Embedding Languages into Lean 4

Sebastian Ullrich

Programming paradigms group - IPD Snelting

Lean is ...

yet another dependently-typed theorem prover

Lean is ...

- opinionated
  - implements a single logic: CIC with proof irrelevance and quotient types
    - good for automation, great for classical mathematics
    - not good for metatheoretic properties

Lean is ...

- opinionated
  - implements a single logic: CIC with proof irrelevance and quotient types
    - good for automation, great for classical mathematics
    - not good for metatheoretic properties
    - “We’re not in the type theory research business”

Lean is ...

- opinionated
  - implements a single logic: CIC with proof irrelevance and quotient types
    - good for automation, great for classical mathematics
    - not good for metatheoretic properties
    - "We're not in the type theory research business"
  - typeclasses as the main abstraction interface

Lean is ...

- opinionated
  - implements a single logic: CIC with proof irrelevance and quotient types
    - good for automation, great for classical mathematics
    - not good for metatheoretic properties
    - "We're not in the type theory research business"
  - typeclasses as the main abstraction interface
- "modern"/un-arcane
  - unobtrusive syntax heavily reliant on Unicode
  - good integration into VS Code
  - simple tooling that "mostly just works": leanpkg, elan

Lean is ...

- opinionated
  - implements a single logic: CIC with proof irrelevance and quotient types
    - good for automation, great for classical mathematics
    - not good for metatheoretic properties
    - "We're not in the type theory research business"
  - typeclasses as the main abstraction interface
- "modern"/un-arcane
  - unobtrusive syntax heavily reliant on Unicode
  - good integration into VS Code
  - simple tooling that "mostly just works": leanpkg, elan
- welcoming
  - excellent introductory text: Theorem Proving in Lean
    - comes with passable online editor
  - *huge* friendly crowd on the Zulip chat
    - real-time chat with beginners-only section is crucial

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 (202X)
  - make Lean a general-purpose language: native back end, FFI, ...
  - reimplement Lean in Lean

Towards a fully extensible frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

Towards a fully extensible frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

- extensible syntax from simple mixfix notations to character-level parsing

Towards a fully extensible frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

- extensible syntax from simple mixfix notations to character-level parsing
- extensible semantics from simple syntax sugars to type-aware elaboration

Towards a fully extensible frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

- extensible syntax from simple mixfix notations to character-level parsing
- extensible semantics from simple syntax sugars to type-aware elaboration
- extensible tooling with access to frontend metadata
    - concrete syntax tree
    - elaboration annotations (TBD)

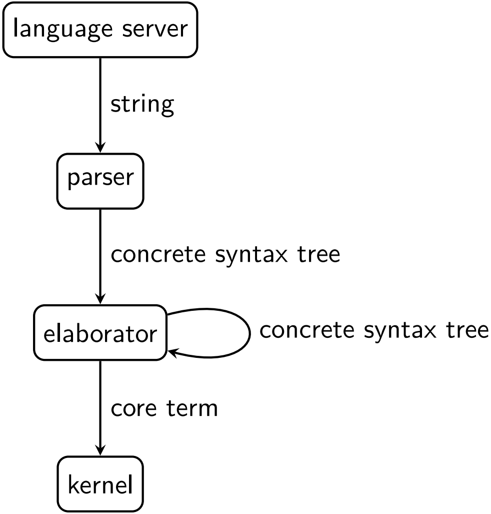
Towards a fully extensible frontend

Goal: *democratize* frontend by removing the barrier between built-in and user-defined notions

- extensible syntax from simple mixfix notations to character-level parsing
- extensible semantics from simple syntax sugars to type-aware elaboration
- extensible tooling with access to frontend metadata
    - concrete syntax tree
    - elaboration annotations (TBD)

Non-goal: extensible type theory

Frontend: overview



Concrete syntax tree

provide

- precise source locations
- whitespace and comments
- erroneous input

for

- code editors
- documentation generators
- code formatters
- refactoring tools
- better LaTeX highlighting...

Extensible concrete syntax tree

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

structure SourceInfo :=
(leading  : Substring)
(pos      : String.Pos)
(trailing : Substring)

abbrev SyntaxNodeKind := Name
```

```
a → b
```

```
(Term.arrow `a "→" `b)
```

Parser

Karlsruhe Institute of Technology

- Lean 3: basic lexer, LL(1) recursive descent parser
- Isabelle: basic lexer, Earley parser for arbitrary context-free grammars, delimited terms

Parser

- Lean 3: basic lexer, LL(1) recursive descent parser
- Isabelle: basic lexer, Earley parser for arbitrary context-free grammars, delimited terms
- Lean 4: arbitrary, character-based parser; combinators including Pratt parser and longest-prefix matching

Parser

Karlsruhe Institute of Technology

- Lean 3: basic lexer, LL(1) recursive descent parser
- Isabelle: basic lexer, Earley parser for arbitrary context-free grammars, delimited terms
- Lean 4: arbitrary, character-based parser; combinators including Pratt parser and longest-prefix matching
  - problem: monadic parser combinators allocate like crazy, lexing should be cached

Parser state

```
def ParserFn := ParserContext → ParserState → ParserState

structure ParserContext :=
(input    : String)
(fileName : String)
(env      : Environment)
(tokens   : TokenTable)

structure ParserState :=
(pos      : String.Pos)
(cache    : ParserCache)
(errorMsg : Option Error)
(stxStack : Array Syntax)
```

Parser: syntax stack

```
def nodeFn (k : SyntaxNodeKind) (p : ParserFn) : ParserFn :=
fun c s ⇒
  let iniSz := s.stxStack.size;
  let s     := p c s;
  let stack := s.stxStack;
  let newNode := Syntax.node k (stack.extract iniSz stack.size);
  let stack   := stack.shrink iniSz;
  let stack   := stack.push newNode;
  { s with stxStack := stack }
```

```
nodeFn `Term.arrow (identFn ≫ symbolFn "→" ≫ identFn)
```

```
   [..., `a, "→", `b]
~> [..., (Term.arrow `a "→" `b)]
```

Parser: token caching

cache last “token” read

```
def tokenFn : ParserFn :=
fun c s ⇒
  let i := s.pos;
  let tkc := s.cache.tokenCache;
  if tkc.startPos == i then
    let s := s.pushSyntax tkc.token;
    s.setPos tkc.stopPos
  else
    let s := tokenFnAux c s;
    updateCache i s
```

Parser: token caching

```
def identFn : ParserFn :=
fun c s ⇒
  let iniPos := s.pos;
  let s      := tokenFn c s;
  if s.hasError || !s.stxStack.back.isIdent then s.mkErrorAt "identifier" iniPos else s
```

Parser: token caching

Karlsruhe Institute of Technology

```
def identFn : ParserFn :=
fun c s ⇒
  let iniPos := s.pos;
  let s      := tokenFn c s;
  if s.hasError || !s.stxStack.back.isIdent then s.mkErrorAt "identifier" iniPos else s
```

```
structure Parser :=
(fn   : ParserFn)
(info : ParserInfo)

structure ParserInfo :=
(collectTokens : List TokenConfig → List TokenConfig    := id)
(firstTokens   : FirstTokens                            := FirstTokens.unknown)

structure TokenConfig :=
(val     : String)
(lbp     : Option Nat)
```

Pratt parser

token-indexed precedence parsing with longest-match semantics

```
def prattParser (tables : PrattParsingTables) (rbp : Nat := 0) : ParserFn

structure PrattParsingTables :=
(leadingTable    : TokenMap Parser)
(leadingParsers  : List Parser)
(trailingTable   : TokenMap Parser)
(trailingParsers : List Parser)

def leadingParser (tables : PrattParsingTables) : ParserFn :=
fun c s ⇒
  let (s, ps) := indexed tables.leadingTable c s;
  let ps      := tables.leadingParsers ++ ps;
  longestMatchFn ps c s
```

Actual stdlib parsing

Syntactic categories are Pratt parsers extensible via attributes

```
@[init] def regTermCat : IO Unit :=
registerSyntaxCategory `term

def term (rbp : Nat := 0) : Parser :=
categoryParser `term rbp

@[termParser] def anonymousCtor := node `Term.anonymousCtor (
  symbol "⟨" appPrec >> sepBy term ", " >> "⟩")

def optIdent : Parser := optional (try (ident >> " : "))
@[termParser] def «if»  := node `Term.if (
  "if " >> optIdent >> term >> " then " >> term >> " else " >> term)
```

Actual stdlib parsing

Syntactic categories are Pratt parsers extensible via attributes

```
declare_syntax_cat term

syntax "⟨" (sepBy term ", ") "⟩" : term

syntax optIdent := (try (ident " : "))?
syntax "if " optIdent term " then " term " else " term : term
```

Macros

Karlsruhe Institute of Technology

```
syntax "if " optIdent term " then " term " else " term : term
```

Apply meaning to syntax via recursive syntactic substitutions (or *macros*):

```
macro_rules
| `(if $h : $cond then $t else $e) ⇒ `(dite $cond (fun $h ⇒ $t) (fun $h ⇒ $e))
| `(if $cond then $t else $e)      ⇒ `(if h : $cond then $t else $e)
```

Macros

```
syntax "if " optIdent term " then " term " else " term : term
```

Apply meaning to syntax via recursive syntactic substitutions (or *macros*):

```
macro_rules
| `(if $h : $cond then $t else $e) ⇒ `(dite $cond (fun $h ⇒ $t) (fun $h ⇒ $e))
| `(if $cond then $t else $e)      ⇒ `(if h : $cond then $t else $e)
```

```
if True then h else True.intro
```

Macros

```
syntax "if " optIdent term " then " term " else " term : term
```

Apply meaning to syntax via recursive syntactic substitutions (or *macros*):

```
macro_rules
| `(if $h : $cond then $t else $e) ⇒ `(dite $cond (fun $h ⇒ $t) (fun $h ⇒ $e))
| `(if $cond then $t else $e)      ⇒ `(if h : $cond then $t else $e)
```

```
if True then h else True.intro  -- unknown identifier 'h'
```

Macros

```
syntax "if " optIdent term " then " term " else " term : term
```

Apply meaning to syntax via recursive syntactic substitutions (or *macros*):

```
macro_rules
| `(if $h : $cond then $t else $e) ⇒ `(dite $cond (fun $h ⇒ $t) (fun $h ⇒ $e))
| `(if $cond then $t else $e)      ⇒ `(if h : $cond then $t else $e)
```

```
if True then h else True.intro  -- unknown identifier 'h'
```

Lean 4 macros are *hygienic* ⇒ `dite` resolved in the declaration context, references in `t` in the caller context, ...

Macros!

Karlsruhe Institute of Technology

```
syntax [if] "if " optIdent term " then " term " else " term : term
```

```
@[macro «if»] def expandIf (stx : Syntax) : MacroM Syntax :=
match_syntax stx with
| `(if $h : $cond then $t else $e) ⇒ `(dite $cond (fun $h ⇒ $t) (fun $h ⇒ $e))
| `(if $cond then $t else $e)      ⇒ `(if h : $cond then $t else $e)
| _                                ⇒ throwUnsupportedSyntax
```

Macros can be arbitrary *syntax transformers*

Macros!

```
syntax [if] "if " optIdent term " then " term " else " term : term
```

```
@[macro «if»] def expandIf (stx : Syntax) : MacroM Syntax :=
match_syntax stx with
| `(if $h : $cond then $t else $e) ⇒ `(dite $cond (fun $h ⇒ $t) (fun $h ⇒ $e))
| `(if $cond then $t else $e)      ⇒ `(if h : $cond then $t else $e)
| _                                ⇒ throwUnsupportedSyntax
```

Hygiene is tied to syntax quotations, which are *monadic values*

```
class MonadQuotation (m : Type → Type) :=
(getCurrMacroScope : m MacroScope)
```

`` `(...) : m Syntax `` given `[MonadQuotation m]`

Macros!!

```
syntax [if] "if " optIdent term " then " term " else " term : term
```

```
@[termElab «if»] def elabIf : TermElab :=
adaptMacro $ fun stx ⇒ match_syntax stx with
| `(if $h : $cond then $t else $e) ⇒ `(dite $cond (fun $h ⇒ $t) (fun $h ⇒ $e))
| `(if $cond then $t else $e)      ⇒ `(ite $cond $t $e)
| _                                ⇒ throwUnsupportedSyntax

def adaptMacro (exp : Syntax → MacroM Syntax) : TermElab :=
fun stx expectedType? ⇒ do
  stx' ← exp stx;
  elabTerm stx' expectedType?

def elabTerm (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr
```

Lean 4 macros are actually just “tail-recursive elaborators”

Macros!!!

Karlsruhe Institute of Technology

```
syntax [anonCtor] "⟨" (sepBy term ", ") "⟩" : term
```

```
@[termElab anonCtor] def elabAnonCtor : TermElab :=
fun stx expectedType? ⇒ match_syntax stx with
| `(⟨$args*⟩) ⇒ do
  tryPostponeIfNoneOrMVar expectedType?;
  match expectedType? with
  | some expectedType ⇒ do
    match Expr.getAppFn expectedType with
    | Expr.const constName _ _ ⇒ do
      ctors ← getCtors constName;
      match ctors with
      | [ctor] ⇒ do
        stx ← `($(mkCTermId ctor) $(getSepElems args)*);
        elabTerm stx expectedType?
...  -- error handling
```

Macros?

1. there exists $(classRelation)_O$ [such that $(statement)_S$] [, $(notions)_T$]
   $\Rightarrow$ for some $\mathbf{n}(O)$ (O [and S] [and there exists T])
2. there exists no $(classRelation)_O$ [such that $(statement)_S$]
   $\Rightarrow$ for every $\mathbf{n}(O)$ (not (O [and S]))
3. there exists $(notion)_O$ [, $(notions)_T$]
   $\Rightarrow$ for some $\mathbf{n}(O)$ ($\mathbf{n}(O)$ is $\bar{O}$ [and there exists T])
4. there exists no $(notion)_O$
   $\Rightarrow$ for every $\mathbf{n}(O)$ ($\mathbf{n}(O)$ is not $\bar{O}$)

Demo

Conclusion

- arbitrarily extend the Lean language using a tower of abstraction levels
- extend Lean with other languages... with some preliminary caveats
    - token handling should be refined and made customizable