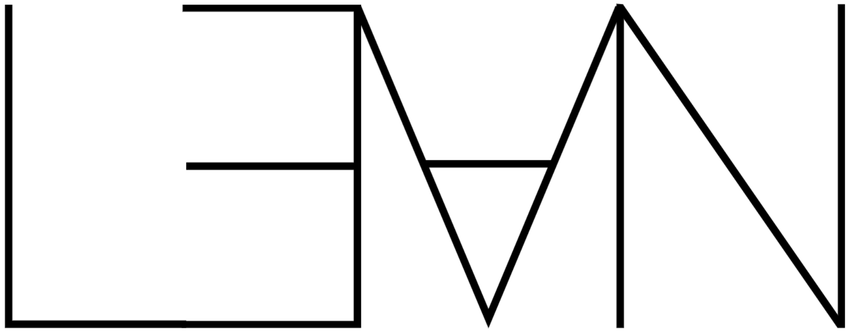


Lean 4 - metaprogramming

Leonardo de Moura - Microsoft Research
Sebastian Ullrich - Karlsruhe Institute of Technology

January 6, 2021

Lean package

Lean source code is in the Lean package and Lean namespace

You don’t need to import Lean package to use Lean

`import Lean` is required when

- Manipulating `Expr, Environment` objects
- Writing decision procedures
- Writing your own elaboration functions

Helper terms for writing macros

`let* x := v; t`

Similar to `let`, but `t` is elaborated before `v`. We use it to compile the Do-DSL

Join-points aka Goto's

`typeof! t`

```
def ex (x : Nat) : typeOf! x :=
let r : typeOf! x := x+1;
r + 2
```

`ensureTypeOf! t msg s`

```
def ex (x : Nat) : Nat :=
let y : Nat := x
»let y := ensureTypeOf! y "invalid reassignment, term" (y == 1)
y + 1
```

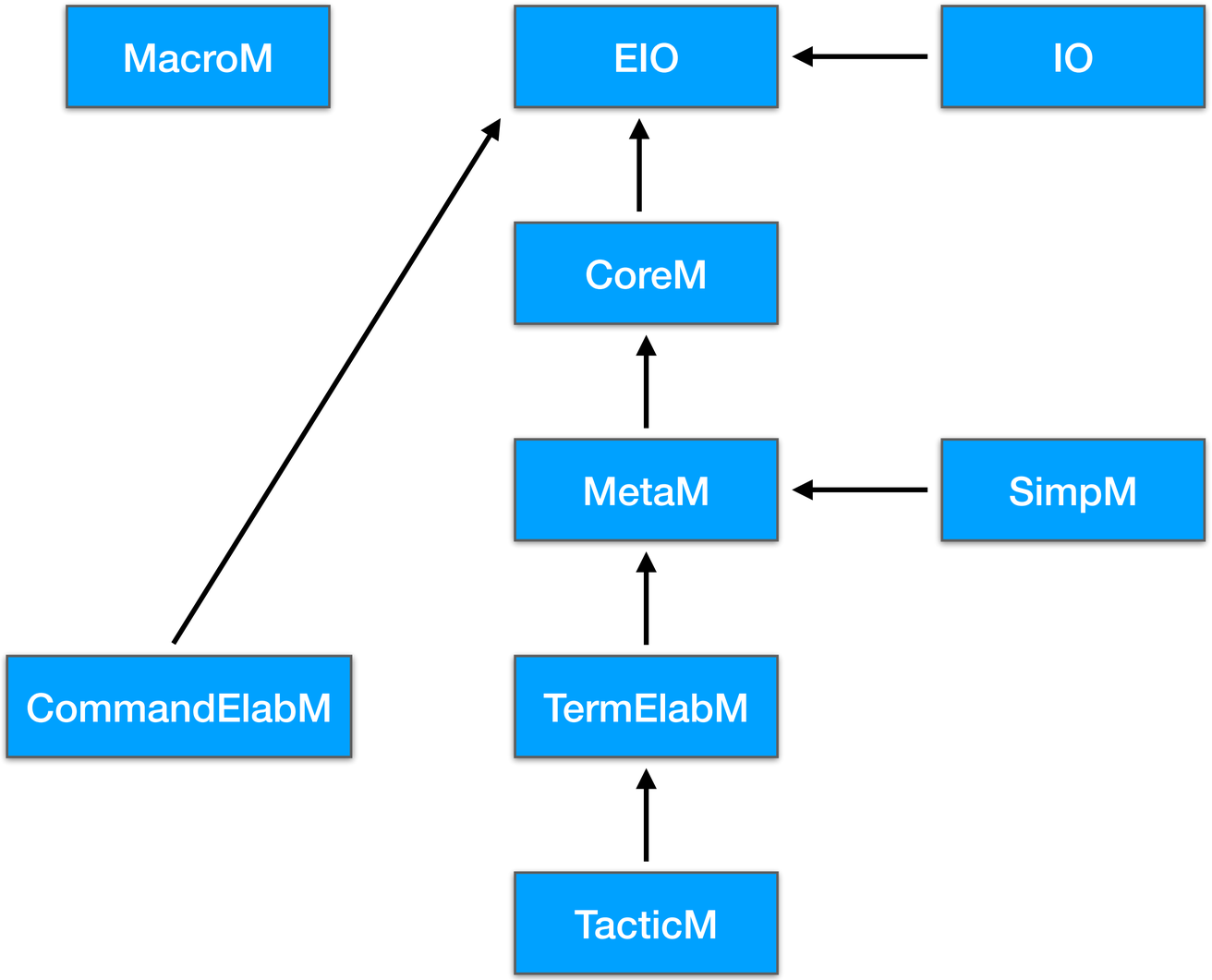
```
error: invalid reassignment, term has type
  Bool
but is expected to have type
  Nat
```

Expr

```
inductive Expr where
  | bvar    : Nat → Data → Expr                          -- bound variables
  | fvar    : FVarId → Data → Expr                       -- free variables
  | mvar    : MVarId → Data → Expr                       -- meta variables
  | sort    : Level → Data → Expr                        -- Sort
  | const   : Name → List Level → Data → Expr            -- constants
  | app     : Expr → Expr → Data → Expr                  -- application
  | lam     : Name → Expr → Expr → Data → Expr           -- lambda abstraction
  | forallE : Name → Expr → Expr → Data → Expr           -- (dependent) arrow
  | letE    : Name → Expr → Expr → Expr → Data → Expr    -- let expressions
  | lit     : Literal → Data → Expr                      -- literals
  | mdata   : MData → Expr → Data → Expr                 -- metadata
  | proj    : Name → Nat → Expr → Data → Expr            -- projection
  deriving Inhabited
```

```
inductive Literal where
  | natVal (val : Nat)
  | strVal (val : String)
  deriving Inhabited, BEq
```

Lean Monad Zoo



CoreM

```
namespace Lean
namespace Core

structure State where
  env             : Environment
  nextMacroScope  : MacroScope    := firstFrontendMacroScope + 1
  ngen            : NameGenerator := {}
  traceState      : TraceState    := {}
  deriving Inhabited

structure Context where
  options        : Options := {}
  currRecDepth   : Nat := 0
  maxRecDepth    : Nat := 1000
  ref            : Syntax := Syntax.missing
  currNamespace  : Name := Name.anonymous
  openDecls      : List OpenDecl := []

abbrev CoreM := ReaderT Context $ StateRefT State (EIO Exception)
```

```
inductive Exception where
  | error (ref : Syntax) (msg : MessageData)
  | internal (id : InternalExceptionId) (extra : KVMap := {})
```

CoreM

```
namespace Lean
namespace Core

structure State where
  env             : Environment
  nextMacroScope  : MacroScope    := firstFrontendMacroScope + 1
  ngen            : NameGenerator := {}
  traceState      : TraceState    := {}
  deriving Inhabited

structure Context where
  options        : Options := {}
  currRecDepth   : Nat := 0
  maxRecDepth    : Nat := 1000
  ref            : Syntax := Syntax.missing
  currNamespace  : Name := Name.anonymous
  openDecls      : List OpenDecl := []

abbrev CoreM := ReaderT Context $ StateRefT State (EIO Exception)
```

```
inductive Exception where
  | error (ref : Syntax) (msg : MessageData)
  | internal (id : InternalExceptionId) (extra : KVMap := {})
```

CoreM interfaces

```
instance : MonadRef CoreM where
  getRef := return (← read).ref
  withRef ref x := withReader (fun ctx => { ctx with ref := ref }) x

instance : MonadEnv CoreM where
  getEnv := return (← get).env
  modifyEnv f := modify fun s => { s with env := f s.env }

instance : MonadOptions CoreM where
  getOptions := return (← read).options

instance : AddMessageContext CoreM where
  addMessageContext := addMessageContextPartial

instance : MonadNameGenerator CoreM where
  getNGen := return (← get).ngen
  setNGen ngen := modify fun s => { s with ngen := ngen }

instance : MonadRecDepth CoreM where
  withRecDepth d x := withReader (fun ctx => { ctx with currRecDepth := d }) x
  getRecDepth := return (← read).currRecDepth
  getMaxRecDepth := return (← read).maxRecDepth

instance : MonadResolveName CoreM where
  getCurrNamespace := return (← read).currNamespace
  getOpenDecls := return (← read).openDecls
```

MetaM

```
structure State where
  mctx         : MetavarContext := {}
  cache        : Cache := {}
  zetaFVarIds  : NameSet := {}
  postponed    : PersistentArray PostponedEntry := {}
  deriving Inhabited

structure Context where
  config          : Config          := {}
  lctx            : LocalContext    := {}
  localInstances  : LocalInstances  := #[]

abbrev MetaM  := ReaderT Context $ StateRefT State CoreM
```

```
def whnf (e : Expr) : MetaM Expr
```

```
def inferType (e : Expr) : MetaM Expr
```

```
def isDefEq (t s : Expr) : MetaM Bool
```

```
def trySynthInstance (type : Expr) (maxResultSize? : Option Nat := none) : MetaM (LOption Expr)
```

```
def mkAppM (constName : Name) (xs : Array Expr) : MetaM Expr
```

MetaM example

```
import Lean

def f {α} [Add α] (x : α) : List α :=
  [x, x, x+x]

open Lean
open Lean.Meta

def test : MetaM Unit := do
  let t ← mkAppM `f #[mkNatLit 2]
  trace[Meta.debug]! "t: {t}"
  let t ← whnf t
  trace[Meta.debug]! "after whnf: {t}"
  let type ← inferType t
  trace[Meta.debug]! "type: {type}"
  let m ← mkFreshExprMVar (mkSort levelOne)
  let p ← mkAppM `List #[m]
  trace[Meta.debug]! "p: {p}"
  unless (← isDefEq type p) do throwError! "unexpected"
  trace[Meta.debug]! "p: {p}"
  trace[Meta.debug]! "m: {m}"

set_option trace.Meta.debug true
»#eval test
```

```
[Meta.debug] t: f 2
[Meta.debug] after whnf: [2, 2, 2 + 2]
[Meta.debug] type: List Nat
[Meta.debug] p: List ?m.5
[Meta.debug] p: List Nat
[Meta.debug] m: Nat
```

MetaM reduce

```
partial def reduce (e : Expr) (explicitOnly skipTypes skipProofs := true) : MetaM Expr :=
  let rec visit (e : Expr) : MonadCacheT Expr Expr MetaM Expr :=
    checkCache e fun _ => Core.withIncRecDepth do
      if (← (skipTypes <&&> isType e)) then
        return e
      else if (← (skipProofs <&&> isProof e)) then
        return e
      else
        let e ← whnf e
        match e with
        | Expr.app .. =>
          let f     := e.getAppFn
          let nargs := e.getAppNumArgs
          let finfo ← getFunInfoNArgs f nargs
          let mut args  := e.getAppArgs
          for i in [:args.size] do
            if i < finfo.paramInfo.size then
              let info := finfo.paramInfo[i]
              if !explicitOnly || info.isExplicit then
                args ← args.modifyM i visit
            else
              args ← args.modifyM i visit
          pure (mkAppN f args)
        | Expr.lam ..     => lambdaTelescope e fun xs b => do mkLambdaFVars xs (← visit b)
        | Expr.forallE .. => forallTelescope e fun xs b => do mkForallFVars xs (← visit b)
        | _               => return e
  visit e |>.run
```

```
if skipTypes && (← isType e) then
```

MetaM reduce

```
partial def reduce (e : Expr) (explicitOnly skipTypes skipProofs := true) : MetaM Expr :=
  let rec visit (e : Expr) : MonadCacheT Expr Expr MetaM Expr :=
    checkCache e fun _ => Core.withIncRecDepth do
      if (← (skipTypes <&&> isType e)) then
        return e
      else if (← (skipProofs <&&> isProof e)) then
        return e
      else
        let e ← whnf e
        match e with
        | Expr.app .. =>
          let f     := e.getAppFn
          let nargs := e.getAppNumArgs
          let finfo ← getFunInfoNArgs f nargs
          let mut args  := e.getAppArgs
          for i in [:args.size] do
            if i < finfo.paramInfo.size then
              let info := finfo.paramInfo[i]
              if !explicitOnly || info.isExplicit then
                args ← args.modifyM i visit
            else
              args ← args.modifyM i visit
          pure (mkAppN f args)
        | Expr.lam ..     => lambdaTelescope e fun xs b => do mkLambdaFVars xs (← visit b)
        | Expr.forallE .. => forallTelescope e fun xs b => do mkForallFVars xs (← visit b)
        | _               => return e
  visit e |>.run
```

MetaM intro, revert, apply tactics

```
abbrev introN (mvarId : MVarId) (n : Nat) (givenNames : List Name := []) : MetaM (Array FVarId × MVarId)

abbrev introNP (mvarId : MVarId) (n : Nat) : MetaM (Array FVarId × MVarId)

def intro (mvarId : MVarId) (name : Name) : MetaM (FVarId × MVarId)

abbrev intro1 (mvarId : MVarId) : MetaM (FVarId × MVarId)

abbrev intro1P (mvarId : MVarId) : MetaM (FVarId × MVarId)

def revert (mvarId : MVarId) (fvars : Array FVarId) (preserveOrder : Bool := false) : MetaM (Array FVarId × MVarId)

def apply (mvarId : MVarId) (e : Expr) : MetaM (List MVarId)
```

MetaM rewrite

```
def kabstract (e : Expr) (p : Expr) (occs : Occurrences := Occurrences.all) : MetaM Expr
```

```
structure RewriteResult where
  eNew     : Expr
  eqProof  : Expr
  mvarIds  : List MVarId -- new goals

def rewrite (mvarId : MVarId) (e : Expr) (heq : Expr)
    (symm : Bool := false) (occs : Occurrences := Occurrences.all) (mode := TransparencyMode.reducible) : MetaM RewriteResult
```

TermElabM

```
structure Context where
  fileName         : String
  fileMap          : FileMap
  declName?        : Option Name         := none
  macroStack       : MacroStack          := []
  currMacroScope   : MacroScope          := firstFrontendMacroScope
  mayPostpone      : Bool                := true
  errToSorry       : Bool                := true
  autoBoundImplicit  : Bool              := false
  autoBoundImplicits : Std.PArray Expr := {}

structure State where
  levelNames         : List Name       := []
  syntheticMVars     : List SyntheticMVarDecl := []
  mvarErrorInfos     : List MVarErrorInfo := []
  messages           : MessageLog := {}
  letRecsToLift      : List LetRecToLift := []
  deriving Inhabited

abbrev TermElabM := ReaderT Context $ StateRefT State MetaM
abbrev TermElab  := Syntax → Option Expr → TermElabM Expr
```

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

TermElabM : YAACN

```
import Lean

syntax (name := actor) "{| " term,*,? " |}" : term

open Lean Lean.Meta Lean.Elab Lean.Elab.Term

@[termElab actor] def elabACtor : TermElab := fun stx expectedType? =>
  match stx with
  | `({| $[$args],* |}) => do
    for ctorName in (← getCtors expectedType?) do
      let ctorInfo ← getConstInfoCtor ctorName
      if ctorInfo.nfields == args.size then
        let newStx ← `($(mkCIdentFrom stx ctorName) $(args)*)
        return (← withMacroExpansion stx newStx <| elabTerm newStx expectedType?)
    throwError! "did not find compatible constructor {args.size}"
  | _ => throwUnsupportedSyntax
where
  getCtors (expectedType? : Option Expr) : MetaM (List Name) := do
    match expectedType? with
    | none      => throwError! "expected type is not known"
    | some type =>
      match (← whnf type).getAppFn with
      | Expr.const declName _ _ =>
        match (← getEnv).find? declName with
        | ConstantInfo.inductInfo val => return val.ctors
        | _ => throwExpectedInductive type
      | _ => throwExpectedInductive type

  throwExpectedInductive {α} (type : Expr) : MetaM α :=
    throwError! "expected inductive type application{indentExpr type}"
```

TermElabM YAACN test drive

```
def test1 : List Nat :=
 {| |}

def test2 : Nat × Bool :=
 {| 1, true |}

def test3 : List Nat :=
» let x := {| 1, [] |} -- Error: expected inductive type application
 x ++ x
```

TermElabM: YAACN refinement

```
@[termElab actor] def elabACtor : TermElab := fun stx expectedType? =>
  match stx with
  | `({| $[$args],* |}) => do
    tryPostponeIfNoneOrMVar expectedType?
    for ctorName in (← getCtors expectedType?) do
      let ctorInfo ← getConstInfoCtor ctorName
      if ctorInfo.nfields == args.size then
```

```
def test1 : List Nat :=
 {| |}

def test2 : Nat × Bool :=
 {| 1, true |}

def test3 : List Nat :=
 let x := {| 1, [] |}
 x ++ x
```

TacticM

```
structure Context where
  main : MVarId

structure State where
  goals : List MVarId
  deriving Inhabited

abbrev TacticM := ReaderT Context $ StateRefT State TermElabM
abbrev Tactic  := Syntax → TacticM Unit
```

```
partial def evalTactic : Syntax → TacticM Unit
```

TacticM

```
@[builtinTactic Lean.Parser.Tactic.intro] def evalIntro : Tactic := fun stx => do
  match stx with
  | `(tactic| intro)                  => introStep `_
  | `(tactic| intro $h:ident)         => introStep h.getId
  | `(tactic| intro _)                => introStep `_
  | `(tactic| intro $pat:term)        => evalTactic (← `(tactic| intro h; match h with | $pat:term => _; clear h))
  | `(tactic| intro $h:term $hs:term*) => evalTactic (← `(tactic| intro $h:term; intro $hs:term*))
  | _ => throwUnsupportedSyntax
where
  introStep (n : Name) : TacticM Unit :=
    liftMetaTactic fun mvarId => do
      let (_, mvarId) ← Meta.intro mvarId n
      pure [mvarId]
```

CommandElabM

```
structure Scope where
  header        : String
  opts          : Options := {}
  currNamespace : Name := Name.anonymous
  openDecls     : List OpenDecl := []
  levelNames    : List Name := []
  varDecls      : Array Syntax := #[]
  deriving Inhabited

structure State where
  env            : Environment
  messages       : MessageLog := {}
  scopes         : List Scope := [{ header := "" }]
  nextMacroScope : Nat := firstFrontendMacroScope + 1
  maxRecDepth    : Nat
  nextInstIdx    : Nat := 1 -- for generating anonymous instance names
  ngen           : NameGenerator := {}
  deriving Inhabited

structure Context where
  fileName       : String
  fileMap        : FileMap
  currRecDepth   : Nat := 0
  cmdPos         : String.Pos := 0
  macroStack     : MacroStack := []
  currMacroScope : MacroScope := firstFrontendMacroScope
  ref            : Syntax := Syntax.missing

abbrev CommandElabCoreM (ε) := ReaderT Context $ StateRefT State $ EIO ε
abbrev CommandElabM := CommandElabCoreM Exception
abbrev CommandElab  := Syntax → CommandElabM Unit
abbrev Linter := Syntax → CommandElabM Unit
```

CommandElabM

```
@[builtinCommandElab Lean.Parser.Command.check] def elabCheck : CommandElab
  | `(#check%$tk $term) => withoutModifyingEnv $ runTermElabM (some `_check) fun _ => do
    let e ← Term.elabTerm term none
    Term.synthesizeSyntheticMVarsNoPostponing
    let (e, _) ← Term.levelMVarToParam (← instantiateMVars e)
    let type ← inferType e
    unless e.isSyntheticSorry do
      logInfoAt tk m!"{e} : {type}"
  | _ => throwUnsupportedSyntax
```

DEMO