \text{)}] \ (S : \text{Profinite}) \ (V : \text{pBanach} \ p) :$

\[\forall i > 0, \text{Ext}^i(\mathcal{M}_{\{p'\}} S) V = 0 := \text{ -- the proof . . .}\]

“Also we simplified it.”

“[…] absolutely insane […]”
The *Lean*ing Tower of Macros

\[ \text{infixl:65 "++" => append -- e.g. `x ++ y`} \]

\[ \text{notation:65 x:65 "++" y:66 => append x y} \]

\[ \text{macro:65 x:term:65 "++" y:term:66 : term => `(append $x $y)`} \]

\[ \text{syntax:65 term:65 "++" term:66 : term} \]

\[ \text{macro_rules} \]
\[ | `(\$x ++ \$y) => `(append \$x \$y)` \]

\[ \text{elab_rules : term} \]
\[ | `(\$x ++ \$y) => elabTerm `(append \$x \$y)` \]

All built-in syntax in Lean is expressed in one of these stages!

- unary/binary notation
- n-ary mixfix notation
- arbitrary syntax transformation in arbitrary category
- separate syntax & semantics
- extensible rule set
- syntactic pattern matching
- type-aware surface syntax -> core transformation

compare: *micros* [Krishnamurthi et al. '99], Klister [Barrett et al. '20]
Macro Showcase: leanprover/doc-gen4

```lean
declare_syntax_cat jsxElement
syntax "<" ident jsxAttr* ">": jsxElement

---

macro_rules
| `(<$n $attrs* />) =>
  `(Html.element $(quote (toString n.getId)) …)
| `(<$n $attrs* >$children*$)</m> => …

---

def instanceToHtml (name : Name) : HtmlM Html :=
  return <li><a href={←toLink name}>{name}</a></li>

def instancesToHtml (instances : Array Name) : HtmlM
Html :=
  return
  <details class="instances">
    <summary>Instances</summary>
    <ul>
      [← instances.mapM instanceToHtml]
    </ul>
  </details>
```
Macro Showcase: dwrensha/lean4-maze

```lean
syntax "░" : game_cell  -- empty
syntax "▓" : game_cell  -- wall
syntax "@" : game_cell  -- player

syntax "│" game_cell* "│\n" : game_row

...  

macro_rules
| `(┌ $tb:horizontal_border* ┐
  $rows:game_row*
 └ $bb:horizontal_border* ┘) => ...
```

```lean
def maze1 :=

```

Macro Showcase: dwrensha/lean4-maze

syntax "░" : game_cell -- empty
syntax "▓" : game_cell -- wall
syntax "@" : game_cell -- player
syntax "│" game_cell* "|
  "\n" : game_row

macro_rules
  | `(┌ $ tb:horizontal_border* ┐
     $ rows:game_row*
    └ $ bb:horizontal_border* ┘) => ...

macro "west" : tactic => `(apply step_west; ...)
Lean Macros in Comparison

Grammar enforces **static structure** on macros

⇒ get structured *concrete syntax tree*, automatic pretty printer, …

\[ x \quad ++ \quad y \quad ++ \quad z \]

\[
\text{term}_+ + _+
(\text{term}_+ + _+
"\text{term}_+ + _+" :317: `\text{x}:318: " " \text{term}_+ + _+":319: ++":321: " " :322: `\text{y}:323: "
"":324: ++":326: "
"":327: `\text{z}:328: "" )
]

Compare Rust, Honu: (some) structure discovered during expansion
In this paper, we offer *Honu* as an example in the middle ground between the syntactic minimalism of Lisp and maximal grammatical freedom of Lean. Our immediate goal is to produce a syntax that is more natural for many programmers than Lisp notation—most notably, using infix notation for operators—but that is similarly easy for programmers to extend.

[Rafkind & Flatt ‘12]
Lean Macros in Comparison

Lean modules processed command-by-command

```
def g := f
def f := 0

def h x y := x ++ y
infixl:65 "++" => append
```
Lean Macros in Comparison

Lean modules processed command-by-command
⇒ syntax/macro must be defined strictly before use
⇒ no local macros, restricted macro-macros
⇒ trivial macro resolution, simplified & optimized hygiene algorithm
Typed Syntax

Lean is dependently typed... but the syntax tree is untyped

```lean
syntax "mk_0" ident : command
macro_rules
  | `(mk_0 $id) => `(def $id := 0)
```
Typed Syntax

Lean is dependently typed... but the syntax tree is untyped

```lean
syntax "mk_0" ident : command
macro_rules
  | `(mk_0 $id) => `(def $id := 0) -- oops, not the same kind of identifier

syntax declId := ident ("." ident,+ ")?
syntax defCmd := "def" declId ...
```
Lean is dependently typed... but the syntax tree is untyped

```lean
syntax "mk_0" ident : command
macro_rules
| `(mk_0 $id) => `(def $id \[id=\] := 0)

syntax declId := ident ("." ident,+: ")?
syntax defCmd := "def" declId ...
```
Typed Syntax

Lean is dependently typed... but the syntax tree is untyped

```lean
syntax "mk_0" ident : command

macro_rules
  | `(mk_0 $id) => `(def $id := 0) -- oops, not the same kind of identifier

But we know its grammar! Let’s make use of that

argument
  id
has type
  TSyntax `ident : Type
but is expected to have type
  TSyntax `declId : Type
```
Typed Syntax

Even better: introduce custom coercion to auto-fix syntax tree

instance : Coe (TSyntax `ident) (TSyntax `declId) where
coe id := `(declId| $id:ident)

syntax "mk_0" ident : command
macro_rules
  | `(mk_0 $id) => `(def $id := 0) -- now accepted as is!
Summary

Lean’s macro system is derived from Racket’s, yet with fundamental differences

More focus on syntax, less focus on advanced macro features

Safe syntax manipulations with typed syntax

slides, papers, docs: leanprover.github.io