Metaprograms and Proofs: Macros in Lean 4

Sebastian Ullrich (KIT)

Karlsruher Institut für Technologie

THEOREM PROVER

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

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

for all $i \geq 1$.

Johan Commelin et al., 2022: “It is.”

```
theorem liquid_tensor_experiment (p' p : ℝ≥0) [fact (0 < p')] [fact (p' < p)]
    [fact (p ≤ 1)] (S : Profinite) (V : pBanach p) :
  ∀ i > 0, Ext i (𝓜_{p'} S) V ≅ 0 := -- the proof ...
```

“Also we simplified it.”

“[...] absolutely insane [...]”

The *Lean*ing Tower of Macros

```
infixl:65 "++" => append  -- e.g. `x ++ y`
```

unary/binary notation

```
notation:65 x:65 "++" y:66 => append x y
```

n-ary mixfix notation

```
macro:65 x:term:65 "++" y:term:66 : term => `(append $x $y)
```

arbitrary syntax transformation in arbitrary category

```
syntax:65 term:65 "++" term:66 : term
macro_rules
  | `($x ++ $y) => `(append $x $y)
```

separate syntax & semantics
extensible rule set
syntactic pattern matching

```
elab_rules : term
  | `($x ++ $y) => elabTerm `(append $x $y)
```

type-aware surface syntax -> core transformation
compare: *micros* [Krishnamurthi et al. ‘99],
Klister [Barrett et al. ‘20]

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

Macro Showcase: leanprover/doc-gen4

```
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>
```

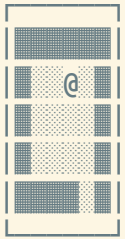
Macro Showcase: dwrensha/lean4-maze

```
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* ┘) => …
```

```
def maze1 :=
```



Macro Showcase: dwrensha/lean4-maze

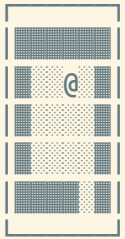
```
syntax "░" : game_cell  -- empty
syntax "▓" : game_cell  -- wall
syntax "@" : game_cell  -- player

syntax "|" game_cell* "|\n" : game_row
…

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

macro "west" : tactic => `(apply step_west; …)
```

```
def maze1 :=
```



```
example : can_escape maze1 := by
  west
  west
  east
  south
  south
  east
  east
  south
  out
```

Lean Macros in Comparison

Grammar enforces static structure on macros

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

```
x ++ y ++ z
```

```
(term_++_
  (term_++_ "":317:`x:318:" " "":319:"++":321:" " "":322:`y:323:" ")
  "":324:"++":326:" "
  "":327:`z:328:""))
```

Compare Rust, Honu: (some) structure discovered during expansion

Lean Macros in Comparison

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

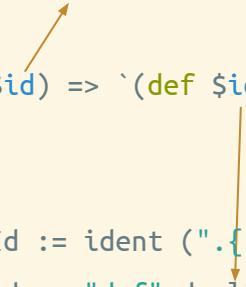
```
syntax "mk_0" ident : command
macro_rules
  | `(mk_0 $id) => `(def $id := 0)
```

Typed Syntax

Lean is dependently typed… but the syntax tree is untyped

```
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 ...
```



Typed Syntax

Lean is dependently typed… but the syntax tree is untyped

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


syntax declId := ident (".{" ident,+ "}")?
syntax defCmd := "def" declId ...
```

Typed Syntax

Lean is dependently typed… but the syntax tree is untyped

```
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